MIPS TO C-CODE CONVERSION

Given some MIPS assembly, this program will attempt to convert it to C. The goal of this project is to support de-compilation projects, which aim to write C code that yields byte-identical output when compiled with a particular build system. It primarily focuses on supporting popular compilers of the late 1990's.However, it may also work with other compilers or hand-written assembly. The focus of `mips-to-c` is to aid in the process of producing "matching" C source files. These differentiates it from other de-compilation suites, such as IDA or Ghidra . Right now the de-compiler is fairly functional, though it sometimes generates suboptimal code (especially for loops).The input is expected to match a particular assembly format, such as that produced by tools like [`mipsdisasm`](https://github.com/queueRAM/sm64tools).

See the `tests/` directory for some example input and output.

Install

This project requires Python 3.6 or later. To install the Python dependencies:

```bash

python3 -m pip install --upgrade pycparser

Optional: If you are on python3.6, you will also need to install "dataclasses"

python3.6 -m pip install --upgrade dataclasses

You might need to install `pip` first; on Ubuntu this can be done with:

```bash

sudo apt update

sudo apt install python3-pip

```

Usage

```bash

python3 mips\_to\_c.py [options] [--context <context file>] [-f <function name>] <asmfile>...

```

Run with `--help` to see which options are available.

Context files provided with `--context` are parsed and cached, so subsequent runs with the same file are faster. The cache for `foo/bar.c` is stored in `foo/bar.m2c`. These files can be ignored (added to `.gitignore`), and are automatically regenerated if context files change. Caching can be disabled with the `--no-cache` argument.

Multiple functions

By default, `mips\_to\_c` decompiles all functions in the text sections from the input assembly files.`mips\_to\_c` is able to perform a small amount of cross-function type inference, if the functions call each other.You can limit the function(s) that decompiled by providing the `-f <function name>` flags (or the "Function" dropdown on the website).

Global Declarations & Initializers

When provided input files with `data`, `rodata`, and/or `bss` sections, `mips\_to\_c` can generate the initializers for variables it knows the types of.Qualifier hints such as `const`, `static`, and `extern` are based on which sections the symbols appear in, or if they aren't provided at all.The output also includes prototypes for functions not declared in the context.`mips\_to\_c` cannot generate initializers for structs with bitfields (e.g. `unsigned foo: 3;`) or for symbols that it cannot infer the type of.For the latter, you can provide a type for the symbol the context.This feature is controlled with the `--globals` option (or "Global declarations" on the website):- `--globals=used` is the default behavior, global declarations are emitted for referenced symbols. Initializers are generated when the data/rodata sections are provided.- `--globals=none` disables globals entirely; only function definitions are emitted.- `--globals=all` includes all of the output in `used`, but also includes initializers for unreferenced symbols. This can be used to convert data/rodata files without decompiling any functions.

Struct Field Inference

By default, `mips\_to\_c` can use type information from decompiled functions to help fill in unknown struct fields.

This behavior can be disabled with `--no-unk-inference` ("Disable unknown struct/type inference" on the website).For structs in the context, the following fields treated as "unknown" space that can be inferred:

- `char` arrays with a name starting with `unk\_`, e.g. `char unk\_10[4];`

- any field with a type that starts with `UNK\_` or `MIPS2C\_UNK`, e.g. `UNK\_TYPE4 foo;`

Currently, struct field inference only works on structs without bitfields or [unnamed union fields](https://gcc.gnu.org/onlinedocs/gcc/Unnamed-Fields.html).

The output will include declarations for any struct with at least one inferred field.

Specifying stack variables

By default, `mips\_to\_c` infers the types of stack (local) variables, and names them with the `sp` prefix based on their offset.

Internally, the stack is represented as a struct, so it is possible to manually specify the names & types of stack variables by providing a struct declaration in the context. `mips\_to\_c` looks in the context for a struct with the tag name `\_mips2c\_stack\_<function name>` (e.g. `struct \_mips2c\_stack\_test` for a function `test()`).The size of the stack must exactly match the detected frame size, or `mips\_to\_c` will return an error.If you run `mips\_to\_c` with the `--stack-structs` option ("Stack struct templates" on the website), the output will include the inferred stack declaration, which can then be edited and provided as context by re-running `mips\_to\_c`.

Example

Here is an example for specifying the stack for the `custom\_stack` end-to-end test.First, run `mips\_to\_c` with the `--stack-structs` option to get the inferred struct for the `test()` function:

<details>

<summary><code>python3 mips\_to\_c.py tests/end\_to\_end/custom\_stack/irix-o2.s -f test --stack-structs</code></summary>

```c

struct \_mips2c\_stack\_test {

/\* 0x00 \*/ char pad0[0x20];

/\* 0x20 \*/ s8 sp20; /\* inferred \*/

/\* 0x21 \*/ char pad21[0x3]; /\* maybe part of sp20[4]? \*/

/\* 0x24 \*/ s32 sp24; /\* inferred \*/

/\* 0x28 \*/ s32 sp28; /\* inferred \*/

/\* 0x2C \*/ s8 sp2C; /\* inferred \*/

/\* 0x2D \*/ char pad2D[0x3]; /\* maybe part of sp2C[4]? \*/

/\* 0x30 \*/ s8 sp30; /\* inferred \*/

/\* 0x31 \*/ char pad31[0x3]; /\* maybe part of sp30[4]? \*/

/\* 0x34 \*/ s8 sp34; /\* inferred \*/

/\* 0x35 \*/ char pad35[0x2]; /\* maybe part of sp34[3]? \*/

/\* 0x37 \*/ s8 sp37; /\* inferred \*/

}; /\* size = 0x38 \*/

? func\_00400090(s8 \*); /\* static \*/

s32 test(void \*arg0); /\* static \*/

s32 test(void \*arg0) {

s8 sp37;

s8 sp34;

s8 sp30;

s8 sp2C;

s32 sp28;

s32 sp24;

s8 sp20;

s32 temp\_t4;

func\_00400090(&sp37);

func\_00400090(&sp34);

func\_00400090(&sp30);

func\_00400090(&sp2C);

func\_00400090(&sp20);

sp37 = arg0->unk0 + arg0->unk4;

sp34 = arg0->unk0 + arg0->unk8;

temp\_t4 = arg0->unk4 + arg0->unk8;

sp30 = temp\_t4;

sp20 = arg0->unk0 \* sp37;

sp24 = arg0->unk4 \* (s16) sp34;

sp28 = arg0->unk8 \* temp\_t4;

if (sp37 != 0) {

sp2C = arg0;

} else {

sp2C = &sp20;

}

return sp37 + (s16) sp34 + (s32) sp30 + \*(s32 \*) sp2C + sp24;

}

```

</details>

Now, based on the body of the `test()` function, we can make some guesses about the types of these variables, and give them more descriptive names:

```c

// Save this file as `test\_context.c`

struct Vec {

s32 x, y, z;

};

struct \_mips2c\_stack\_test {

char pad0[0x20];

struct Vec vec;

struct Vec \*vec\_ptr;

s32 scale\_z;

s16 scale\_y;

char pad36[1];

s8 scale\_x;

}; /\* size 0x38 \*/

int test(struct Vec \*vec\_arg);

```

Finally, re-run `mips\_to\_c` with our custom stack as part of the `--context`. The `--context` option can be specified multiple times to combine files.

<details>

<summary><code>python3 mips\_to\_c.py tests/end\_to\_end/custom\_stack/irix-o2.s -f test --context test\_context.c</code></summary>

```c

? func\_00400090(s8 \*); /\* static \*/

s32 test(struct Vec \*vec\_arg) {

s8 scale\_x;

s16 scale\_y;

s32 scale\_z;

struct Vec \*vec\_ptr;

struct Vec vec;

s32 temp\_t4;

func\_00400090(&scale\_x);

func\_00400090((s8 \*) &scale\_y);

func\_00400090((s8 \*) &scale\_z);

func\_00400090((s8 \*) &vec\_ptr);

func\_00400090((s8 \*) &vec);

scale\_x = vec\_arg->x + vec\_arg->y;

scale\_y = vec\_arg->x + vec\_arg->z;

temp\_t4 = vec\_arg->y + vec\_arg->z;

scale\_z = temp\_t4;

vec = vec\_arg->x \* scale\_x;

vec.y = vec\_arg->y \* scale\_y;

vec.z = vec\_arg->z \* temp\_t4;

if (scale\_x != 0) {

vec\_ptr = vec\_arg;

} else {

vec\_ptr = &vec;

}

return scale\_x + scale\_y + scale\_z + vec\_ptr->x + vec.y;

}

```

</details>

### Formatting

The following options control the formatting details of the output, such as braces style or numeric format. See `./mips\_to\_c.py --help` for more details.

(The option name on the website, if available, is in parentheses.)

- `--valid-syntax`

- `--allman` ("Allman braces")

- `--pointer-style` ("`\*` to the left")

- `--unk-underscore`

- `--hex-case`

- `--comment-style {multiline,oneline,none}` ("Comment style")

- `--comment-column N` ("Comment style")

- `--no-casts`

Note: `--valid-syntax` is used to produce output that is less human-readable, but is likely to directly compile without edits. This can be used to go directly from assembly to the permuter without human intervention.

### Debugging poor results (Advanced)

There are several options to `mips\_to\_c` which can be used to troubleshoot poor results. Many of these options produce more "primitive" output or debugging information.

- `--no-andor` ("Disable &&/||"): Disable complex conditional detection, such as `if (a && b)`. Instead, emit each part of the conditional as a separate `if` statement. Ands, ors, nots, etc. are usually represented with `goto`s.

- `--no-switches` ("Disable irregular switch detection"): Disable "irregular" `switch` statements, where the compiler emits a single `switch` as a series of branches and/or jump tables. By default, these are coalesced into a single `switch` and marked with an `/\* irregular \*/` comment.

- `--no-unk-inference` ("Disable unknown struct/type inference"): Disable attempting to infer struct fields/types in unknown struct sections and global symbols. See the [\_Struct Field Inference\_](#struct-field-inference) section above.

- `--gotos-only` ("Use gotos for everything"): Do not detect loops or complex conditionals. This format is close to a 1-1 translation of the assembly.

- Note: to use a goto for a single branch, don't use this flag, but add `# GOTO` to the assembly input.

- `--debug` ("Debug info"): include debug information inline with the code, such as basic block boundaries & labels.

- `--void` ("Force void return type"): assume that the decompiled function has return type `void`. Alternatively: provide the function prototype in the context.

#### Visualization

`mips\_to\_c` can generate an SVG representation of the control flow of a function, which can sometimes be helpful to untangle complex loops or early returns.

Pass `--visualize` on the command line, or use the "Visualize" button on the website. The output will be an SVG file.

Example to produce `my\_fn.svg` of `my\_fn()`:

```sh

python3 ./mips\_to\_c.py --visualize --context ctx.c -f my\_fn my\_asm.s > my\_fn.svg

```

## Contributing

There is much low-hanging fruit still. Take a look at the issues if you want to help out.

We use `black` to auto-format our code and `mypy` for type checking. We recommend using `pre-commit` to ensure only auto-formatted code is committed. To set these up, run:

```bash

pip install pre-commit black mypy

pre-commit install

```

Your commits will then be automatically formatted per commit. You can also manually run `black` on the command-line.

Type annotations are used for all Python code. `mypy` should pass without any errors.

To get pretty graph visualizations, install `graphviz` using `pip` and globally on your system (e.g. `sudo apt install graphviz`), and pass the `--visualize` flag.

## Tests

There is a small test suite, which works as follows:

- As you develop your commit, occasionally run `./run\_tests.py` to see if any tests have changed output.

These tests run the decompiler on a small corpus of IDO 5.3-compiled MIPS assembly.

- Before pushing your commit, run `./run\_tests.py --overwrite` to write changed tests to disk, and commit resultant changes.

### Running Decompilation Project Tests

It's possible to use the entire corpus of assembly files from decompilation projects as regression tests.

For now, the output of these tests are not tracked in version control.

You need to run `./run\_tests.py --overwrite ...` \*\*before\*\* making any code changes to create the baseline output.

As an example, if you have the `oot` project cloned locally in the parent directory containing `mips\_to\_c`, the following will decompile all of its assembly files.

```bash

./run\_tests.py --project ../oot --project-with-context ../oot

```

- Need to use the `ver/us` or `ver/jp` subfolder, e.g. `--project ../papermario/ver/us`

#### Creating Context Files

The following bash can be used in each decompilation project to create a "universal" context file that can be used for decompiling any assembly file in the project.

This creates `ctx.c` in the project directory.

```bash

cd mm # Or oot, papermario, etc.

find include/ src/ -type f -name "\*.h" | sed -e 's/.\*/#include "\0"/' > ctx\_includes.c

tools/m2ctx.py ctx\_includes.c

```

#### Notes for Majora's Mask

The build system in the MM decompilation project is currently being re-written.

It uses "transient" assembly that is not checked in, and in the normal build process it re-groups `.rodata` sections by function.

To use the MM project, run the following to \*just\* build the transient assembly files (and avoid running `split\_asm.py`).

```bash

cd mm

make distclean

make setup

make asm/disasm.dep

```

The repository should be setup correctly if there are `asm/code`, `asm/boot`, and `asm/overlays` folders with `.asm` files, but there \*should not\* be an `asm/non\_matchings` folder.

### Adding an End-to-End Test

You are encouraged to add new end-to-end tests using the `./tests/add\_test.py` script.

You'll need the IDO `cc` compiler and [sm64tools](https://github.com/queueRAM/sm64tools).

A good reference test to base your new test on is [`array-access`](tests/end\_to\_end/array-access).

Create a new directory in `tests/end\_to\_end`, and write the `orig.c` test case.

If you want the test to pass in C context, also add `irix-o2-flags.txt` & `irix-g-flags.txt` files.

After writing these files, run `add\_test.py` with the path to the new `orig.c` file, as shown below.

This example assumes that sm64tools is cloned & built in your home directory, and that the IDO compiler is available from the OOT decompilation project.

You should change these exported paths to match your system.

```bash

export SM64\_TOOLS=$HOME/sm64tools/build/

export IDO\_CC=$HOME/oot/tools/ido\_recomp/linux/7.1/cc

./tests/add\_test.py $PWD/tests/end\_to\_end/my-new-test/orig.c

```

This should create `irix-o2.s` and `irix-g.s` files in your test directory.

Now, run `./run\_tests.py --overwrite` to invoke the decompiler and write the output to `irix-o2-out.c` and `irix-g-out.c`.

Finally, `git add` your test to track it.

```bash

./run\_tests.py --overwrite

git add tests/end\_to\_end/my-new-test

```

CODE 1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* hanoi.c, Version 1.0, 18 Sep 1992 --- AAA \*/

/\* Version 1.1, 06 Jul 1993 --- AAA \*/

/\* Added print out for all disks \*/

/\* Version 1.2, 28 May 1994 --- AAA \*/

/\* Calculate average Moves/25usec \*/

/\* Version 1.2, 27 Oct 1994 --- AAA \*/

/\* Only avg for RunTime > 0.1 sec \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Adapted from the MUSBUS test suite. \*/

/\* hanoi.c, V3.5 87/08/06 08:11:14 kenj \*/

/\* \*/

/\* Various timer routines are included. Please \*/

/\* select the appropriate timer routine for your \*/

/\* system from the options below. If no option is \*/

/\* available then you will need to create your \*/

/\* own similar 'dtime()' timer routine. \*/

/\* \*/

/\* You can uncomment one of the timer options \*/

/\* below to use it, or you can compile with the \*/

/\* option name. For example you can compile with \*/

/\* '-DUNIX', or '-DVMS', '-DMSC',..., etc. \*/

/\* \*/

/\* Example compilation: \*/

/\* cc -DUNIX hanoi.c -o hanoi, or \*/

/\* gcc -DUNIX -O hanoi.c -o hanoi, or \*/

/\* cc -dAmiga hanoi.c -o hanoi, ..., etc \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* #include <stdio.h> \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Timer options. You MUST uncomment one of the options below \*/

/\* or compile, for example, with the '-DUNIX' option. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* #define Amiga \*/

/\* #define UNIX \*/

/\* #define UNIX\_Old \*/

/\* #define VMS \*/

/\* #define BORLAND\_C \*/

/\* #define MSC \*/

/\* #define MAC \*/

/\* #define IPSC \*/

/\* #define FORTRAN\_SEC \*/

/\* #define GTODay \*/

/\* #define CTimer \*/

/\* #define UXPM \*/

/\* #define MAC\_TMgr \*/

/\* #define PARIX \*/

/\* #define POSIX \*/

#define other(i,j) (6-(i+j))

int num[4];

long count;

void \_start()

{

register disk, Loops = 0;

disk = 1;

while ( 1 )

{

disk++;

num[0] = 0;

num[1] = disk;

num[2] = 0;

num[3] = 0;

count = 0;

mov(disk,1,3);

Loops = Loops + 1;

if ( disk == 10 ) break;

}

}

mov(n,f,t)

{

register o;

if(n == 1)

{

num[f]--;

num[t]++;

count++;

return;

}

o = other(f,t);

mov(n-1,f,o);

mov(1,f,t);

mov(n-1,o,t);

return;

}

CODE TRANSLATED

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* hanoi.c, \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Adapted from the MUSBUS test suite. \*/

/\* own similar 'dtime()' timer routine. \*/

/\* \*/

/\* You can uncomment one of the timer options \*/

/\* below to use it, or you can compile with the \*/

/\* option name. For example you can compile with \*/

/\* '-DUNIX', or '-DVMS', '-DMSC',..., etc. \*/

/\* \*/

/\* Example compilation: \*/

/\* cc -DUNIX hanoi.c -o hanoi, or \*/

/\* gcc -DUNIX -O hanoi.c -o hanoi, or \*/

/\* cc -dAmiga hanoi.c -o hanoi, ..., etc \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* #include <stdio.h> \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Timer options. You MUST uncomment one of the options below \*/

/\* or compile, for example, with the '-DUNIX' option. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* #define Amiga \*/

/\* #define UNIX \*/

/\* #define UNIX\_Old \*/

/\* #define VMS \*/

/\* #define BORLAND\_C \*/

/\* #define MSC \*/

/\* #define MAC \*/

/\* #define IPSC \*/

/\* #define FORTRAN\_SEC \*/

/\* #define GTODay \*/

/\* #define CTimer \*/

/\* #define UXPM \*/

/\* #define MAC\_TMgr \*/

/\* #define PARIX \*/

/\* #define POSIX \*/

#define other(i,j) (6-(i+j))

int num[4];

long count;

void \_start()

{

register disk, Loops = 0;

disk = 1;

while ( 1 )

{

disk++;

num[0] = 0;

num[1] = disk;

num[2] = 0;

num[3] = 0;

count = 0;

mov(disk,1,3);

Loops = Loops + 1;

if ( disk == 10 ) break;

}

}

mov(n,f,t)

{

register o;

if(n == 1)

{

num[f]--;

num[t]++;

count++;

return;

}

o = other(f,t);

mov(n-1,f,o);

mov(1,f,t);

mov(n-1,o,t);

return;

}