The ArchC Language Support & Tools for Automatic Generation of Binary Utilities

version 2.1

User Manual

The ArchC Team
http://www.archc.org

July 2011

Av. Albert Einstein, 1251 13084-971
PO Box 6176 - Campinas/SP - Brazil

Contents

Chapter 1

Introduction

This manual presents the ArchC language support and tools for the generation of binary utilities, such as assemblers, disassemblers, linkers and debuggers. In this introductory chapter we give a brief overview of the binary utilities generation process, present a quick start guide and discuss current limitations.

1.1 Overview

Figure 1.1 shows the generation flow of binary utilities. Users start by describing the required information at a high abstraction level using the ArchC architecture description language (step (a)). The available constructs and support files used in this step are subject of Chapter 2. The generation tool reads the target model and support files in order to create the binary utilities (step (b)). Chapter 3 explains how to use and the command line options for the generation tool. Starting with an assembly source code, the assembler and linker produce the corresponding executable object code (step (c)). Disassemblers and debuggers can inspect object code and help finding bugs in the original program (step (d)). Chapter 3 also explains how to use these generated tools.

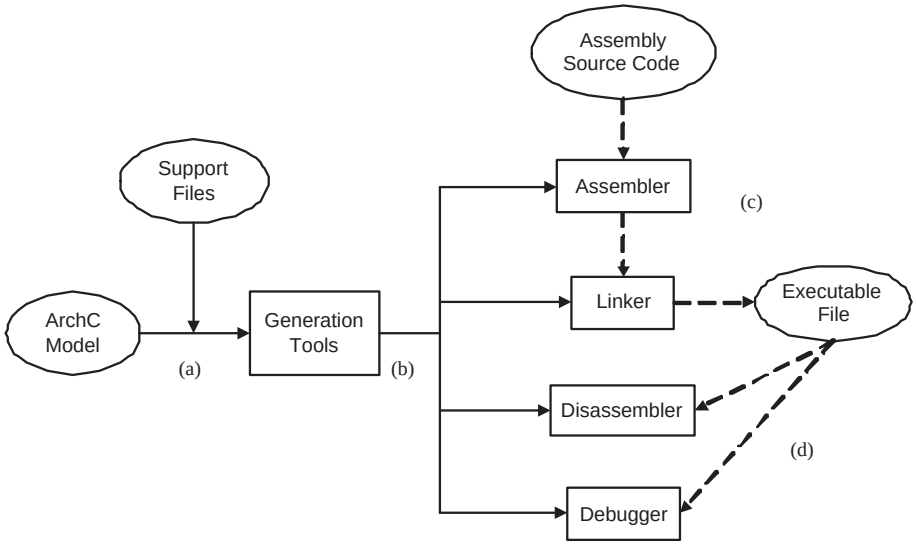

Figure 1.1: Generation Flow

1.2 Quick Start

This quick start guides you through the generation of binary tools for the MIPS architecture. First, create a directory named `quickstart` anywhere in your local home and download the following items to that directory:

- `ArchC` – ArchC language and tools (`www.archc.org`)
  Name used: `archc-v2.1.tgz`
- `MIPS` – MIPS model (`www.archc.org`)
  Name used: `mips1-v0.7.8.tgz`
- `Binutils` – GNU Binutils source code (`www.gnu.org/software/binutils/`)
  (latest version tested: 2.15)
  Name used: `binutils-2.15.tar.gz`
- `Gdb` – The GNU Project Debugger source code (`sourceware.org/gdb/`)
  (latest version tested: 6.4)
  Name used: `gdb-6.4.tar.gz`

Note that the lines with `Name used` identify the package names we will use in this quick start. You should replace them with the name of the packages you downloaded.

Now unpack the packages inside the `quickstart` directory:

```
$ tar zxvf archc-v2.1.tgz
$ tar zxvf mips1-v0.7.8.tgz
$ tar zxvf binutils-2.15.tar.gz
$ tar zxvf gdb-6.4.tar.gz
```

To compile the ArchC package, enter the ArchC directory you have just unpacked and issue the following command:

```
$ ./configure --with-binutils=‘pwd‘/../binutils-2.15 \
              --with-gdb=‘pwd‘/../gdb-6.4 \
              --prefix=‘pwd‘
$ make
$ make install
```

This will install the binary utility generation script into the `bin` subdirectory. Now change to the model directory and generate the binary tools:

```
$ cd ../mips-v0.7.8/
$ ../archc-v2.1/bin/acbingen.sh -amips1 -i‘pwd‘/../bin/ mips1.ac
```

This process may take from several minutes to hours depending on your host machine. The binary utilities will be created and placed into the directory `quickstart/bin`.

1.3 Latest Changes

New features in this version:

Improved parser generator

Now it is possible to describe an instruction syntax containing multiples mnemonic suffixes. This is important in assembly languages whose mnemonics presents many variations, denoted by a different suffix. Suffixes can be handled as any other parameter. In fact, as a special one, with no spaces between it and the mnemonic.

Better support for user defined maps

The user can now define distinct maps whose symbols names may be the same. This enables a powerful technique in describing languages, namely, defining nullable symbols. By creating a map containing the null (empty string) symbol, the user specifies that these symbols may be omitted. Previously, this still could be done, but only in one map per model, since symbols with the same name weren't allowed in the same model.

Concatenated fields

There is no more limitation to the number of fields assigned to a specific operand. The user may concatenate any number of fields using the + operator.

Defining comment characters

There is no more need to change the assembler generated source file in order to define different comment characters. Now there is a specific directive to specify assembler comment characters directly in the model file.

Lists of symbols

The natural way of describing instructions syntax in an ArchC model is using one operand identifier per instruction operand. This operand identifier is linked to a particular translation method (from assembly language source code to binary representation in the instruction field), and if the operand identifier is specified by the user through user defined maps, uses the symbol-value list provided by the map. Using this paradigm, it is not easy to define a parameter that represents a list (this usually occurs as variable-sized list of registers). Now there is a specific mechanism to describe lists as operands.

Dynamic linking support

The GNU libbfd generated backend now offers support for shared library creation, as well as linking a regular object file against a shared library. The new ArchC simulator also supports loading ELF executable which depends on a shared library.

1.4 Current Limitations

The following limitations exist:

Output object file
: by default, the generated assembler and companion tools all handle only ELF object files. Thus, if you use the generated assembler to produce any binary, the output object file format will be ELF.

Chapter 2

Language Support

In this chapter we present the ArchC language constructs and support files required for the generation of binary utilities. We show examples from several microprocessors to illustrate how each of the constructs are used.

2.1 Overview

There is a total of 10 ArchC constructs related to the binary utilities: `set_endian`, `ac_format`, `ac_instr`, `set_decoder`, `ac_asm_map`, `set_asm`, `pseudo_instr`, `assembler.set_comment`, `assembler.set_line_comment` and `map_to`. While the first 4 of them are also used by the simulator generator tool, the last 6 are only used by the binary utility generator. For further information about `set_endian`, `ac_format`, `ac_instr` and `set_decoder`, please refer to the ArchC language reference manual [].

In the rest of this chapter we present the constructs `ac_asm_map`, `set_asm`, `pseudo_instr`, `assembler.set_comment`, `assembler.set_line_comment` and `map_to` which describe machine-dependent aspects of binary utilities at a high abstraction level. All of them must be described in the `AC_ISA` part of an ArchC processor model. Some of them may require further description which is done through support files, outside the ArchC model. Support is provided for processor-specific assembly language symbols (such as register names), syntax and operand encoding. The user can also describe synthetic instructions.

2.2 Assembly language symbols

Assembly language-level symbols and their corresponding values are defined in ArchC through the `ac_asm_map` construct. This construct groups a set of symbol-value pairs under a common name, which can be later used to specify the assembly language syntax.

The most common use of `ac_asm_map` is to map processor's register names to their encoding values. For example, Figure 2.1 shows the MIPS-I register names mapping. Line 1 declares `reg` as the mapping identifier. Lines 2 to 9 define each symbol and the corresponding encoding value. A symbol is specified between quotation marks at the left side, followed by the equal sign (`=`), its value and a semicolon (`;`). It is possible to specify a range of values by using the square brackets notation (`[]`).

For instance, line 5 maps symbols `kt0` and `kt1` to values `26` and `27`, respectively. Note that it is also possible to assign the same value to different symbols, as in lines 2 and 3 (`$0` and `$zero` map to `0`).

```
1 ac_asm_map reg {
2     "$"[0..31] = [0..31];
3     "$zero" = 0;
4     "$at" = 1;
5     "$kt"[0..1] = [26..27];
6     "$gp" = 28;
7     "$sp" = 29;
8     "$fp" = 30;
9     "$ra" = 31;
10 }
```

Figure 2.1: MIPS-I register names and encoding values

2.3 Assembly language syntax and instruction encoding

Every ArchC instruction (declared with `ac_instr`) provides two properties to define its assembly syntax and binary encoding: `set_asm` and `set_decoder`. The former specifies both assembly language syntax and operand encoding, while the latter specifies the opcode. For further information about `set_decoder`, please refer to the ArchC language reference manual []. We concentrate here in describing the `set_asm` construct.

Figure 2.2 shows the general form of `set_asm`. Here, `insn` is an ArchC instruction whose assembly syntax and operand encoding are being defined. The construct is split into a *syntax string* and an optional *operand list*. The syntax string (`"mno %op1, %op2"`) is made up of literal characters (`mno`, `,`) and operand identifiers (`%op1`, `%op2`). The set of characters up to the first white space constitutes the instruction mnemonic (`mno`). Operand identifiers are specified with the special character `%` and act as placeholders for binary values assigned at assembling and/or linking time. The place in the instruction where these values are encoded is specified in the operand list. For each operand identifier there must be a corresponding operand field. In the given example, the value hold by `op1` will be placed in instruction field `field1`, while the value hold by `op2` will be placed in instruction field `field2`.

2.3.1 Assembly syntax

The ArchC language specifies three types of operand identifiers: (1) `imm`, used for immediate integer-like operands; (2) `addr`, used for symbolic operands; and (3) `exp`, used for expressions (a combination of immediate and symbolic operands). Additional operand types can be declared via `ac_asm_map`, as seen in section 2.2. Consider, for instance, the declarations showed in Figure 2.3. In this example we are using the identifier `reg` as defined in Figure 2.1. Line 1 shows the syntax for instruction `lw` with 3 operands: a `reg`, an `imm` and another `reg`. They are bound to the instruction fields `rt`, `imm` and `rs`, respectively. Line 2 shows an instruction whose operands are all registers, whereas line 3 has an operand of type `exp`.

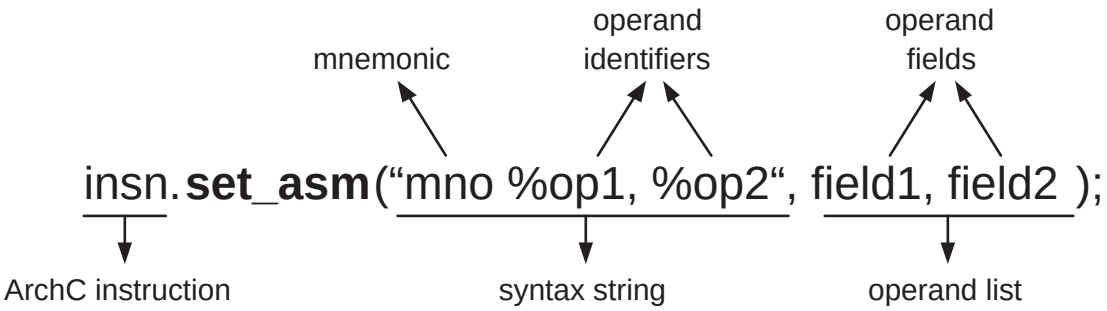

Figure 2.2: Generation Flow

```
1 lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
2 add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
3 addi.set_asm("addi %reg, %reg, %exp", rt, rs, imm);
```

Figure 2.3: Describing the MIPS-I assembly language syntax

The `set_asm` construct specifies how the generated assembler will parse an assembly source code file and emit the binary code. If the definitions showed in Figure 2.3 are used, the generated assembler will correctly recognize the instruction syntax showed in Figure 2.4. Note that the syntax of the instructions match their definitions presented in Figure 2.3. For instance, the last operand of instruction `addi` (line 3) is an expression comprised of a pre-defined symbol (`_start`) and an integer (`10`).

```
1 lw   $3, 10($11)
2 add  $sp, $gp, $0
3 addi $2, $30, _start + 10
```

Figure 2.4: Valid instruction syntaxes

Symbol disambiguation

While defining the operand list in `set_asm` constructs, it is possible to bind a field to a fixed value (constant). This is presented in the code below.

```
1 add.set_asm("add %reg %reg", op1, op2, op3="r01");
```

Note that field `op3` is not bound to any specific `add` operand, but to a constant. The constant can be any number representing the value to be codified in this field, but can also be an user defined symbol. This is the case in the presented code, as `op3` is assigned to the value of symbol `r01`.

Suppose the symbol `r01` exists in two different maps. Now, which value of `r01` will be chosen to be codified into field `op3`? To avoid these situations, it is recommended to use the `map_to` construct:

```
1 add.set_asm("add %reg %reg", op1, op2, op3=reg.map_to("r01"));
```

Using this code, `op3` can only be assigned to the value of the symbol `r01` as described in the map `reg`.

2.3.2 Instruction encoding

Consider now the instruction encoding. This process is performed primarily by the assembler and optionally by the linker (if relocation is present). To understand the encoding behavior, first consider one of the instruction formats of the MIPS-I showed in Figure 2.5 (commonly known as I-type). The first 6 bits (`op`) comprise the instruction opcode field. The remaining 3 fields are the operand fields, named `rs`, `rt`, and `imm`, respectively. This format is used to specify the operand encoding for the instructions `lw` and `addi` in Figure 2.3.

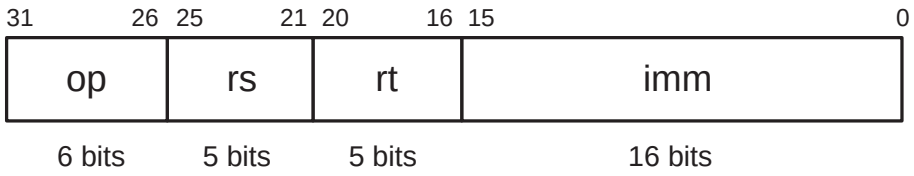

Figure 2.5: MIPS-I instruction format (I-type)

To understand how the assembler emits binary code, consider the instruction description in line 1 of Figure 2.3 and an instance of this instruction as showed in line 1 of Figure 2.4. The assembler first recognizes the `lw` instruction and attempts to encode its operands. The first operand found is the register `$3` which has encoding value 3. The assembler converts it to a 5-bit unsigned value (00011) and place it into the `rt` field (bits 6 to 10). In the same way, the second operand (10) and the third operand (`$11`) are placed into fields `imm` and `rs`, according to the encoding description in line 1 of Figure 2.3. The final binary code emitted by the assembler is showed in Figure 2.6. Part (a) shows the ArchC description, part (b) shows an instruction instance which matches the ArchC description, and part (c) shows the corresponding binary code emitted by the assembler.

2.3.3 Modifiers

The encoding scheme presented in section 2.3.2 is the default encoding behavior. It handles the common case, but it may not suffice if a transformation is to be applied to an operand value before encoding takes place. Such a case happens with pc-relative operands, where the encoding value is the result of the subtraction of the instruction address (probably added to an offset) from the symbol value. To deal with non-conventional cases, ArchC introduces the notion of *modifiers*. A modifier is a function that transforms a given operand value. If a modifier is specified, the assembler and/or linker first executes the modifier code using the original operand value as input. The modifier output is then used as the encoding value.

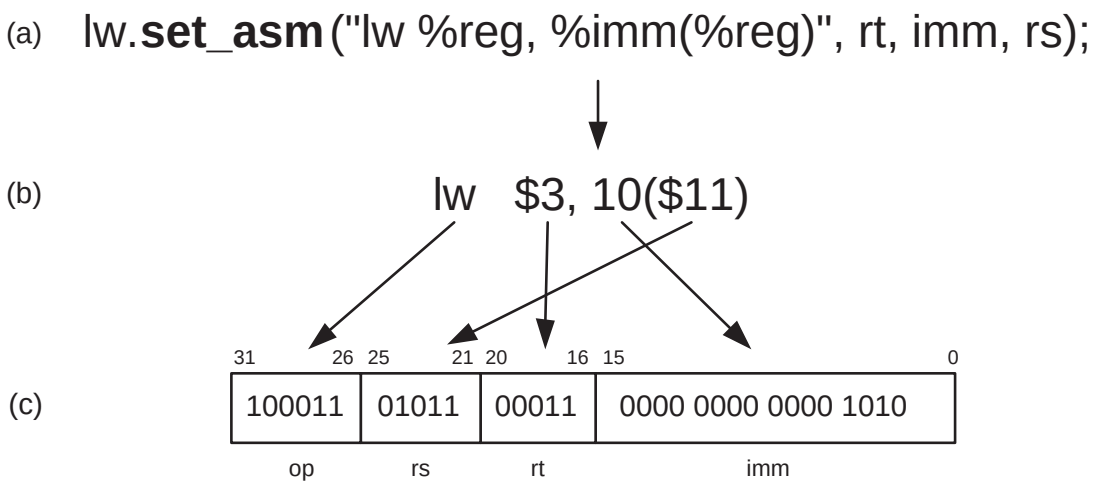

Figure 2.6: Operand encoding: (a) ArchC description, (b) instruction instance, (c) emitted binary code

In ArchC, a modifier can be attached to any operand identifier. All you have to do is to specify a modifier name and an optional addend after the operand identifier between parentheses. For instance, the following description:

```
1 ba.set_asm("ba %exp(pcrel)", disp22, an=0);
```

specifies the SPARC instruction `ba` (branch always) with an operand of type `exp`. A modifier named `pcrel` is assigned to this operand, meaning that the operand value must be transformed by the `pcrel` modifier. The modifier code is specified outside the ArchC model, in a file named `modifiers` living in the same directory as the ArchC source files. Two versions need to be specified: one for encoding (used by the assembler and linker) and another for decoding (used by the disassembler and debugger). The code is described in the C language.

Figure 2.7 shows the description of the `pcrel` modifier. The keywords `ac_modifier_encode` and `ac_modifier_decode` are used to specify the encoding and decoding modifiers, respectively. The name of the modifier must follow the keywords inside parentheses (lines 1 and 6). At least 4 special variables are defined within the modifier context, accessed through the `reloc` pointer: `input` contains the operand value; `address` contains the instruction address at assembling or linking time; `addend` contains an optional value defined as part of the modifier (not used in the SPARC example); and `output` contains the modifier's result. In line 3 of Figure 2.7 you can see the C code for the encoding modifier `pcrel`. The encoding value (`reloc->output`) is computed by subtracting the current instruction address (`reloc->address`) from the symbol value (`reloc->input`). Since the value is stored in words (4 bytes), an additional shift to the right by 2 must be performed (line 3). The decoding modifier is similarly defined in line 8.

To illustrate the use of addends, consider the pc-relative instructions of the i8051 architecture. It is somewhat similar to the SPARC instructions but they also add an offset in the calculation expression. Some instructions requires adding 2, others adding 3 or 4 (depending on the instruction size). Figure 2.8 shows how such instructions are described in ArchC. Note in line 1 that the `pcrel` modifier is followed by the number 2 and in line 2 by number 3. These addends can be accessed later in the modifier code by using `reloc->addend` as illustrate in line 3 of Figure 2.9. The variable will automatically be assigned to 2 or 3 according to the instruction being encoded.

```
1 ac_modifier_encode(pcrel)
2 {
3   reloc->output = (reloc->input - reloc->address) >> 2;
4 }
5
6 ac_modifier_decode(pcrel)
7 {
8   reloc->output = (reloc->input << 2) + reloc->address;
9 }
```

Figure 2.7: Modifier code (SPARC)

```
1 jc.set_asm("jc %addr(pcrel,2)", byte2);
2 jb.set_asm("jb %imm,%addr(pcrel,3)", byte2, byte3);
```

Figure 2.8: Modifier addend (i8051)

Modifiers can represent complex encoding schemes. You can also have direct access to the instruction formats and fields (declared with `ac_format`) inside a modifier. This will come in handy if multiple fields must have their values set, since a single `output` variable will not suffice. As an example, consider the immediate data processing operands of the ARM architecture. One single operand may have multiples encoding values and must be encoded into two different instruction fields. The declaration of such an instruction will be as follows:

```
1 and3.set_asm("and %reg, %reg, #%imm(aimm)", rd, rn, rotate+imm8);
```

Note that the third operand (`imm`) is bound to the pre-defined fields `rotate` and `imm8` (the symbol `+` is used here for field concatenation). The modifier `aimm` is attached to the operand identifier and its code is presented in Figure 2.10. Note that variables and common C structures such as loops (line 8) can be used inside the modifier. Since the encoding affects 2 fields, a single `output` variable is not sufficient. The code hence accesses the instruction formats and fields directly (lines 10 and 11).

```
1 ac_modifier_encode(pcrel)
2 {
3   reloc->output = (reloc->input - (reloc->address + reloc->addend));
4 }
```

Figure 2.9: Using the addend (i8051)

```
1  ac_modifier_encode(aimm)
2  {
3    unsigned int a;
4    unsigned int i;
5
6  #define rotate_left(v, n) (v << n | v >> (32 - n))
7
8    for (i = 0; i < 32; i += 2)
9      if ((a = rotate_left (reloc->input, i)) <= 0xff) {
10       reloc->Type_DPI3.rotate = i >> 1;
11       reloc->Type_DPI3.imm8 = a;
12       return;
13     }
14
15   reloc->error = 1;
16 }
```

Figure 2.10: Complex modifier code (ARM)

2.3.4 Lists of symbols

The natural way of describing instructions syntax in an ArchC model is using one operand identifier per instruction operand. This operand identifier is linked to a particular translation method (from assembly language source code to binary representation in the instruction field), and if the operand identifier is specified by the user through user defined maps, uses the symbol-value list provided by the map. Using this paradigm, it is not easy to define a parameter that represents a list (this usually occurs as variable-sized list of registers). Now there is a specific mechanism to describe lists as operands. We first present it in the following example:

```
1 ldm.set_asm("ldm%cond %reg, {%reg...(listmodifier)}", cond, reg1, listreg);
```

This is a simplified syntax for ARM's multiple data transfer instructions. They may be load or store instructions, and are useful for saving or loading many registers at once using the stack, as in functions prologues and epilogues. As operands, these instructions take a list of registers subject to transfer (to or from memory). The list is of arbitrary size, so the problem fits the solution provided by our description of lists of symbols.

The second operand of the `ldm` instruction in our example uses the lists of symbols feature. Like modifiers, it uses a function's name enclosed in parenthesis, but additionally has ellipsis (`...`) before the first parenthesis, indicating there may be more than one operand of the type `reg`. These elements of type `reg` are separated by commas. There may be also hyphen in order to represent range. The following instruction matches the syntax described by our example:

```
1 ldm r1, {r0, r3, r5-r9}
```

Now, the assembler can parse such syntaxes. But how these lists elements will be codified into the designated field (in our example, the list is bound to the field `listreg`)? This is the purpose of the modifier associated with the list (`listmodifier` in the example). This modifier must be capable of walking through the list of elements and progressively codify the field. To enable the user to write such code, the user may access the list using the variable `reloc->list_results`, and to extract information of the list, these functions may be used (all of them uses the `reloc->list_results` as parameter, by value or by reference, depending on the operation):

`list_results_has_data` `(list_op_results list)` uses its parameter by value. It returns 1 if there are still elements to be checked in this list, 0 otherwise.

`list_results_next` `(list_op_results *list)` uses its parameter by reference (you must use the operator & to obtain the reference). It returns the value of the current element as unsigned int, and advances to the next.

`list_results_get_separator` `(list_op_results list)` uses its parameter by value. It returns the character (char) used to separate this element from the next (remember it may be comma or hyphen).

`list_results_store` `(list_op_results *list, unsigned value)` uses its parameter by reference. It should be used to perform the reverse operation of reading, to create a list based on a field value. This is used in `ac_modifier_decode` constructs. It creates an element into the list. This function is an exception, as it receives one additional parameter (unsigned int) to specify the value of the element. Does not return any value.

It is interesting to study the code presented in Figure 2.11. It presents the complete list modifier for the ARM model, which reads a list of registers, outputting to a 16-bit field. Each bit represents the presence of a register in the list (`r0` in bit 0, `r1` in bit 1, etc...).

```
1 /* Multiple data transfer register list element - codifies
2  * a register number into a bit in the register list.
3  * Uses the list operator to obtain list of registers parsed.
4  */
5 ac_modifier_encode(listmodifier)
6 {
7   int init_range = -1;
8   unsigned i = 0;
9
10  while (list_results_has_data(reloc->list_results))
11  {
12    char separator = list_results_get_separator(reloc->list_results);
13    unsigned int result = list_results_next(&(reloc->list_results));
14    if (init_range != -1)
15    {
16      for (i = init_range; i <= result; i++)
17              reloc->output = reloc->output | (1 << i);
18    }
19    else
20      reloc->output = reloc->output | (1 << result);
21    init_range = -1;
22    if (separator == '-')
23      init_range = result;
24  }
25 }
26
27 ac_modifier_decode(listmodifier)
28 {
29  unsigned i = 0;
30  unsigned val = reloc->input;
31
32  for (i = 0; i < 16; i++)
33  {
34    unsigned aux = val >> i;
35    if (aux & 1) {
36      list_results_store(&(reloc->list_results), i);
37    }
38  }
39 }
```

Figure 2.11: Modifier code for handling lists of symbols (ARM)

2.3.5 Syntax overload

The `set_asm` construct also allows one to assign multiples syntaxes to the same ArchC instruction. It is useful if an instruction has different syntaxes for its operands. Figure 2.12 shows an example taken from the SPARC model. Lines 1 to 4 shows four different syntaxes assigned to the ArchC instruction `ldi` (SPARC load immediate). It is also possible, as showed in line 4, to explicitly define operand values. In that case, the `rs1` field was given the default value of `%g0`, and one of the operands between the brackets was suppressed (the register one).

When two or more syntax definitions are ambiguous (a given instruction matches two or more definitions), the assembler uses the definition specified earlier in the source code. Therefore, the order in which the definitions are specified in the source file is important.

```
1 ldi.set_asm("ld [%reg + \%lo(%expL10)], %reg", rs1, simm13, rd);
2 ldi.set_asm("ld [%reg + %imm], %reg", rs1, simm13, rd);
3 ldi.set_asm("ld [%imm + %reg], %reg", simm13, rs1, rd);
4 ldi.set_asm("ld [%imm], %reg", simm13, rd, rs1="%g0");
5
6 addi.set_asm("add %reg, \%lo(%expL10), %reg", rs1, simm13, rd);
7 addi.set_asm("add %reg, %imm, %reg", rs1, simm13, rd);
```

Figure 2.12: Syntax overloading (SPARC)

Simple pseudo instructions can also be defined through the `set_asm` construct. Figure 2.13 gives an example of this use for some instructions of the SPARC-V8 architecture. Line 1 of Figure 2.13 shows the syntax of the instruction `or`, while lines 2 and 3 defines the pseudo instructions `clr` and `mov` based on it. Lines 5, 6 and 7 show other examples of simple pseudo instruction declarations. They are declared by explicitly setting some of the instruction field to a default value. For example, the `mov` pseudo instruction of line 3 is an `or` instruction with the first register (`rs1` field) set to the value 0.

```
1 or_reg.set_asm("or %reg, %reg, %reg", rs1, rs2, rd);
2 or_reg.set_asm("clr %reg", rs1="%g0", rs2="%g0", rd);
3 or_reg.set_asm("mov %reg, %reg", rs1="%g0", rs2, rd);
4
5 jmpl_reg.set_asm("jmpl %reg + %reg, %reg", rs1, rs2, rd);
6 jmpl_reg.set_asm("jmp %reg + %reg", rs1, rs2, rd="%g0");
7 jmpl_reg.set_asm("call %reg + %reg", rs1, rs2, rd="%o7");
```

Figure 2.13: Simple pseudo instruction definitions (SPARC)

Mnemonic suffixes

Section 2.3 on page 10 explains the mnemonic string as all characters in the instruction syntax up to the first white space. Albeit true, some assembly languages presents instructions with the same mnemonic but many variations, typically adding a suffix. In fact, all normal ARM instructions (not Thumb or some

other special cases) have at least 16 different possible suffixes, one for each condition code. ARM uses this mechanism which bounds the instruction execution conditionally to the environment status. Thus, consider the `add` instruction. ARM assembly language also recognizes `addeq` which is valid only when the last successful comparison instruction returned equal, or `addgt`, `addle`, and so on.

This condition code, in ARM, is codified into a 4-bit field `cond`, and it is not necessary to write one instruction syntax for each one of the 16 different suffixes (through syntax overload). The user may define a map (Section 2.2 on page 9) whose symbols names are the different condition codes, in isolation (`eq`, `ne`, etc...) and values corresponding to its codification into `cond` field. When writing the instruction assembly syntax, it is possible to define an operand identifier even before any white spaces, composing the mnemonic. Figure 2.14 illustrates this concept through code. The code will provide enough language parsing information to generate an assembler capable of recognizing instructions like in Figure 2.15.

Figure 2.14 also shows a different form of defining operand identifiers, in brackets. Although operand identifiers commonly may be perfectly described without brackets, its use is important when it is necessary to isolate the operand identifier from other parsing strings, like the remaining `s` at `add`'s second syntax. This is done for illustrative purposes, as this `s` could be described as a second mnemonic suffix, using a map with two symbols (string `"s"` and empty string `""`). As the alert reader will already have observed, this also illustrates an use of nullable symbols (empty string as symbol name in the map), an interesting tool when applied to mnemonic suffixes.

```
1  ac_asm_map reg {
2    "r"[0..15] = [0..15];
3  }
4  ac_asm_map cond {
5    "eq" = 0;
6    "ne" = 1;
7    "cs", "hs" = 2;
8    "cc", "lo" = 3;
9    /* continues ... */
10   "" = 15;
11 }
12
13 ISA_CTOR(example) {
14   add.set_asm("add%cond %reg %reg %reg", cond, op1, op2, op3, s=0);
15   add.set_asm("add%[cond]s %reg %reg %reg", cond, op1, op2, op3, s=1);
16   add.set_decoder();
17 }
```

Figure 2.14: Example of mnemonic suffixes specification, just like any other regular instruction operand

2.4 Synthetic Instructions

Synthetic instructions (aka pseudo instructions) are created based on another previously defined native instructions. ArchC provides the `pseudo_instr` construct for the definition of pseudo instructions.

```
1 addeq r1, r0, r5
2 addcc r15, r4, r5
3 add r0, r1, r3
4 addlos r0, r0, r2
```

Figure 2.15: Example of instructions recognized by generated assembler of Figure 2.14

The first step in describing a synthetic instruction is to declare its syntax. Note that only the syntax string is necessary. The operand field is not specified since the pseudo instruction does not have *real* fields. Following the syntax string, a list of native instructions (those defined with `set_asm`) is specified. Parameters from the pseudo instruction syntax can be used by the native ones by employing the `%` character and a number indicating which parameter from the pseudo must be replaced (similar to the `macro` construct used by GNU assemblers).

Figure 2.16 shows two definitions of synthetic instructions used in the MIPS model. The first one, lines 1 to 4, creates the pseudo instruction `ble` which uses 3 operands. It is defined based on two native instructions (lines 2 and 3): `slt` and `beq`. The character `%` indicates a substitution of parameters. For example, the instruction `slt` in line 2 uses the literal `$at` as the first operand, the second (`%1`) is the string associated with the second `%reg` in the pseudo instruction definition, and the third operand (`%0`) is associated with the first pseudo instruction operand.

```
1  pseudo_instr("ble %reg, %reg, %exp") {
2    "slt $at, %1, %0";
3    "beq $at, $zero, %2";
4  }
5
6  pseudo_instr("mul %reg, %reg, %imm") {
7    "addiu $at, $zero, %2";
8    "mult  %1, $at";
9    "mflo  %0";
10 }
```

Figure 2.16: Defining synthetic instructions (MIPS)

The second synthetic instruction definition, lines 6 to 10, creates the instruction `mul` with 3 operands. When an instruction such as `mul $2, $3, 10` is found by the generated assembler, it will be expanded into the following three:

```
1 addiu $at, $zero, 10
2 mult  $3, $at
3 mflo  $2
```

2.5 Comment Characters

There are two sets of characters representing special characters in the assembly source code. They are comment characters delimiters (anything after this character to the end of the line is ignored) and line comment characters (when put at the beginning of the line, the whole line is ignored).

If not specified, the generated assembler will use # and ! as comment characters and # as line comment character. As an example, consider the ARM assembler. Not only its comment character is different, but using # will corrupt the assembler, since any immediate operand must be prefixed with #. In fact, # may be used as comment character in ARM assembly source codes, but only at the beginning of a line (ignoring the whole line). To specify your own comment character sets, use `assembler.set_comment` and `assembler.set_line_comment` constructs. They are used under `ISA_CTOR` scope, like `set_asm` and `set_decoder`. To complete our example, see how this is done in the ARM model:

```
1 ISA_CTOR(xscale) {
2   /* Defining assembler-specific constraints */
3   assembler.set_comment("@");
4   assembler.set_line_comment("@#");
5
6   /* Instructions syntax descriptions goes next ... */
7   ...
8 }
```

Figure 2.17: Defining assembler comment character sets in the ARM model

Chapter 3

Binary Utility Generation

This chapter describes the binary utilities generation process and how to use the generated tools. The binary utility generator originally runs on a GNU/Linux compatible system (successful installation on Cygwin system has also been reported). Before starting, make sure to have at least the following packages and their corresponding versions installed on your system:

- Bison 2.1
- Flex 2.5.4
- GCC 3.4.6
- GNU Binutils 2.15 source code
- GNU Gdb 6.4 source code

3.1 Generation Process

There are three main steps required to generate the binary utilities, assuming a processor model is already finished:

1. Build the ArchC package;
2. Generate the binary tools source code through the binary utility generator tool;
3. Build the binary utility tools.

3.1.1 Building the ArchC package

First get the latest version of the ArchC package on our website (`www.archc.org`) and the source code for the binutils and gdb (if you intend to generate debuggers). Unpack the packages on a directory of your choice. To ease the explanation, let's say the following shell variables (`bash`) have being defined:

`BINUTILSDIR` – directory where the binutils source code has been unpacked;

`GDBDIR` – directory where the gdb source code has been unpacked (gdb is optional);

`DESTDIR` – directory where the binary tools should be generated;

`ACDIR` – directory where the ArchC source code has been unpacked.

To define a shell variable on `bash`, use the `export` command. For instance:

```
$ export BINUTILSDIR=/home/myuser/binutils
```

will define the `BINUTILSDIR` variable to be `/home/myuser/binutils`.

The ArchC package uses the well-known GNU autotools framework. To compile it, you need to issue the following commands:

```
$ $ACDIR/configure --with-binutils=$BINUTILSDIR --with-gdb=$GDBDIR
$ make
$ make install
```

This will install the ArchC package on `/usr/local` by default. To change the target directory you can use the `--prefix` command of the `configure` utility. Inside the target directory there will be a subdirectory named `bin` where the generator tools will be placed. Make sure to include it on your path so that the binary files can be used in the next steps.

3.1.2 Generating the binary utility source code

This process will use the binary utility generator to create the binary utilities source code and insert them into the binutils source tree where they can be compiled. The script which automates this process is named `acbingen.sh` and is installed by the ArchC package as described in section 3.1.1.

Figure 3.1 shows the command line arguments for the `acbingen.sh` script, showed if option `-h` is used. The only required argument is the ArchC main source code file name. You can give the architecture a specific name with the `-a` option. Note that the architecture name must be unique. If the name is already used inside the binutils package the script will show a warning message and ask for permission before proceeding with the installation. The `-i` option can be used to force the script to build the binary tools. If it is not used, you need to do it manually as explained in section 3.1.3. With the `-c` option, the script only generates the source files, but does not attempt to copy them to the binutils tree and compile. This option is mainly used for debug purposes.

3.1.3 Building the binary utilities

If the option `-i` was not used in the `acbingen.sh` script as explained in section 3.1.2, you still need to build the binary tools. This process is similar to building any other binary tools from the binutils package, meaning that will use the autotools framework. The next commands perform the required action:

```
$ $BINUTILSDIR/configure --prefix=$DESTDIR --target=<arch-name>
$ make
$ make install
```

```
Usage: acbingen.sh [options] <model-file>

Create binary utilities source files and optionally build them.

Options:
 -a<name>    sets the architecture name (if omitted, it defaults to
             <model-file> without the extension)
 -i<dir>     build and install the binary utilities in directory <dir>
             NOTE: <dir> -MUST- be an absolute path
 -c          only create the files, do not copy to binutils tree
 -h          print this help
 -v          print version number

Report bugs and patches to ArchC Team.
```

Figure 3.1: `acbingen.sh` command line options

Note that `arch-name` is the name you gave to the architecture. This can either be the ArchC model name or a specific name passed through the `-a` option to the generator script. The same process must be repeated for the gdb if its generation is required.

3.2 The generated tools

The tools generated by ArchC are standard binutils and gdb tools. This means that the machine independent command line options supported by conventional tools are still supported by the generated tools. The generated assembler also extends the command line options with the following:

`-i`, `--insensitive-syms`
: the assembler considers symbolic names as being case insensitive;

`-s`, `--sensitive-mno`
: the assembler considers mnemonic strings as being case sensitive.

These options changes the default behavior of conventional binutils assembler. There is a third command line option called `--archc` which displays the ArchC version used to generate the tool and the architecture name.

Chapter 4

Dynamic Linking Support

The ArchC generated binary tools include a *dynamic linking* enabled linker, that is, capable of producing shared libraries. Because this is still considered an advanced feature for embedded platforms, the processor model do not need to specify extra information required for dynamic linking. If these information are not specified, ArchC will not complain and will still generate a linker, although this linker will fail to produce shared libraries if the necessary information is absent.

Note that dynamic linking support is often not needed. If you are building an ArchC processor model, you may safely skip this information. Shared libraries are necessary, however, if you use the linker to compile the dynamic version of glibc targeting your processor.

4.1 Specifying information for dynamic linking

In order to produce a correct dynamic linking capable linker, you need to add two files to your processor model. The first one, `dynamic_info.ac`, contains regular C array declarations and definitions used by the linker when performing dynamic linking related tasks.

```
 /* The name of the dynamic interpreter.  This is put in the .interp
    section.  */
#define ELF_DYNAMIC_INTERPRETER     "/usr/lib/ld.so.1"

#define PLT_HEADER_SIZE 16

/* The size in bytes of an entry in the procedure linkage table.  */
#define PLT_ENTRY_SIZE 16

/* The first entry in a procedure linkage table looks like
   this.  It is set up so that any shared library function that is
   called before the relocation has been set up calls the dynamic
   linker first.  */
ac_plt0_entry (PLT_HEADER_SIZE / 4) =
  {
    0xe52de004,/* str   lr, [sp, #-4]! */
    0xe59fe010,/* ldr   lr, [pc, #16]  */
```

```
    0xe08fe00e,/* add   lr, pc, lr     */
    0xe5bef008,/* ldr   pc, [lr, #8]!  */
  };

/* Subsequent entries in a procedure linkage table look like
   this.  */
ac_plt_entry (PLT_ENTRY_SIZE / 4) =
  {
    0xe28fc600,/* add   ip, pc, #NN */
    0xe28cca00,/* add  ip, ip, #NN */
    0xe5bcf000,/* ldr  pc, [ip, #NN]! */
    0x00000000,/* unused */
  };

#define ac_model_can_patch_plt
```

The example above was extracted from the ARM model. In this file, you need to specify the dynamic loader (typically a path to the system `ld.so`), although the ArchC simulator ELF loader itself ignores this information, as described later. This string is used only if the produced software is intended to be deployed on a real platform, running the Linux operating system.

Also, you need to specify the contents of the PLT table, which are completely target dependent. Note that we specify the size in bytes, and the contents as an array of words, which codifies instructions. You may not represent the table directly in assembly syntax (as annotated in the comment area) because at this stage, the assembler is not yet available. You may use a pre-existent assembler to codify the necessary instructions and then use `objdump`, available in the `binutils` package, to dump the object file contents and discover the hexadecimal code for the instructions.

It is important that you study how the PLT is assembled in your platform. This information is described in the processor ABI.

The second file is the `dynamic_patch.ac`, and contains C function definitions, including C code, that instructs the linker specifically how the PLT should be patched in your architecture. This information also is target dependent, and each ABI defines a specific way of patching a PLT entry in order to encode the position of the related GOT entry. Below is the example of the ARM ABI.

```
/* ARM Model - Dynamic linking helper functions */

/* Input: got_displacement, plt_address */
ac_patch_plt0_entry()
{

  /* Calculate the displacement between the PLT slot and
     &GOT[0]. */
  got_displacement -= 16;

  /* The displacement value goes in the otherwise-unused last word
     of the second entry.  */
```

```
  /* 32 bits, location to write, value to write */
  ac_patch_bits (32, plt_address + 28, got_displacement);
}

/* Input: got_displacement, plt_offset, plt_address */
ac_patch_plt_entry()
{
  unsigned int entry_copy[PLT_ENTRY_SIZE / 4];

  /* got_displacement has the displacement between this PLT entry
     and its related GOT entry */
  got_displacement -= 8; /* We'll use it PC relative, and ARM
                            pc-relative is PC + 8 */

  /* Get a copy of the plt entry instructions already stored there */
  entry_copy[0] = (unsigned int) ac_get_bits(32, plt_address
                                             + plt_offset + 0);
  entry_copy[1] = (unsigned int) ac_get_bits(32, plt_address
                                             + plt_offset + 4);
  entry_copy[2] = (unsigned int) ac_get_bits(32, plt_address
                                             + plt_offset + 8);

  /* Relocate these plt entry instructions using ARM rotation
     mechanism to store large immediates */

  entry_copy[0] |=  ((got_displacement & 0x0ff00000) >> 20);
  entry_copy[1] |=  ((got_displacement & 0x000ff000) >> 12);
  entry_copy[2] |=  (got_displacement & 0x00000fff);

  /* Write back the result */

  ac_patch_bits(32, plt_address + plt_offset + 0, entry_copy[0]);
  ac_patch_bits(32, plt_address + plt_offset + 4, entry_copy[1]);
  ac_patch_bits(32, plt_address + plt_offset + 8, entry_copy[2]);
}
```

Please note that you must calculate the target GOT address using the input parameters `got_displacement`, `plt_offset` and `plt_address`. In order to retrieve the codified instructions from the PLT table, use the C function `ac_get_bits`() as in the example. When writing back the patched instruction with the calculated address, use the C function `ac_patch_bits`().

4.2 The dynamic linker and loader

When shared libraries and code linked against them are used, a regular simulator is not sufficient to load this special code and begin platform simulation. To address this problem, the ArchC interpreted

```
# This file contains a one-to-one mapping between ARM ABI relocation
# codes and ArchC's generated linker relocation codes. Its intended
# use is to perform a conversion using the acrelconvert tool.

# R_ARM_RELATIVE
23 = 1
# R_ARM_COPY
20 = 2
# R_ARM_JUMPSLOT
22 = 3
# R_ARM_GLOBDAT
21 = 4
# R_ARM_ABS32
2 = 7
# R_ARM_REL32
3 = 10

# End of file
```

Figure 4.1: ac_rtld.relmap file used in the ARM model.

simulator loader was enhanced to load ELF executables, find its shared libraries dependencies and resolve all symbol references (i.e., when the executable calls a library function and the address must be resolved depending on the shared library load position). Because of all these activities involved in the loading of a dynamically linked executable, the component responsible for them is called *dynamic linker and loader*, since it is not just a regular loader.

If the shared libraries were produced by a third party linker, the relocation codes used may not be compliant with the relocation codes used by the ArchC linker. In order to support loading these libraries, you can write a model addend called `ac_rtld.relmap` (the relocation map for ArchC runtime linker and loader). This is a simple text file with direct relations converting target ABI relocation codes to ArchC's version. Figure 4.1 shows the ARM example. On the left there are ARM ABI relocation codes, while on the right there are ArchC linker relocation codes, and the equal sign express the equivalence between them.

4.3 Relocation code conversion tool

When necessary, you may also use `acrelconvert`, a simple tool that reads a map (in the form presented in Figure 4.1, but no restricted to a specific file name) and converts relocation codes from object files.