

Getting started with the Hotstate machine

The quickest way is to try out some of the examples.

The parser is a fully instrumented hardware implementation.

Go to `hotstate/examples/parser` (see `Example-Parser.pdf`)

The hotstate compiler takes in a subset of C and produces

- 1) the microcode to drive the machine
- 2) a large LUT to decode the input value for expressions
- 3) the parameters to build a state machine to exactly fit the code
- 4) output capable of being debugged with gdb or any other debugger
- 5) a makefile if none exists

Example:

Assume you have a program called `example.c`

if you compile this with the command

```
hotstate example.c
```

`hotstate` will print the code in a way that can be emulated by software and make a makefile if none exists.

```
# auto generated makefile
```

```
alz: simple.c
    hotstate -A simple.c
emu: simple.c
    hotstate -o emu.c simple.c
    gcc -ggdb -o emu emu.c
    gdb emu
tb: simple.c
    hotstate -T simple.c
mem_tb: simple.c
    hotstate -M simple.c
mem: simple.c
    hotstate -m simple.c
hot: simple.c
    hotstate -H simple.c
sa: simple.c
    hotstate -S simple.c
clean:
    rm -f *.mem
    rm -f *tb*.v
    rm -f *template*.v
    rm -f emu.c
    rm -f emu
    rm -f _user.o
```

```
rm -f *.hot
```

If you use the built in function `_user()` the makefile will look like this

```
# auto generated makefile
alz: simple.c
    hotstate -A simple.c
_user.o: _user.c
    gcc -ggdb -c _user.c
emu: simple.c _user.o
    hotstate -o emu.c simple.c
    gcc -ggdb _user.o -o emu emu.c
    gdb emu
tb: simple.c
    hotstate -T simple.c
mem_tb: simple.c
    hotstate -M simple.c
mem: simple.c
    hotstate -m simple.c
hot: simple.c
    hotstate -H simple.c
sa: simple.c
    hotstate -S simple.c
clean:
    rm -f *.mem
    rm -f *tb*.v
    rm -f *template*.v
    rm -f emu.c
    rm -f emu
    rm -f _user.o
    rm -f *.hot
```

If you add the builtin function `_user()` to you code and you have not edited the makefile you can delete the makefile and run

```
hotstate example.c
```

This will create a file `_user.c` and add the above lines to makefile.

In either case if you want to emulate your code in a C compiler type

```
make emu
```

This will compile your code with `gcc` and try to go into the gnu debugger `gdb`.

To simulate your code type

```
make tb, make mem_tb, or make sa
```

make tb will create a testbench where the memory files are explicitly called out in the testbench. The testbench will then load by feeding the files in for each of memories in the state machine.

make mem_tb will generate the memories files needed by your program. The generated testbench will then load these in from the external files

make sa will generate the memories files and directly load them into the state machine memories.

Each of these commands will create a example_tb.v file and a example_template.v file which will be overwritten each time. A user.v file will be generated if no user.v file exists. The user.v file is included into the testbench and allows you to customize the testbench. Do not edit the testbench itself as it will be overwritten

Create a Vivado project and add the testbench and the user.v file. Also add the IP file by adding sources hotstate/IP/*.sv.

Once in the simulator the main signals you want to look at are the clock, reset and hlt signals as well as the input variables, the output states, and the debug_adr signals.

The debug_adr corresponds to the address file in the analyze output (make alz).

You might edit the makefile and put -O in the hotstate commands.

Compare outputs of

```
hotstate -A example.c
```

and

```
hotstate -A -O example.c
```

to see what the optimizer does to the output code.