The ArchC Assembler Generator

v1.5

Reference Manual

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

July 2005

Copyright © 2005 The ArchC Team
Av. Albert Einstein, 1251 13084-971
PO Box 6176 - Campinas/SP - Brazil

Contents

1 Introduction **7**
1.1 Goals . . . 7
1.2 Current limitations . . . 7
1.3 Feedback . . . 8
1.4 Authors . . . 8

2 The ArchC language new constructs **9**
2.1 Assembly language symbols . . . 9
2.2 Assembly language syntax and operand encoding . . . 10
2.2.1 Operand types . . . 12
2.2.2 Modifiers . . . 12
2.2.3 Syntax overloading . . . 15
2.3 Defining synthetic instructions . . . 16

3 Generating assemblers **19**
3.1 Before starting... . . . 19
3.2 The building process . . . 19
3.2.1 Source files generation . . . 20
3.2.2 Compiling the assembler . . . 21
3.3 A detailed example . . . 22
3.4 Summary of the building process . . . 24

4 Using the assembler **25**
4.1 Assembler behavior . . . 25
4.2 Command line options . . . 26

A New construct syntaxes **29**

List of Figures

2.1 Defining the MIPS-I register names 10
2.2 Describing the MIPS-I assembly language syntax 12
2.3 Wrong encoding of the SPARC instruction *ba* 13
2.4 Right encoding of the SPARC instruction *ba* 15
2.5 Syntax overloading in the SPARC 16
2.6 Simple pseudo instruction definition in the SPARC model 16
2.7 Defining some MIPS-I synthetic instructions 17

3.1 `asmgen.sh` command line options 21

4.1 `acasm` assemblers command line options 26
4.2 Relocation addends in the MIPS and SPARC 27

Chapter 1

Introduction

This manual introduces the new ArchC constructs used to describe the assembly language syntax and operand encoding. It also introduces the `acasm` tool, which can automatically generate assemblers from ArchC models using the new constructs. We describe the use of the tool as well as the operation and behavior of the generated assemblers.

1.1 Goals

The main goals we have set before starting the `acasm` project were:

1. To extend the ArchC language to support assembly language syntax and operand encoding information;
2. To create a tool which could automatically generate assemblers based on the new information collected from an ArchC model;
3. To allow the use of the object files created by the generated assemblers in the ArchC simulators without any change.

We have accomplished these goals. Two models, one for the MIPS and another for the SPARC architectures were expanded to include the new constructs. We are working on removing some of the current limitations (discussed next) for the next version.

1.2 Current limitations

The development of `acasm` and the ArchC language is in constant progress. For the first version of `acasm` we have put some limitations in order to reach our goals. We

will be removing some (or all) of the limitations over time, as we expand the language and more ArchC models are available.

In its current version, the tool has the following limitations:

1. All instructions must have the same size;
2. Size of the instructions must be 8, 16, 24 or 32-bit. Note that we have not tested any model which uses less than 32-bit;
3. Conditional pseudo instructions cannot be described;
4. Assembler output file format: relocatable ELF (see section 4.1 for more details). Note that this format can be used as input in the ArchC simulators.

We have organized this manual in the following way: Chapter 2 introduces and gives details about the new language constructs; Chapter 3 describes how to use the `acasm` tool to build assemblers, while Chapter 4 shows how to operate and the behavior of the generated assemblers.

1.3 Feedback

Please, let us know if you have any problems using this manual. If you find any errors or have any suggestions, you can reach us at [2].

1.4 Authors

This manual was written and is maintained by Alexandro Baldassin with help from Sandro Rigo and Marcus Bartholomeu. The `acasm` tool is part of the ArchC distribution [2].

Chapter 2

The ArchC language new constructs

In this section we present the new constructs added to the ArchC language, their syntax and semantic. There are three new constructs which allow one to describe the assembly language syntax and operand encoding easily. All of them must be written in the ISA part of an ArchC model. Besides, if you want only to generate an assembler, the instructions behavior do not need to be written. Note that all of the new constructs are optional, that is, if they aren't used the model will not break.

The formal syntax of the new constructs can be found in appendix A.

2.1 Assembly language symbols

Symbols specific to an architecture (like register names) can be defined in ArchC by using the keyword `ac_asm_map`. This construct groups a set of symbol-value pairs under an identifier name specified when constructing the mapping.

Figure 2.1 shows an example of `ac_asm_map` to define the set of register names and values for the MIPS-I architecture. These same register names are used by the MIPS GNU assembler.

The example of Figure 2.1 creates a group of symbols definition under the identifier **reg**. Lines 2 to 9 define each symbol and its associated value. The literal symbol string must be specified between double quotes. The syntax is flexible enough to allow range of values to be used. For example, line 1 creates the symbols `$0`, `$1`, `$2`, ..., `$31`, with values `0`, `1`, `2`, ..., `31`, respectively.

Note that it is also possible to assign two or more different symbols to the same value. For example, both lines 2 and 3 of figure 2.1 map to the same value, `0`. The opposite, however, is not valid.

```
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: Defining the MIPS-I register names

The identifier can be seen as an operand type, in the sense that it forces the assembler to recognize only a specific set of symbols. Later on, it can be used in the declaration of the assembly language syntax as seen in section 2.2.

Important!

Not all characters can be used in a symbol name. At the current version, the following characters are forbidden:

`; ' /* */`

This restriction is necessary since those characters have a special meaning to the assembler. Other characters such as #, `!` and `@` migh cause unexpected behaviors, since the generated assemblers use them internally as comment characters. Comment characters will be configurable in next realeases.

2.2 Assembly language syntax and operand encoding

Both the assembly language syntax and operand encoding are described using a single construct in ArchC, named `set_asm`. In fact, this construct is a property of every

ArchC instruction created by the `ac_instr` command. Its syntax ressembles that of the `printf/scanf` family in the C language:

```
set_asm(char *sintax_string, ...)
```

The syntax string specifies an instruction syntax. It is made up of literal characters and conversion specifiers. Conversion specifiers always start with the `%` character, like in the `scanf` function, but in ArchC they play a different role, limiting the range of valid symbols recognized by the assembler. For example, one should expect an operand of an immediate instruction to be a sequence of decimal digits rather than a label name. We refer to conversion specifiers as operand types from now on (although they do not specify a 'real' type, like those from high level languages such as Pascal and C).

The first white space found in the syntax string splits it into two parts: the mnemonic and the operand string. The operand string might be empty (optional). Each operand specifier must have an instruction field name specified in the argument list of the construct. As a first example, consider the following syntax definition:

```
insn.set_asm("mno %op1, %op2", field1, field2);
```

It defines an assembly syntax for the instruction `insn`. The mnemonic is the string `mno` and it has two operands of types `op1` and `op2`, respectively. The value assigned to the `op1` operand will be encoded into the `field1` while the `op2` value will be encoded into the `field2`. Both `field1` and `field2` identifiers must be fields of the format associated with the `insn` instruction.

When describing the operand string, white spaces between tokens can be supressed. A token is a sequence of the characters [a-zA-Z 0-9_$.]. Other characters are considered single tokens. There are some restrictions on the characters used when describing the syntax string:

1. **Mnemonic** – can only begin with the characters [a-zA-Z_]. The characters `;`, `'`, `/*` and `*/` are not valid when describing the mnemonic string. Furthermore, one should not use characters which have a special meaning for the assembler, like comment ones.

2. **Operand string** – the ArchC parser does not put restriction on the characters used by the operand string but, as we have said, some characters which have special meaning for the assembler should not be used. It is possible to use the literal character `%` through the scape sequence `\%`.

2.2.1 Operand types

Operand types specific to an architecture can be created using the `ac_asm_map` construct. The identifier grouping the symbols can then be used while describing the instruction syntaxes. For example, the `%reg` operand type (created in section 2.1) could be used to specify register operands in the MIPS-I instructions.

Types which are not specific to a given architecture are provided as builtin by ArchC. There are three builtin types:

1. **exp** – any arithmetic expression. It may include operation between symbols and constants, for example: `label-5`;

2. **imm** – only immediate integer values are recognized. Note that `10-5`, for example, is not a valid `imm` type;

3. **addr** – only symbolic references are recognized.

The basic builtin type is **exp**. The other ones (**imm** and **addr**) are specializations of the basic type.

Figure 2.2 shows `set_asm` declaration examples extracted from the MIPS-I model.

```
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 2.2: Describing the MIPS-I assembly language syntax

The instruction `lw` in line 1 of figure 2.2 uses 3 operands. The first one is of type **reg** and its value will be encoded in the instruction field **rt**. The second one was given an **imm** type and the instruction field **imm**. The last one, of type **reg**, will have its value encoded in the **rs** field.

2.2.2 Modifiers

A value assigned to an instruction operand is always stored in a variable of size equal to the architecture word. When the value is going to be encoded in the instruction field, the encoding routine truncates the value to the field size. That is, if an architecture word is 32-bit and a field has 20-bit, the high 12-bit of the operand value is discarded in the encoding routine. This *default* behavior does not always lead to a correct encoding process.

Take for example, the SPARC instruction `ba` defined in section (a) of figure 2.3. Section (b) shows a hyphotetic code fragment in assembly language where the `ba` instruction is used. In section (c) we show how the value assigned to the instruction operand `addr` is encoded. Note that the value is initially stored in a 32-bit variable. The encoding routine then only uses the lower 22-bit (the field size). Furthermore, the `ba` instruction is PC-relative and the value encoded is word-aligned, that is, it sees the memory as blocks of 4-bytes. The correct value to be encoded in this example should be -1, but as showed in figure 2.3, it was `0x010108`.

(a) syntax and operand encoding:

set_asm("ba %addr", addr) // addr is a 22-bit field

(b) assembly language fragment:

Virtual address	label	instruction
0x810108	label1:	nop
0x81010C		ba label1
0x810110		nop

(c) operand encoding:

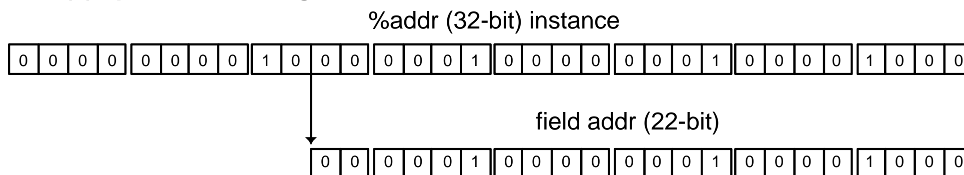


Figure 2.3: Wrong encoding of the SPARC instruction *ba*

In order to get the correct value encoded, additional operations must be performed. In the example of figure 2.3, it is necessary to subtract the PC value from the operand value and shift it to the right by 2. ArchC allows such additional operations to be performed through the concept of *modifiers*.

A modifier is specified using a single letter together with the operand type in a `set_asm` declaration. In the current version, the provided builtin modifiers are:

`L[ n ] [ s|u ]` – lower bits
: Selects the n lower bits. By default, it does not perform sign extension (u), which is accomplished by specifying s. If n is not specified, it defaults to the field size. Note that the **L** modifier is the default modifier (`%imm == %immL`).

`H[ n ] [ c ] [ s|u ]` – higher bits
: Selects the n higher bits. By default, it does not perform sign extension (u), which is accomplished by specifying s (the highest bit is used to extend the sign). If n is not specified, it defaults to the field size. The option c adds the carry bit from the lower part.

`R[ n ] [ b ]` – PC relative
: Subtracts the PC value from the operand value. If an argument n is specified, it adds that to the computation (i.e., MIPS-I uses PC+4). Negative values must be specified using the b option (i.e., `%addrR4b` implies PC-4).

`A[ n ] [ u|s ]` – n-byte aligned addressing
: The value is aligned in blocks of n bytes, that is, its value needs to be shifted right by $log n$. By default, sign extension is performed. The option u allows one to change this behavior.

As we have said, a modifier is specified together with operand types. To change the behavior of the encoding of an operand of type `addr` to PC-relative, append the character R to it: `addrR`. If, like in the MIPS-I branch instructions, an addend needs to be added to the PC, specify it following the modifier character: `addrR4` (PC+4).

Figure 2.4 shows the correct specification for the SPARC `ba` instruction. It also shows how the encoding process works using the modifiers. Compare it with the example of figure 2.3 to see the differences.

A note about the `[c]` option of the `H` modifier is worth mentioning. This option adds a carry from the lower part highest bit. For example, the MIPS gas assembler uses this operation with the instruction pair `lui %hi` and `lw %lo`. By doing so, it allows the instruction pair to be used as an offset in memory accesses:

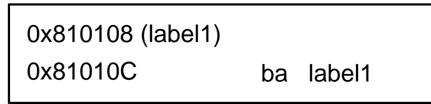
```
lui $10, %hi(0x03408A99)
lw  $11, %lo(0x03408A99)($10)
```

Note that since the MIPS processor extends the sign of the immediate operand when adding it to the register contents in the `lw` instruction, it will not result in a correct address if the `lui` instruction have not added the carry bit:

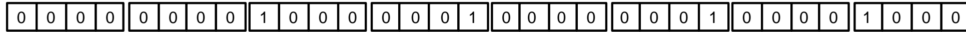
```
lui $10, %hi(0x03408A99)       = $10 <= 0x0340
lw  $11, %lo(0x03408A99)($10)

offset address = 0x03400000 + 0xFFFF8A99 = 0x033F8A99 (wrong)
```

Adding the carry bit from the lower part in the `lui` instruction gives the expected result:

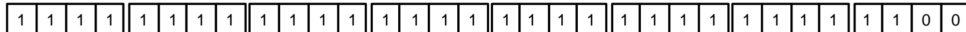
ba.set_asm("ba %expRA", disp22) // disp22 = 22 bits



%addr (32-bit) instance



Step 1. PC subtraction: instance - PC



Step 2. word aligned (>> 2, sign extension)

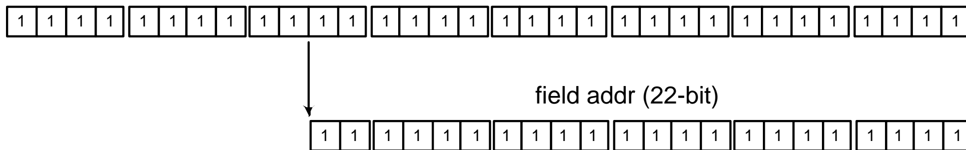


Figure 2.4: Right encoding of the SPARC instruction *ba*

```
lui $10, %hi(0x03408A99)       = $10 <= 0x0341
lw  $11, %lo(0x03408A99)($10)

offset address = 0x03410000 + 0xFFFF8A99 = 0x03408A99 (ok)
```

2.2.3 Syntax overloading

The `set_asm` construct also allows one to assign multiples syntaxes to the same ArchC instruction. It is usefull for an instruction may have different syntaxes for its operands, like happens very often in the SPARC architecture.

Figure 2.5 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 supressed (the register one).

Simple pseudo instructions can also be defined using the `set_asm` construct. Figure 2.6 gives an example of this use for some instructions of the SPARC-V8 architecture. Line 1 of figure 2.6 shows the syntax of the SPARC `or` instruction, while lines 2 and 3 defines the pseudo instructions `clr` and `mov` based on it. Lines 7, 11 and 12 show another examples of simple pseudo instruction declarations. They are declared

```
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 2.5: Syntax overloading in the SPARC

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 or_reg.set_decoder(op=0x02, op3=0x02, is=0x00);
5
6 subcci.set_asm("subcc %reg, %imm, %reg", rs1, simm13, rd);
7 subcci.set_asm("cmp %reg, %imm", rs1, simm13, rd="%g0");
8 subcci.set_decoder(op=0x02, op3=0x14, is=0x01);
9
10 jmpl_reg.set_asm("jmpl %reg + %reg, %reg", rs1, rs2, rd);
11 jmpl_reg.set_asm("jmp %reg + %reg", rs1, rs2, rd="%g0");
12 jmpl_reg.set_asm("call %reg + %reg", rs1, rs2, rd="%o7");
13 jmpl_reg.set_decoder(op=0x02, op3=0x38, is=0x00);
```

Figure 2.6: Simple pseudo instruction definition in the SPARC model

2.3 Defining synthetic instructions

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

One starts a `pseudo_instr` definition by describing 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 is specified. Parameters from the pseudo instruction syntax can be used by the native ones through the `%` character and a number indicating which parameter from the pseudo must be replaced (similar to the `macro` construct used by the GNU assemblers).

Figure 2.7 shows two definitions of synthetic instructions used in the MIPS-I 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` for 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 2.7: Defining some MIPS-I synthetic instructions

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

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

Chapter 3

Generating assemblers

This chapter describes in detail how `acasm` can be used to generate a working assembler for a processor architecture modeled in ArchC. We first give an overview of the building process using the `asmgen.sh` script, and then we show a detailed example of an assembler generation for a MIPS-I ISA using the MIPS-I ArchC model. If you want a quick view of the building process, go to the section 3.4 where we present a summary.

3.1 Before starting...

Before using `acasm` you need to have the GNU Binutils source tarball (versions 2.15 and 2.16 are supported) [3] descompacted in a directory. Note that you will also need some extra tools required by the ArchC distribution, mainly the Bison and Flex ones. Check the ArchC language manual for details [2].

3.2 The building process

The process of building the executable assembler can be split in two steps: the assembler source files generation through `asmgen.sh`; and the assembler compilation through the GNU Autotools framework.

The source files generation is performed by the `asmgen.sh` script. Its purpose is not only to generate the files but also to automate some tasks like creating a directory tree and patching some Binutils files. The compilation step deals with running the GNU Autotools scripts (`configure`, `make` and `make install`) and must be done by the user.

3.2.1 Source files generation

Once you have built the ArchC distribution [1], the `bin` directory will hold the `acasm` executable file. After running `acasm`, you'll still need to perform other tasks (i.e., merging some files and some other post-processing stuff) in order to produce the final assembler source files. There will still remain the task of copying the files to the Binutils source tree and the patching of some of its files. Together, all these tasks require quite a lot of work and may be very time consuming.

All those tasks can be automated by using the `asmgen.sh` `bash` script. This script is also located in the ArchC `bin` directory and executes the following actions:

1. Creates a local directory holding a Binutils sub-tree.
2. Executes the `acasm` tool to generate the source files which are stored in the Binutils sub-tree previously created.
3. Creates the file `tc-[arch].c` from `tc-templ.c` and `tc-funcs.c`.
4. Patches the Binutils configuration files (mainly `configure.in` and `Makefile.am`).
5. Copies the final source files to the Binutils source tree.

The `[arch]` string used in the third action refers to the architecture name given by the user when generating the assembler, as we will see soon.

If you want to override the GNU Binutils path given while installing the ArchC package (or set a new one if no one was specified), set the environment variable `BINUTILS_PATH`:

`BINUTILS_PATH`
: Must point to the root of a GNU Binutils source distribution tree.

The `asmgen.sh` command line is shown in figure 3.1. `<archc source file>` is the main ArchC model file, while `<architecture name>` is the name in which the architecture is going to be recognized under the Binutils framework. This name is the same as `[arch]` showed earlier, and is part of a *configuration name* (the *cpu* part) [2].

---

[1] Check the ArchC language manual for further information on how to build the ArchC tools. It can be found at [2]

[2] The GNU Autotools name all types of systems using a standardized format known as a configuration name. A configuration name is permitted to have four parts on systems which migh distinguish the kernel and the operation system, and it is written as: *cpu-manufacturer-kernel-operating_system*.

```
Usage: asmgen.sh [options] <archc source file> <architecture name>

Create gas target source files and copy them to the binutils tree.

Options:
  -c, --create-only   only create the files, do not copy to binutils tree
  -h, --help          print this help
  -v, --version       print version number

Report bugs and patches to ArchC Team.
```

Figure 3.1: `asmgen.sh` command line options

Important!

When specifying an <`architecture name`> you **cannot** choose one that is already used by Binutils. You can check if a cpu name is already been used by running the `config.sub` script located in the Binutils root directory.
Assume you want to use the name `[arch]` for your architecture; you could run:

```
config.sub [arch]-elf
```

in the Binutils root directory to check if `[arch]` has already been used. If it has not been used, you are free to use it in the building process. Note that the `asmgen` script will detect if an `[arch]` name already exists in the Binutils scope, and not patch the tree in that case.

Actually, the only effective command line option provided by `asmgen.sh` is `'-c'`. When it is specified, the script does not perform steps 4 and 5 as showed earlier, that is, it only generates the source files but does not touch the Binutils tree.

3.2.2 Compiling the assembler

After executing the `asmgen.sh` script, your Binutils source tree will have all the files necessary to build a new assembler for a given ArchC processor model. From this point on, you should proceed pretty much the same way you do when building any other assembler with the GNU Autotools framework (running the `configure`, `make` and `make install` sequence).

The detailed steps are:

1. Execute the script `configure` found in the Binutils root directory (it is usually done outside of that directory). You will need to provide the Binutils architecture name, as you did when executing `asmgen.sh`, using: `--target=[arch]-elf`, where `[arch]` is the architecture name you used when running `asmgen.sh`. You will probably use `--prefix` to specify a directory where the assembler must be installed.

2. Execute the command `make all-gas`. Note that you should **not** use only the command `make`, because that will try to build all the GNU Binutils tools. Since not all the tools from the package have been retargetted, this command will surely provoke an error.

3. Execute the command `make install-gas` to copy the binary files generated to an install directory (the one you specified with the `--prefix` option in `configure`).

3.3 A detailed example

We describe in this section all the steps necessary to build an assembler for a MIPS-I ArchC model using our tool. We assume the user will be using a Bourne Again Shell (bash) compatible shell, with the following directories:

`/archc`
: ArchC root directory.

`/models/mips1`
: MIPS-I model files directory.

`/install`
: directory which will hold the executable assembler.

`/build`
: a temporary directory to be used while building the assembler.

`/binutils-2.16`
: GNU Binutils source distribution directory. Note that the process of creating a new assembler will add some files and change others during the building process. So, if you do not want the source distribution to be changed, you must make a copy of the directory by yourself.

It is also desirable to defined some environment variables. For this example, we define the following ones:

```
export BINUTILS_PATH=/binutils-2.16
export PATH=$ARCHC_PATH/bin:$PATH
export TARGET_NAME=mips1
```

The first variable points to the directory where the GNU Binutils source tarball was descompacted. If you already have specified the Binutils path when the ArchC package was installed, it is not necessary to redefine it. The second and third variables are just to make our lives easier. In the third one, we are expanding the `PATH` definition to reach the ArchC tools, so you can just type their name anywhere when running them. `TARGET_NAME` is the architecture name and must not have already been used.

The steps necessary to create the executable assembler for the MIPS-I processor are:

1. Build the ArchC package (if not done already)
   Just enter the ArchC root directory and type `make`:

   ```
   cd /archc
   make
   ```

2. Generate the assembler source files
   Go into the model directory and run the `asmgen` script:

   ```
   cd /models/mips1
   asmgen.sh mips1.ac $TARGET_NAME
   ```

3. Build the assembler using the GNU autotools
   Go into the build directory, run the `configure` script, followed by the `make` and `make install` commands:

   ```
   cd /build
   $BINUTILS_PATH/configure --prefix=/install --target=$TARGET_NAME-elf
   make all-gas
   make install-gas
   ```

You can check if the assembler was correctly built by executing it with the `--archc` option:

```
/install/bin/$TARGET_NAME-elf-as --archc
```

3.4 Summary of the building process

The steps used to create an executable assembler from an ArchC model, assuming the ArchC tools have already been compiled, are:

1. **Execute de asmgen.sh script with the main ArchC model file and an assembler name**

   ```
   cd model-files
   asmgen.sh mymodel.ac assembler_name
   ```

2. **Build the assembler using the GNU autotools**

   ```
   mkdir build-dir
   mkdir install-dir
   cd build-dir
   ~/binutils-dir/configure --target=assembler_name-elf
                            --prefix=../install-dir
   make all-gas
   make install-gas
   ```

Where `binutils-dir` is the location where the GNU Binutils package was descompacted.

Chapter 4

Using the assembler

This section provides information about the common behavior of the generated assemblers, as well as their command line options.

4.1 Assembler behavior

An assembler generated by the `acasm` tool accepts as input an assembly language source file for the target architecture. Common diretives and structures (like macros, expressions, labels) are used in the same way they are used by any *gas-like* assembler. For further information about that, check [1].

The default output object file format generated by the assembler is a relocatable ELF. Relocatable object files may have unresolved symbols entries and need a linker/loader to resolve them in order to become executable and actually run in a processor. Since we do not have a linker at the moment, the relocatable ELF object files generated by our assembler have some particularities:

- its ELF header file is of type relocatable (`ET_REL`);
- it does not contain any relocation entries;
- the *text* section is assembled starting from address 0, and all symbols are resolved. If an unresolved symbol is found, the assembler generates an error and no object file is generated;
- the *data* section is assembled starting after the last address used by the *text* section.

The object files generated are absolute ones in the sense that they have all symbols resolved and assigned to static addresses. They can be used directly as input in the

ArchC simulators (we have implemented a loader which can load relocatable ELF files assembled just like our assemblers do). This eventually will change when we have finished our linker generator tool, but it will be transparent for the end-user.

Sometimes one might want to change the default behavior of the assembler. For example, we did not want to resolve extern references to compare the files generated by our assemblers with the assemblers from the GNU Binutils package. Next section gives details about command line options which can change the default behavior of the assemblers.

4.2 Command line options

There are some command line options common to all *gas-like* assemblers. They can be displayed with the `--help` options. You can check them all at [1]. At the end of the options list there will be some under the `md options` (machine dependent options). The assemblers generated by `acasm` have the md options showed in figure 4.1.

```
--ignore-undef           ignore undefined symbols
--output-mode=<mode>     set the format of the output file
                         <mode>: absolute, rel, rela
--text-addr=<addr>       set the start address of .text section to <addr>
--data-addr=<addr>       set the start address of .data section to <addr>
--archc                  display ArchC information
```

Figure 4.1: `acasm` assemblers command line options

We present next a detailed description of each one:

`--ignore-undef`
If the assembler finds an undefined (unresolved) symbol while generating the object file, it will not generate an error with this option. Instead, all unresolved symbols will be given the default value 0.

`--output-mode=<mode>`
This option specifies the output mode for the object files. The default mode is, as already said in section 4.1, absolute. Two other options are available, and they deal with aspects of relocatable ELF files:

- `rel` – the relocation addend is stored implicitly in the location to be modified;
- `rela` – the relocation addend is stored explicitly in the `addend` field of the relocation entry.

The difference between a rel and a rela mode is illustrated by figure 4.2: in (a) there is a MIPS add instruction at the top and its respective encoding at the bottom, generated by the MIPS gas assembler (we used the readelf tool to visualize the sections); in (b) we have the same fragment for the SPARC. We have used a label as an operand in the instructions in both architectures, located at address 4. From this figure it is possible to see that MIPS saves the addend in the instruction location, while the SPARC saves it in the relocation entry. Note that, in the SPARC case, one more field in the relocation entry is necessary.

Note also that modes rel and rela options are just to make the text and data sections of the files generated by the assemblers compatible with default gas assemblers. In none of these cases a relocation entry is saved in the output file.

--text-addr=<addr>
The assembler will start assembling the text section at address <addr>.

--data-addr=<addr>
The assembler will start assembling the data section at address <addr>. You can force it to assemble the data section after the text one by using .text as argument.

--archc
Displays information about the generated assembler.

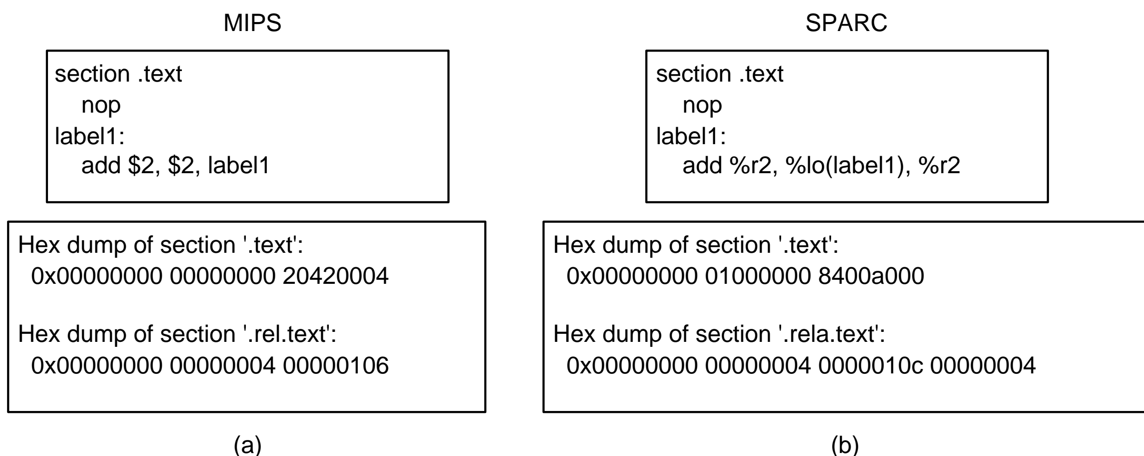


Figure 4.2: Relocation addends in the MIPS and SPARC

Appendix A

New construct syntaxes

We use the extended BNF and regular expressions while describing the syntaxes.

```
id        := [a-zA-Z][a-zA-Z0-9_]+
digit     := [0-9]+
string    := "[^"]*"

op_range := '[' digit '..' digit ']'


symb_def := 'ac_asm_map' id '{' (list_map)* ';' '}'
list_map := string (',' string)* '=' digit          |
            string op_range '=' op_range            |
            op_range string '=' op_range            |
            string op_range string '=' op_range


synt_def := 'set_asm' '(' string (args)* ')' ';'
args     := ','(id | id '=' digit | id '=' string)


pseu_def := 'pseudo_instr' '(' string ')' '{' (string ';')+ '}'
```

Bibliography

[1] D. Elsner et al. *Using as, the GNU assembler.* Free Software Foundation, Inc., March 1993. version 2.15.

[2] http://www.archc.org. The ArchC Website.

[3] http://www.gnu.org/software/binutils. The GNU Binutils Website.