armips assembler v0.11

- Author: Kingcom
- Source: https://github.com/Kingcom/armips
- Automated builds: http://buildbot.orphis.net/armips

1. Introduction

Note: This file is still incomplete, some information is missing or may be outdated.

1.1 Usage

The assembler is called from the command line. The usage is as follows:

```
armips code.asm [optional parameters]
```

`code.asm` is the main file of your assembly code, which can open and include other files. The following optional command line parameters are supported:

-temp <filename>

Specifies the output name for temporary assembly data. Example output:

```
; 1 file  included
; test.asm

00000000 .open "SLPM_870.50",0x8000F800  ; test.asm line 1
8000F800 .org 0x800362DC                 ; test.asm line 5
800362DC   jal     0x801EBA3C            ; test.asm line 7
800362E0 .Close                          ; test.asm line 9
```

-sym <filename>

Specifies the output name for symbol data in the sym format. This format is supported by the debuggers in NO$PSX and NO$GBA. Example output:

```
00000000 0
80000000 .dbl:0010
80000010 main
8000002C subroutine
80240000 newblock
```

-sym2 <filename>

Specifies the output name for symbol data in the sym2 format. This format is supported by the debuggers in PCSX2 and PPSSPP. Example output:

```
00000000 0
80000000 .dbl:0010
80000010 Main
8000002C Subroutine,0000001C
80240000 NewBlock,00000014
```

-erroronwarning

Specifies that any warnings shall be treated like errors, preventing assembling. This has the same effect as the `.erroronwarning` directive.

`-equ <name> <replacement>`

Equivalent to using `name equ replacement` in the assembly code.

`-strequ <name> <replacement>`

Equivalent to using `name equ "replacement"` in the assembly code.

`-definelabel <name> <replacement>`

Equivalent to using `.definelabel name, replacement` in the assembly code.

`-root <directory>`

Specifies the working directory to be used during execution.

`-stat`

Outputs statistics for bytes used within areas after completion. Example output:

```
Total areas and regions: 5342 / 7934
Total regions: 916 / 1624
Largest area or region: 0x0806E80C, 564 / 1156
Most free area or region: 0x0806E80C, 564 / 1156 (free at 0x0806EA40)
Most free region: 0x0806E80C, 564 / 1156 (free at 0x0806EA40)
```

2. Installation

2.1 Download binary

Download the latest Windows 32-bit binary from the Automated armips builds site. You will need the Microsoft Visual Studio 2015 x86 Redistributable.

2.2 Building from source

The latest code is available at the armips GitHub repository. Make sure to also initialize and update submodules. This can be accomplished with one command:

```
$ git clone --recursive https://github.com/Kingcom/armips.git
```

You will need CMake and a C++17 compliant compiler (recent versions of Visual Studio, GCC and Clang have been tested). All CMake generators should be supported, but Ninja is recommended and the most well tested. Create a build directory, invoke CMake from there, and then simply run the chosen build tool. E.g. on Unix platforms:

```
$ mkdir build && cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ cmake --build .
```

Or on Windows using Visual Studio:

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build . --config Release
```

Please refer to the CMake documentation for further information.

3. Overview

The assembler includes full support for the MIPS R3000, MIPS R4000, Allegrex and RSP instruction sets, partial support for the EmotionEngine instruction set, as well as complete support for the ARM7 and ARM9 instruction sets, both THUMB and ARM mode. Among the other features of the assembler are:

- a full fledged C-like expression parser. It should behave exactly like in any C/C++ code, including all the weirdness. All immediate values can be specified by an expression, though some directives can't use variable addresses including labels
- you can open several files in a row, but only one output file can be open at any time. You can specify its address in memory to allow overlay support. Any file can cross-reference any other included file
- local, static, and global labels (see 4.3 Labels)
- table support for user defined text encodings (see 4.7 String encoding)
- several MIPS macros to make writing code easier and faster (see 5.1 General directives)
- user defined macros (see 6.3 User defined macros)
- built-in checks for possible load delay problems (see 4.6 Load delay detection)

- optional automatic fix for said problems by inserting a nop between the instructions
- output of the assembled code to a text file, with memory addresses and origin (see 1.1 Usage)
- a directive to ensure that data is not bigger than a user defined size (see 4.8 Areas)

4. Features

4.1 Files

Unlike other assemblers, you don't specify the input/output file as a command line argument. You have to open the file in the source code, and also close it yourself. This was done in order to support overlays, which are very common in PSX and NDS games. Instead of only having one output file, you can have as many as you need - each with its own address in memory. The files can cross-reference each other without any problems, so you can call code from other files that are currently not opened as well.

```
.Open "SLPS_035.71", 0x8000F800
; ...
.Close
.Open "System\0007.dat", 0x800CC000
; ...
.Close
```

4.2 Syntax

Comments

Both `;` and `//` style single-line comments are supported. `/* */` style block comments are also accepted.

Statement separator

Statements are separated by newlines or `::` can be used between statements on the same line. For example, to insert four `nop` instructions, this could be written on one line:

```
nop :: nop :: nop :: nop
```

Statement line spanning

Single statements can continue on to the next line by inserting a `\` at the end of a line. Comments and whitespace can follow. For example:

```
.ascii "NSM", (VERSION == "us") ? "E" : \
              (VERSION == "jp") ? "J" : \
              (VERSION == "eu") ? "P" : \
                                  "X"
```

4.3 Labels

A label is defined by writing its name followed by a colon. It creates a symbol with that name as its identifier, and with the current memory address as its value. There is support for both local, global and static labels. Local labels are only valid in the scope between the previous and the next global or static label. Specific directives, like `.org`, will also terminate the scope. All labels can be used before the point where they are defined.

```
GlobalLabel:       ; This is a global label.
@StaticLabel:      ; This is a static label.
@@LocalLabel:      ; This is a local label, it is only valid
                   ; until the next global or static one.
OtherGlobalLabel:  ; this will terminate the scope where
                   ; @@LocalLabel can be used.
  b   @@LocalLabel ; as a result, this will cause an error.
```

Static labels behave like global labels, but are only valid in the very file they were defined. Any included files or files that include it cannot reference it. They can, however, contain another static label with the same name. When a static label is defined insde a (nested) macro, it is treated as being defined in the file where the top-level macro call occurred, rather than the file holding the macro definition where the static label is created.

A label name can contain all characters from A-Z, numbers, and underscores. However, it cannot start with a digit. All label names are case insensitive.

Additionally, `.` is a special label and can be used to reference the current memory address; it is equivalent to calling the expression function `org()`.

A label can also be defined using the `.func` / `.function` directive. The example below will create a label `MyLabel` pointing to the current memory address. In addition, if the `-sym2` command line flag is used to output a sym2 file, the size of the function block (from `.func` to `.endfunc`) will also be written to the symfile.

```
.func MyLabel
  ; assembly code
.endfunc
```

4.4 equ

The `equ` directive works as a direct text replacement on the assembly source level and is defined as follows. Unlike labels, an `equ` must be defined before it can be used.

```
GlobalEqu   equ 1
@StaticEqu  equ 2
@@LocalEqu  equ 3
```

There has to be at least one whitespace character before and after `equ`. The assembler will replace any occurrence of `GlobalEqu`, `@StaticEqu` and `@@LocalEqu` with `1`, `2` and `3` respectively. Similarly to labels, a global `equ` is valid anywhere, a static `equ` is only valid in the file it was defined, and a local `equ` is only valid in the current section, which is terminated by any global or static label or specific directives. The replacement value can be any sequence of valid tokens. Any usage of the equ name identifier is replaced by the replacement tokens in-place, before any parsing is done. The replacement can therefore also contain partial commands or expressions. For example, this code:

```
@@StringPointer equ 0x20(r29)

  lw  a0,@@StringPointer
  nop
  sw  a1,@@StringPointer
```

will assemble to this:

```
  lw  a0,0x20(r29)
  nop
  sw  a1,0x20(r29)
```

4.5 Expression parser

A standard expression parser with operator precedence and bracket support has been implemented. It is intended to behave exactly like any C/C++ parser and supports all unary, binary and ternary operators of the C language. Every numeral argument can be given as an expression, including label names. However, some directives do not support variable addresses, so labels cannot be used in expressions for them. The following bases are supported:

- `0xA` and `0Ah` for hexadecimal numbers
- `0o12` and `12o` for octal numbers
- `1010b` and `0b1010` for binary numbers

Everything else is interpreted as a decimal numbers, so a leading zero does not indicate an octal number. Be aware that every number has to actually start with a decimal digit. For example, as `FFh` is a perfectly valid label name, you have to write `0FFh` or `0xFF` in this case. Labels, on the other hand, cannot start with a digit.

A few examples:

```
mov  r0,10+0xA+0Ah+0o12+12o+1010b
ldr  r1,=ThumbFunction+1
li   v0,Structure+(3*StructureSize)
```

Value types

Three value types are supported: integers, floats and strings. Integers are defined by writing just a number in one of the supported bases. Writing a character surrounded by single quotation marks will also give its Unicode value as an integer. For example, `'a'` is equivalent to `97`.

A float is defined by an integer numerator, followed by a period, followed by the denominator, e.g. `2.5`. Floats can also use a different base prefix; in this case, both the numerator and denominator are evaluated using that base. For example, `11.5` is equivalent to `0xB.8`. Alternatively, a float can also be defined by exponential notation. This is formatted as an integer, followed by the letter `e`, followed by (optionally) a plus or minus representing the exponent sign, followed by an integer representing the exponent. For example, `314e-2` is equivalent to `3.14`; `5e3` and `5e+3` are equivalent to `5000.0`.

Strings are defined by text wrapped in double quotation marks (e.g. `"text"`). Double quotation marks can be escaped by prefixing them with a backslash (`\`). Any backslash not followed by a double quotation mark is kept as-is. If you want to use a backslash at the end of a string, prefix it by another backslash. For example, to write a double quotation mark followed by a backslash:

```
.ascii "\"\\"
```

String concatenation is possible with the `+` binary operator. Concatenating integers or floats with a string will convert those integers or floats to a string representation.

Strings can also be compared to other strings using the standard comparison operators or compared to numbers using the `==` and `!=` operators.

Built-in functions

Below is a table of functions built into the assembler that can be used with the expression parser for runtime computation.

Function	Description
`version()`	armips version encoded as int, e.g. armips v3.4.5 returns `3045` (`3*1000 + 4*10 + 5`)
`endianness()`	current endianness as string, e.g. `"big"` or `"little"`
`outputname()`	currently opened output filename, exactly as written in `.create` or `.open` directive
`org()`	current memory address (like `.`)
`org(label)`	memory address of `label`
`orga()`	current absolute file address
`orga(label)`	absolute file address of `label`
`headersize()`	current header size (displacement of memory address against absolute file address)
`headersize(label)`	header size of `label` (displacement of memory address against absolute file address)
`defined(symbol)`	`1` if `symbol` is a defined symbol, `0` otherwise

Function	Description
`fileexists(file)`	`1` if `file` exists, `0` otherwise
`filesize(file)`	size of `file` in bytes
`tostring(val)`	string representation of int or float `val`
`tohex(val, optional digits = 8)`	hex string representaion of int `val`
`round(val)`	float `val` rounded to nearest int
`int(val)`	cast float `val` to int, dropping fractional part
`float(val)`	cast int `val` to float
`frac(val)`	fractional part of float `val`
`abs(val)`	absolute value of int or float `val`
`hi(val)`	High half of 32-bit value `val`, adjusted for sign extension of low half (only available in MIPS)
`lo(val)`	Sign-extended low half of 32-bit value `val` (only available in MIPS)
`min(a, b, ...)`	minimum of int or float parameters `a`, `b`, ...; result type is int if all parameters are int, float otherwise
`max(a, b, ...)`	maximum of int or float parameters `a`, `b`, ...; result type is int if all parameters are int, float otherwise
`strlen(str)`	number of characters in `str`
`substr(str, start, count)`	substring of `str` from `start`, length `count`
`regex_match(source, regex)`	`1` if `regex` matched entire `source`, `0` otherwise
`regex_search(source, regex)`	`1` if `regex` matched subsequence of `source`, `0` otherwise
`regex_extract(source, regex, optional index = 0)`	string of `regex` matched in `source`
`find(source, substr, optional start = 0)`	lowest index of `substr` in `source` from `start`, else `-1`

Function	Description
`rfind(source, substr, optional start = -1)`	highest index of `substr` in `source` from `start`, else `-1`
`readbyte(file, optional pos = 0)`	read unsigned 8-bit value from `file` at position `pos`
`readu8(file, optional pos = 0)`	read unsigned 8-bit value from `file` at position `pos`
`readu16(file, optional pos = 0)`	read unsigned 16-bit value from `file` at position `pos`
`readu32(file, optional pos = 0)`	read unsigned 32-bit value from `file` at position `pos`
`readu64(file, optional pos = 0)`	read unsigned 64-bit value from `file` at position `pos`
`reads8(file, optional pos = 0)`	read signed 8-bit value from `file` at position `pos`
`reads16(file, optional pos = 0)`	read signed 16-bit value from `file` at position `pos`
`reads32(file, optional pos = 0)`	read signed 32-bit value from `file` at position `pos`
`reads64(file, optional pos = 0)`	read signed 64-bit value from `file` at position `pos`
`readascii(file, optional start = 0, optional len = 0)`	read ASCII string from `file` at `start` length `len` until null terminator
`isarm()`	`1` if in ARM mode, `0` otherwise (only available in ARM/THUMB)
`isthumb()`	`1` if in THUMB mode, `0` otherwise (only available in ARM/THUMB)

User defined functions

It is possible to define additional expression functions. These can contain any number of parameters, but their content is limited in scope to a single arbitrarily long expression. This expression can contain calls to other functions or user defined functions - including recursive calls to the function itself. User defined functions are defined as follows:

```
.expfunc name(parameters), content
```

`name` is the name of the function and must be unique. `parameters` has to be a comma separated list of one or more identifiers. `content` has to be a single expression and encodes the return value of the function. The result of evaluating `content` is returned to the caller.

As an example, the following will define a function to calculate a fibonacci number and then use it to print the 10th such number:

```
.expfunc fib(n), n <= 2 ? 1 : fib(n-1)+fib(n-2)
.notice "The 10th fibonacci number is " + fib(10)
```

The ternary operator can be useful to chain various conditions in a single expression.

4.6 Load delay detection

This feature is still unfinished and experimental. It works in most cases, though. On certain MIPS platforms (most notably the PlayStation 1), any load is asynchronously delayed by one cycle and the CPU won't stall if you attempt to use it before. Attempts to use it will return the old value on an actual system (emulators usually do not emulate this, which makes spotting these mistakes even more difficult). Therefore, the assembler will attempt to detect when such a case happens. The following code would result in a warning:

```
lw   a0,0x10(r29)
lbu  a1,(a0)
```

This code doesn't take the load delay into account and will therefore only work on emulators. The assembler detects it and warns the user. In order to work correctly, the code should look like this:

```
lw   a0,0x10(r29)
nop
lbu  a1,(a0)
```

The assembler can optionally automatically insert a `nop` when it detects such an issue. This can be enabled with the `.fixloaddelay` directive. However, as there is no control flow analysis, there is a chance of false positives. For example, a branch delay slot may cause a warning for the opcode that follows it, even if there is no chance that they will be executed sequentially. The following example illustrates this:

```
    bnez  a0,@@branch1
    nop
    j     @@branch2
    lw    a0,(a1)
  @@branch1:
    lbu   a2,(a0)
```

You can fix the false warning by using the `.resetdelay` directive before the last instruction.

```
    bnez  a0,@@branch1
    nop
    j     @@branch2
    lw    a0,(a1)
  .resetdelay
  @@branch1:
    lbu   a2,(a0)
```

4.7 String encoding

You can write ASCII text by simply using the `.db` / `.ascii` directive followed by the string to write. Using `.asciiz` will insert a zero byte after the string.

You can also write text with custom encodings. In order to do that, you first have to load a table using `.loadtable <tablefile>`, and then use the `.string` directive to write the text. It behaves exactly like the `.db` instruction (so you can also specify immediate values as arguments), with the exception that it uses the table to encode the text, and appends a termination sequence after the last argument. This has to be specified inside the table, otherwise 0 is used. The termination sequence can also be omitted with `.stringn`.

```
.loadtable "custom.tbl"
.string "Custom text",0xA,"and more."
```

The first and third arguments (`"Custom test"` , `"and more."`) are encoded according to the table, while the second one (`0xA`) is written as-is.

4.8 Areas

If you overwrite existing data, it is critical that you don't overwrite too much. The area directive will take care of checking if all the data is within a given space. In order to do that, you just have to specify the maximum size allowed.

```
.area 10h
  .word 1,2,3,4,5
.endarea
```

This would cause an error on assembling, because the word directive takes up 20 bytes instead of the 16 that the area is allowed to have. This, on the other hand, would assemble without problems:

```
.org 8000000h
.area 8000020h-.
  .word 1,2,3,4,5
.endarea
```

Here, the area is 32 bytes, which is sufficient for the 20 bytes used by .word. Optionally, a second parameter can be given. The remaining free size of the area will then be completely filled with bytes of that value. For example, the following code writes `01 02 03 04 05 05 05 05` :

```
.area 8,5
  .byte 1,2,3,4
.endarea
```

Regions

To help manage allocating new data in existing space, you can use `.region` and `.autoregion` for armips to automatically find an area with enough space.

`.region` uses the same parameters as `.area` , but creates a space that shared for future `.autoregion` usage. You can still use code in the region, and the remaining space is considered free (with or without fill.)

Example `.autoregion` usage:

```
.org @FreeSpace
.region 0x4000
.byte 0x12, 0x34
.endregion

.autoregion
@TheAnswer:
  .byte 42
.endautoregion
```

Auto region content will be allocated as if it was placed after the content of the region it's allocated to (potentially after other auto regions.)

A shortcut is available for regions without content at a specific location, to quickly define pools. These are equivalent:

```
.defineregion @FreeSpace,0x4000,0x00
.org @FreeSpace :: .region 0x4000,0x00 :: .endregion
```

By default, `.autoregion` will allocate to any region with sufficient space. It can be limited to a specific range of start addresses if necessary:

```
.autoregion @TextStart,@TextStart+@TextEnd
@CoverAdvisory:
  .asciiz "Don't Panic"
.endautoregion
```

For example, this might be used to ensure the code is reachable by `bl`.

If only the first parameter is given, it will simply require allocation after that virtual address.

Note that after `.endautoregion`, the output position will be reset to what it was before the `.autoregion` directive.

4.9 Symbol files

Functions.

4.10 C/C++ importer

You can link object files or static libraries in ELF format. The code and data is relocated to the current output position and all of its symbols are exported. You can in turn use armips symbols inside of your compiled code by declaring them as `extern`. Note: As armips labels are case insensitive, the exported symbols are treated the same way. Be aware of name mangling when trying to reference C++ functions, and consider declaring them as `extern "C"`.

```
.importobj "code.o"
```

You can optionally supply names for constructor and destructor functions. Functions with those names will be generated that call of the global constructors/destructors of the imported files.

```
.importlib "code.a",globalConstructor,globalDestructor
```

5. Assembler directives

These commands tell the assembler to do various things like opening the output file or opening another source file.

5.1 General directives

Set the architecture

These directives can be used to set the architecture that the following assembly code should be parsed and output for. The architecture can be changed at any time without affecting the preceding code.

Directive	System	Architecture	Comment
`.psx`	PlayStation 1	MIPS R3000	-
`.ps2`	PlayStation 2	EmotionEngine	-
`.psp`	PlayStation Portable	Allegrex	-
`.n64`	Nintendo 64	MIPS R4000	-

Directive	System	Architecture	Comment
`.rsp`	Nintendo 64	RSP	-
`.gba`	Game Boy Advance	ARM7	Defaults to THUMB mode
`.nds`	Nintendo DS	ARM9	Defaults to ARM mode
`.3ds`	Nintendo 3DS	ARM11	Defaults to ARM mode, incomplete
`.arm.big`	-	ARM	Output in big endian
`.arm.little`	-	ARM	Output in little endian

Open a generic file

```
.open     FileName,HeaderSize
.openfile FileName,HeaderSize
.open     OldFileName,NewFileName,HeaderSize
.openfile OldFileName,NewFileName,HeaderSize
```

Opens the specified file `FileName` for output. This directive terminates the scope for local labels and `equ` s. If two file names are specified, the assembler will first copy the file from `OldFileName` to `NewFileName`, then open `NewFileName`. In this case, if the copy operation fails, e.g. because the two paths point to the same file, an error is thrown.

`HeaderSize` specifies the header size, which is the difference between the first byte of the file and its position in memory. So if file position 0x800 is loaded at position 0x80010000 in memory, the header size is 0x80010000-0x800=0x8000F800. It can be changed later with the `.headersize` directive.

If relative include is off, all paths are relative to the current working directory. Otherwise the path is relative to the including assembly file.

Only the changes specified by the assembly code will be inserted, the rest of the file remains untouched.

The following copies the file `input.bin` to `output.bin`, then opens `output.bin` with a header size of `0x8000000`.

```
.open "input.bin","output.bin",0x8000000
```

Create a new file

```
.create     FileName,HeaderSize
.createfile FileName,HeaderSize
```

Creates the specified file for output. If the file already exists, it will be overwritten. This directive terminates the scope for local labels and `equ` s.

If relative include is off, all paths are relative to the current working directory. Otherwise the path is relative to the including assembly file. `HeaderSize` specifies the difference between the first byte of the file and its position in memory. So if file position 0x800 is loaded at position 0x80010000 in memory, the header size is 0x80010000-0x800=0x8000F800. It can be changed later with the `.headersize` directive.

The following creates and opens the file `output.bin` with a header size of `0x8000000` .

```
.create "output.bin",0x8000000
```

Close a file

```
.close
.closefile
```

Closes the currently opened output file. This directive terminates the scope for local labels and `equ` s.

Set the output position

```
.org  RamAddress
org   RamAddress
.orga FileAddress
orga  FileAddress
```

Sets the output pointer to the specified address. `.org` / `org` specifies a memory address, which is automatically converted to the file address for the current output file. `.orga` / `orga` directly specifies the absolute file address. This directive terminates the scope for local labels and `equ` s.

Change the header size

```
.headersize HeaderSize
```

Sets the header size to the given value which is the difference between the file position of a byte and its address in memory. This is used to calculate all addresses up until the next `.headersize` or `.open` / `.create` directive. The current memory address will be updated, but the absolute file offset will remain the same. The header size can be negative so long as the resulting memory address remains positive.

Include another assembly file

```
.include FileName[,encoding]
```

Opens the file called `FileName` to assemble its content. If relative include is off, all paths are relative to the current working directory. Otherwise the path is relative to the including assembly file. You can include other files up to a depth level of 64. This limit was added to prevent the assembler from getting stuck in an infinite loop due to two files including each other recursively. If the included file has an Unicode Byte Order Mark then the encoding will be automatically detected. If no Byte Order Mark is present it will default to UTF-8. This can be overwritten by manually specifying the file encoding as a second parameter.

The following values are supported:

- `SJIS` / `Shift-JIS`
- `UTF8` / `UTF-8`
- `UTF16` / `UTF-16`
- `UTF16-BE` / `UTF-16-BE`
- `ASCII`

Text and data directives

Align the output position

```
.align  [num[,value]]
.aligna [num[,value]]
```

Writes bytes of `value` into the output file until the memory position is a multiple of `num`. `num` has to be a power of two. If `num` isn't specified, then the alignment will be 4. If `value` isn't specified, zeros are inserted. Only the lowest 8 bits of `value` are inserted. `.align` aligns the memory address (i.e. `org()`), whereas `.aligna` aligns the file address (i.e. `orga()`).

Fill space with a value

```
.fill length[,value]
defs  length[,value]
```

Inserts `length` amount of bytes of `value`. If `value` isn't specified, zeros are inserted. Only the lowest 8 bits of `value` are inserted.

Skip bytes

```
.skip length
```

Skips `length` amount of bytes without overwriting them. This is equivalent to `.org .+length`.

Include a binary file

```
.incbin FileName[,start[,size]]
.import FileName[,start[,size]]
```

Inserts the file specified by `FileName` into the currently opened output file. If relative include is off, all paths are relative to the current working directory. Otherwise the path is relative to the including assembly file. Optionally, `start` can specify the start position in the file from it should be imported, and `size` can specify the number of bytes to read.

Write bytes

```
.byte   value[,...]
.db     value[,...]
.dcb    value[,...]
.ascii  value[,...]
.asciiz value[,...]
db      value[,...]
dcb     value[,...]
```

Inserts the specified sequence of bytes. Each parameter can be any expression that evaluates to an integer or a string. If it evaluates to an integer or float, only the lowest 8 bits are inserted. If it evaluates to a string, every character is inserted as a byte using ASCII encoding.

Write halfwords

```
.halfword value[,...]
.dh       value[,...]
.dcw      value[,...]
dh        value[,...]
dcw       value[,...]
```

Inserts the specified sequence of 16-bit halfwords. Each parameter can be any expression that evaluates to an integer or a string. If it evaluates to an integer, only the lowest 16 bits are inserted. If it evaluates to a string, every character is inserted as a halfword using ASCII encoding.

If No$gba semantics are enabled, then `dh` and `.dh` are treated as invalid directives and will return an error.

Write words

```
.word value[,...]
.dw   value[,...]
.dcd  value[,...]
dw    value[,...]
dcd   value[,...]
```

Inserts the specified sequence of 32-bit words. Each parameter can be any expression that evaluates to an integer, a string, or a floating point number. If it evaluates to an integer, only the lowest 32 bits are inserted. If it evaluates to a string, every character is inserted as a word using ASCII encoding. Floats are inserted using an integer representation of the single-precision float's encoding.

If No$gba semantics are enabled, then `dw` and `.dw` are treated as inserting 16-bit halfwords instead (i.e. equivalent to `.halfword`).

Write doublewords

```
.doubleword value[,...]
.dd         value[,...]
.dcq        value[,...]
dd          value[,...]
dcq         value[,...]
```

Inserts the specified sequence of 64-bit doublewords. Each parameter can be any expression that evaluates to an integer, a string, or a floating point number. If it evaluates to a string, every character is inserted as a doubleword using ASCII encoding. Floats are inserted using an integer representation of the double-precision float's encoding.

If No$gba semantics are enabled, then `dd` and `.dd` are treated as inserting 32-bit words instead (i.e. equivalent to `.word`).

Write floating point numbers

```
.float  value[,...]
.double value[,...]
```

`.float` inserts the specified sequence of single-precision floats and `.double` inserts double-precision floats. Each parameter can be any expression that evaluates to an integer or a floating point number. If it evaluates to an integer, it will be converted to a floating point number of that value.

Load a table specifying a custom encoding

```
.loadtable TableName[,encoding]
.table     TableName[,encoding]
```

Loads `TableName` for using it with the `.string` directive. The encoding can be specified in the same way as for `.include`.

The table file format is a line-separated list of key values specified by `hexbyte=string` and optional termination byte sequence by `/hexbytes`

```
02=a
1D=the
2F=you
/FF
```

`FF` will be used as the termination sequence. If it is not given, zero is used instead. Strings are matched using the longest prefix found in the table.

Write text with custom encoding

```
.string  "String"[,...]
.stringn "String"[,...]
.str     "String"[,...]
.strn    "String"[,...]
```

Inserts the given string using the encoding from the currently loaded table. `.string` and `.str` insert the termination sequence specified by the table after the string, while `.stringn` and `.strn` omit it.

Write text with Shift-JIS encoding

```
.sjis  "String"[,...]
.sjisn "String"[,...]
```

Inserts the given string using the Shift-JIS encoding. `.sjis` inserts a null byte after the string, while `.sjisn` omits it.

Conditional directives

Begin a conditional block

```
.if     cond
.ifdef  symbol
.ifndef symbol
```

The content of a conditional block will only be used if the condition is met. In the case of `.if`, it is met if `cond` evaluates to a non-zero integer. `.ifdef` is met if the given symbol (such as a label) is defined anywhere in the code, and `.ifndef` if it is not.

Else case of a conditional block

```
.else
.elseif     cond
.elseifdef  symbol
.elseifndef symbol
```

The else block is used if the condition of the condition of the if block was not met. `.else` unconditionally inserts the content of the else block, while the others start a new if block and work as described before.

End a conditional block

```
.endif
```

Ends the last open if or else block.

Define labels

```
.definelabel Label,value
```

Defines `Label` with a given value, creating a symbol for it. This can be used similar to `equ`, but symbols can be used before labels are defined and can be used in conjunction with the `.ifdef/.ifndef` conditionals. These can also be useful for declaring symbols for existing code and data when inserting new code.

Unlike `Label:`, note that `.definelabel Label,value` is evaluated only once, thus using any expressions that refer to the current state of the assembler (e.g. `org()`, `.`) in combination with `.definelabel` leads to undefined behavior.

Function labels

```
.func     Label
.function Label
```

Creates a symbol `Label` with the current memory address as its value. This is equivalent to `Label:`. However, used in conjunction with the `-sym2` command line flag, the size of the function block will also be written to the symfile along with its memory location. A function block must be terminated with `.endfunc/.endfunction`. This is also implicitly invoked when starting another function block.

```
.func Function1
  ; assembly code
.func Function2
  ; assembly code
.endfunc
```

Areas

```
.area SizeEquation[,fill]
.endarea
```

Opens a new area with the maximum size of `SizeEquation`. If the data inside the area is longer than this maximum size, the assembler will output an error and refuse to assemble the code. The area is closed with the `.endarea` directive and if the `fill` parameter is provided, the remaining free space in the area will be filled with bytes of that value.

Messages

```
.warning "Message"
.error   "Message"
.notice  "Message"
```

Prints the message and sets warning/error flags. Useful with conditionals.

Error on warning

```
.erroronwarning on
.erroronwarning off
```

By specifying `.erroronwarning on`, any warnings emitted by the assembler will be promoted to errors. Errors cause armips to abort the assembly process return a nonzero exit code. This property can also be enabled from the command line with the `-erroronwarning` flag, and can be turned off again with `.erroronwarning off`. By default, this feature is off.

Relative paths

```
.relativeinclude on
.relativeinclude off
```

By default, any paths used in assembly files (such as for `.open`, `.include`, etc.) are treated as relative to the current working directory. By specifying `.relativeinclude on`, any paths specified after it will instead be treated as relative to the path of the current assembly file that uses the path. This can be turned off again with `.relativeinclude off`. By default, this feature is off.

No$gba semantics

```
.nocash on
.nocash off
```

By specifying `.nocash on`, No$gba semantics will be enabled for data directives. This has the effect that `dh` / `.dh` will fail, `dw` / `.dw` will write 16-bit halfwords instead of 32-bit words, and `dd` / `.dd` will write 32-bit words instead of 64-bit doublewords. It can be turned off again with `.nocash off`. By default, this feature is off.

Enable/disable symfile writing

```
.sym on
.sym off
```

By specifying `.sym off`, any symbols (e.g. labels) defined after it will not be written to the symfile (if specified with the `-sym` / `-sym2` command line flag). This can be useful when using labels to define enum values that should not be interpreted as memory addresses. Writing to the symfile can be enabled again with `.sym on`. By default, this feature is on.

5.2 MIPS directives

Load delay

```
.resetdelay
```

Resets the current load delay status. This can be useful if the instruction after a delay slot access the delayed register, as the assembler can't detect that yet.

```
.fixloaddelay
```

Automatically fixes any load delay problems by inserting a `nop` between the instructions. Best used in combination with `.resetdelay`.

```
.loadelf name[,outputname]
```

Opens the specified ELF file for output. If two file names are specified, then the assembler will copy the first file to the second path. If relative include is off, all paths are relative to the current working directory, so from where the assembler was called. Otherwise the path is relative to the including assembly file. All segments are accessible by their virtual addresses, and all unmapped sections can be accessed by their physical position (through `.orga`). Currently this is only supported for the PSP architecture, and only for non-relocateable files. The internal structure of the file may be changed during the process, but this should not affect its behavior.

5.3 ARM Directives

Change instruction set

```
.arm
.thumb
```

These directives can be used to select the ARM or THUMB instruction set. `.arm` tells the assembler to use the full 32 bit ARM instruction set, while `.thumb` uses the cut-down 16 bit THUMB instruction set.

Pools

```
.pool
```

This directive works together with the pseudo opcode `ldr rx,=value`. The immediate is added to the nearest pool that follows it, and the instruction is turned into a PC relative load. The range is limited, so you may have to define several pools. Example:

```
ldr  r0,=0xFFEEDDCC
; ...
.pool
```

`.pool` will automatically align the memory position to a multiple of 4 before writing the pool.

Debug messages

```
.msg
```

Inserts a no$gba debug message as described by GBATEK.

6. Macros

6.1 Assembler-defined MIPS macros

There are various macros built into the assembler for ease of use. They are intended to make using some of the assembly simpler and faster. At the moment, these are all the MIPS macros included:

Immediate macros

```
li   reg,Immediate
la   reg,Immediate
```

Loads Immediate into the specified register by using a combination of `lui` / `ori`, a simple `addiu`, or a simple `ori`, depending on the value of the Immediate.

Immediate float macros

```
li.s reg,Immediate
```

Loads float value Immediate into the specified FP register by using a combination of `li` and `mtc1`.

Memory macros

```
lb   reg,Address
lbu  reg,Address
lh   reg,Address
lhu  reg,Address
```

```
lw   reg,Address
lwu  reg,Address
ld   reg,Address
lwc1 reg,Address
lwc2 reg,Address
ldc1 reg,Address
ldc2 reg,Address
```

Loads a byte/halfword/word from the given address into the specified register by using a combination of `lui` and `lb` / `lbu` / `lh` / `lhu` / `lw` / `ld` / `lwc1` / `lwc2` / `ldc1` / `ldc2` .

```
ulh  destreg,imm(sourcereg)
ulh  destreg,(sourcereg)
ulhu destreg,imm(sourcereg)
ulhu destreg,(sourcereg)
ulw  destreg,imm(sourcereg)
ulw  destreg,(sourcereg)
uld  destreg,imm(sourcereg)
uld  destreg,(sourcereg)
```

Loads an unaligned halfword/word/doubleword from the address in sourcereg by using a combination of several `lb` / `lbu` and `ori` or `lwl` / `lwr` or `ldl` / `ldr` instructions.

```
sb   reg,Address
sh   reg,Address
sw   reg,Address
sd   reg,Address
swc1 reg,Address
swc2 reg,Address
sdc1 reg,Address
sdc2 reg,Address
```

Stores a byte/halfword/word/doubleword to the given address by using a combination of `lui` and `sb` / `sh` / `sw` / `sd` / `swc1` / `swc2` / `sdc1` / `sdc2` .

```
ush  destreg,imm(sourcereg)
ush  destreg,(sourcereg)
usw  destreg,imm(sourcereg)
usw  destreg,(sourcereg)
usd  destreg,imm(sourcereg)
usd  destreg,(sourcereg)
```

Stores an unaligned halfword/word/doubleword to the address in sourcereg using a combination of several `sb` / `sbu` and shifts or `swl` / `swr` / `sdl` / `sdr` instructions.

Branch macros

```
blt   reg1,reg2,Dest
bltu  reg1,reg2,Dest
bgt   reg1,reg2,Dest
bgtu  reg1,reg2,Dest
bge   reg1,reg2,Dest
bgeu  reg1,reg2,Dest
ble   reg1,reg2,Dest
bleu  reg1,reg2,Dest
bltl  reg1,reg2,Dest
bltul reg1,reg2,Dest
bgtl  reg1,reg2,Dest
bgtul reg1,reg2,Dest
bgel  reg1,reg2,Dest
bgeul reg1,reg2,Dest
blel  reg1,reg2,Dest
bleul reg1,reg2,Dest
blt   reg,Imm,Dest
bltu  reg,Imm,Dest
bgt   reg,Imm,Dest
bgtu  reg,Imm,Dest
bge   reg,Imm,Dest
bgeu  reg,Imm,Dest
ble   reg,Imm,Dest
bleu  reg,Imm,Dest
bne   reg,Imm,Dest
beq   reg,Imm,Dest
bltl  reg,Imm,Dest
bltul reg,Imm,Dest
bgtl  reg,Imm,Dest
bgtul reg,Imm,Dest
bgel  reg,Imm,Dest
bgeul reg,Imm,Dest
blel  reg,Imm,Dest
bleul reg,Imm,Dest
bnel  reg,Imm,Dest
beql  reg,Imm,Dest
```

If reg/reg1 is less than/greater than or equal to/equal to/not equal to reg2/Imm, branches to the given address. A combination of `sltu` and `beq` / `bne` or `li` , `sltu` and `beq` / `bne` is used.

Set macros

```
slt   reg1,reg2,Imm
sltu  reg1,reg2,Imm
sgt   reg1,reg2,Imm
sgtu  reg1,reg2,Imm
sge   reg1,reg2,Imm
sgeu  reg1,reg2,Imm
sle   reg1,reg2,Imm
sleu  reg1,reg2,Imm
sne   reg1,reg2,Imm
seq   reg1,reg2,Imm
sge   reg1,reg2,reg3
sgeu  reg1,reg2,reg3
sle   reg1,reg2,reg3
sleu  reg1,reg2,reg3
sne   reg1,reg2,reg3
seq   reg1,reg2,reg3
```

If reg2 is less than/greater than or equal to/equal to/not equal to reg3/Imm, sets reg1 to `1`, otherwise sets reg1 to `0`. Various combinations of `li`, `slt` / `sltu` / `slti` / `sltiu` and `xor` / `xori` are used.

Rotate macros

```
rol    reg1,reg2,reg3
ror    reg1,reg2,reg3
rol    reg1,reg2,Imm
ror    reg1,reg2,Imm
```

Rotates reg2 left/right by the value of the lower 5 bits of reg3/Imm and stores the result in reg1. A combination of `sll`, `srl` and `or` is used.

Absolute value macros

```
abs   reg1,reg2
dabs  reg1,reg2
```

Stores absolute value of word/doubleword in reg2 into reg1 using a combination of `sra` / `dsra32`, `xor`, and `subu` / `dsubu`.

Upper/lower versions

Additionally, there are upper and lower versions for many two opcode macros. They have the same names and parameters as the normal versions, but `.u` or `.l` is appended at the end of the name. For example, `li.u` will output the upper half of the `li` macro, and `li.l` will output the lower half. The following macros support this:
`li` , `la` , `lb` , `lbu` , `lh` , `lhu` , `lw` , `lwu` , `ld` , `lwc1` , `lwc2` , `ldc1` , `ldc2` , `sb` , `sh` , `sw` , `sd` , `swc1` , `swc2` , `sdc1` , `sdc2`

This can be used when the two halves of the macros need to be used in nonconsecutive positions, for example:

```
li.u  a0,address
jal   function
li.l  a0,address
```

6.2 Assembler-defined ARM macros

The assembler will automatically convert the arguments between the following opcodes if possible:

```
mov <-> mvn
bic <-> and
cmp <-> cmn
```

E.g., `mov r0,-1` will be assembled as `mvn r0,0`

Additionally, `ldr rx,=immediate` can be used to load a 32-bit immediate. The assembler will try to convert it into a mov/mvn instruction if possible. Otherwise, it will be stored in the nearest pool (see the .pool directive). `add rx,=immediate` can be used as a PC-relative add and will be assembled as `add rx,r15,(immediate-.-8)`

6.3 User defined macros

The assembler allows the creation of custom macros. This is an example macro, a recreation of the builtin MIPS macro `li` :

```
.macro myli,dest,value
   .if value & ~0xFFFF
      ori   dest,r0,value
   .elseif (value & 0xFFFF8000) == 0xFFFF8000
      addiu dest,r0,value & 0xFFFF
```

```
    .elseif (value & 0xFFFF) == 0
       lui   dest,value >> 16
    .else
       lui   dest,value >> 16 + (value & 0x8000 != 0)
       addiu dest,dest,value & 0xFFFF
    .endif
.endmacro
```

The macro has to be initiated by a `.macro` directive. The first argument is the macro name, followed by a variable amount of arguments. The code inside the macro can be anything, and it can even call other macros (up to a nesting level of 128 calls). The macro is terminated by a `.endmacro` directive. It is not assembled when it is defined, but other code can call it from then on. All arguments are simple text replacements, so they can be anything from a number to a whole instruction parameter list. The macro is then invoked like this:

```
myli  a0,0xFFEEDDCC
```

In this case, the code will assemble to the following:

```
lui   a0,0xFFEF
addiu a0,a0,0xDDCC
```

Like all the other code, any equs are inserted before they are resolved.

Macros can also contain global, static and local labels that are changed to an unique name. The label name is prefixed by the macro name and a counter is appended. This label:

```
.macro Test
  @@MainLoop:
.endmacro
```

will therefore be changed to the following (note that label names are case insensitive):

```
@@test_mainloop_00000000:
```

Each call of the macro will increase the counter. The counter is output as a hexadecimal number, e.g. the eleventh call of the `test` macro will create a label named:

```
@@test_mainloop_000000a
```

Static labels defined inside a (nested) macro are treated as if they were defined inside the file that called the macro.

It is possible to pass an as-of-yet undefined symbol identifier to a macro and define the symbol as a label inside the macro. For example, the following:

```
.macro function,name
  .align 4
  name:
.endmacro

.org 0x2000002
function Main
```

will align the memory address to a multiple of 4, then create a label named `Main`, which will have value `0x2000004` as a result.

7. Meta

7.1 Change log

- Version 0.11
  - new `.aligna` directive for absolute address alignment
  - new expression functions: `org(label)`, `orga(label)`, `headersize(label)`
  - new expression functions: `min` and `max`
  - fixed major bug in MIPS LO/HI ELF symbol relocation
  - COP2, TLB*, RFE instructions added to PSX
  - fixed output of RSP VMOV/VRSQ*/VRCP* instructions
  - RSP CTC2/CFC2 control register name support added
  - fixed edge case bugs in ARM shift handling
  - new `-definelabel` command line argument for defining label
  - `-equ` command line option now normalizes the case of the name
  - additional validations of command line arguments
  - relativeinclude settings now respected in table directives
  - fixed bugs in float exponential notation parsing
  - fixed NaN and string comparisons with `<`, `<=`, `>`, `>=`
  - negative initial header sizes now allowed (with warnings)

    - correct line and column numbers for equ invocations
    - other bugfixes and enhancements
- Version 0.10
    - many bugfixes and enhancements
    - several new MIPS macros and pseudo-ops
    - improved command argument handling, allows the input file argument to be after flag arguments and detects errors better
    - C-style block comments supported
    - expression values are now signed
    - 64-bit data defines now written in generated symbol files, same command as 32-bit for compatibility with current emulators
    - ELF relocator now checks object file machine and endianness before linking
    - new directives: `.asciiz`, `.skip`
    - new expression functions: `hi` (MIPS only), `lo` (MIPS only), `reads{8,16,32,64}`
    - float division by zero in expression now has standard float behaviour (returns `±∞` or `NaN`), while integer divisions by zero returns dummy value `-1`
    - exponential notation for floats supported
- Version 0.9
    - huge rewrite with many enhancements and fixes
    - can now read from UTF8, UTF16, and Shift-JIS files and convert the input correctly
    - several new MIPS pseudo-ops, new COP0 and FPU control register types
    - Nintendo 64 CPU + RSP support
    - PSP support, load ELFs with `.loadelf`
    - able to import and relocate static C/C++ libraries
    - new `-sym2` format for use with PPSSPP and PCSX2
    - new directives: `.sym`, `.stringn`, `.sjis`, `.sjisn`, `.function`, `.endfunction`, `.importlib`, `.loadelf`, `.float`, `.dd`, `.double`
    - removed directives: `.ifarm`, `.ifthumb`, `.radix`
    - added support for floats in data directives
    - added expression functions
    - variable expressions supported in `.org` / `.orga` / `.headersize`
    - new statement syntax with `::` as separator and `\` as line continuation.
- Version 0.7d
    - added automatic optimizations for several ARM opcodes
    - many bugfixes and internal changes
    - added static labels

    - new directives: `.warning`, `.error`, `.notice`, `.relativeinclude`, `.erroronwarning`, `.ifarm`, `.ifthumb`
    - quotation marks can now be escaped in strings using `\"`.
- Version 0.7c
    - Macros can now contain unique local labels
    - `.area` directive added
    - countless bugfixes
    - no$gba debug message support
    - full no$gba sym support
- Version 0.7b
    - ARM/THUMB support
    - fixed break/syscall MIPS opcodes
    - added check if a MIPS instruction is valid inside a delay slot
    - fixed and extended base detection
    - added `.` dummy label to the math parser to get the current memory address
    - added `dcb` / `dcw` / `dcd` directives
- Version 0.5b
    - Initial release

7.2 Migration from older versions

There are several changes after version 0.7d that may break compatibility with code written for older versions. These are as follows:

- String literals now require quotation marks, e.g. for file names
- `$xx` is no longer supported for hexadecimal literals

7.3 License

Releases 2

v0.11.0 Latest
on May 10, 2020

+ 1 release

Packages

No packages published

Contributors 28

+ 17 contributors

Languages

- C++ 92.9%
- Assembly 4.5%
- CMake 1.1%
- Other 1.5%