THE REFERENCE MANUAL FOR THE HOTSTATE MACHINE

A runtime loadable microcoded algorithmic state machine (RLMASM)

Revision 1.0.0

12/1/2022

Contents

Figures and tables

Introduction

Every program consists of two parts: the control plane and the data plane.

- An algorithm can be decomposed into a data plane and a control plane
- The data plane is a cloud of logic, arithmetic, and registers
- The control plane manages data movement and monitors status of the data plane

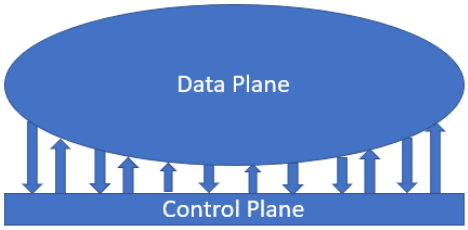

Figure 1: Algorithm data and control planes

Reconfigurable computing is about figuring out how to implement the two different planes in the most efficient manner, and then doing the same thing for the next algorithm. Processor systems use an ALU and register file to try and divide up the control and data planes into small pieces.

The Hotstate RLMASM is an enhanced Moore style state machine that implements the control plane of an algorithm. The data plane is created by the user. The RLMASM is a sequential machine with the exception that different Boolean output states can be toggled at the same time using the comma "," operator. For example state0=1, state1 = 0, state3 = 1; will set all those values in one clock where a processor would take 3 clocks.

The Hotstate machine is strictly sequential in that one address active at any one time and machine behavior is dictated by the microcode bits and various memories. The state machine will go from one address to another

The compiler will create files that contain:

1) the microcode to drive the machine
2) a large lookup table, the uberLUT, to decode the input values for expressions
3) a translation table for switch statements
4) a file containing the data to load into counter/timers
5) a verilog template with parameters to build a state machine to exactly fit the code
6) a testbench you can use for unit test
7) a user.v file that where you put your hardware test code
8) a _user.c file where you put your software test code
9) a makefile that contains common commands
10) code capable of being debugged with gdb or any other debugger

Software Architecture

The Hotstate compiler implements all the control statements in C plus limited function calls.

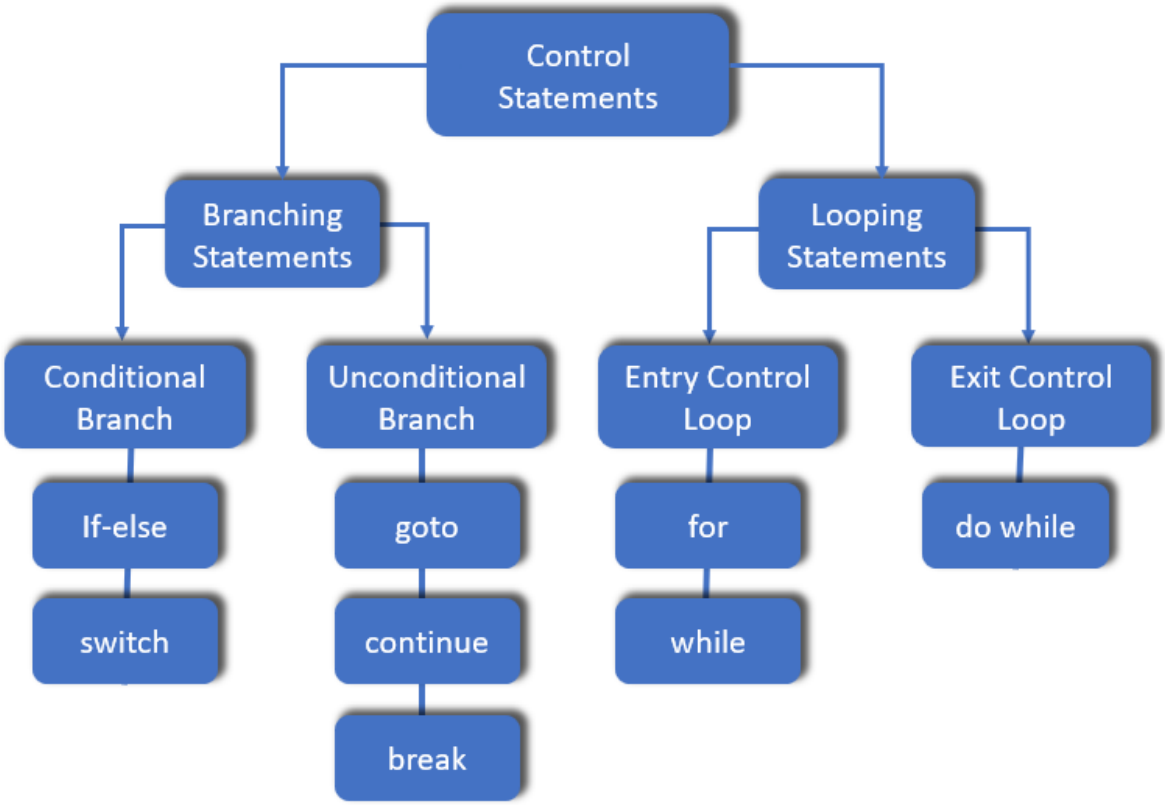

Figure 2: Supported C control statements

These are the control statements of C. There are two different kinds of control statements: statements for looping and statements for branching. Statements that evaluate expressions do so using the Boolean expressions, true or false. These are single bit inputs that you supply by connecting them to hardware status bits.

The RLMASM and some logic replaces the standard CPU core and gives better performance, 1/10 the area and power and 100% C compatibility. Because they are small, they are fast. Concurrency is implemented using multiple RLMASMs. Connecting an output state to the input of itself and/or other RLMASMs you can create a stateful network of RLMASMs. In addition to C control statements there is a limited function capability.

Since all variables are global you can't pass parameters to functions except the built-in debugging function _user(int). You can call functions from other functions and you can have recursive functions. The state machine has a stack so it can be overflown. There is an option to make the stack deeper. Later revisions with have a stack overflow flag.

Hardware Architecture

The Hotstate machine is an enhanced, highly parametrized, runtime loadable microcoded algorithmic state machine. The parameters to "size" the machine are gathered by the complier.

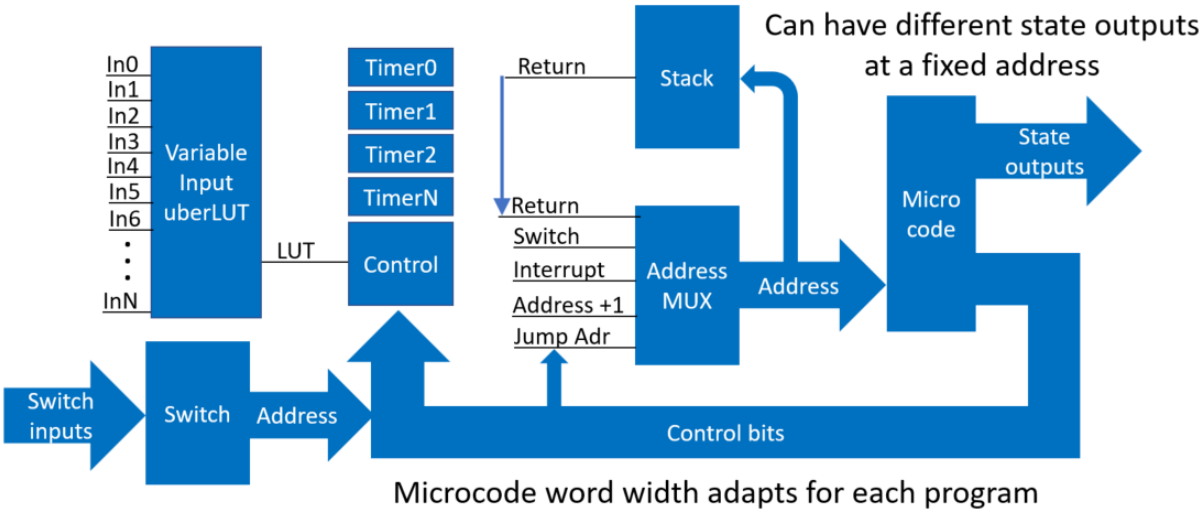

Figure 3: Hotstate block diagram

The Hotstate compiler extracts the hardware parameters from your code and then generates the microcode. The state machine has from 1 to 4 different memories. The microcode memory for control and state, the uberLUT memory for solving expressions, the timer memory holds timer data and the switch memory table holds the all the switch translation tables. These memories may or may not be generated depending on which statements you use. If you don't use the "switch" or "for" statements or function calls, those circuits and memories will not be generated.

This is not a processor core. A processor core is always the same size which means that the program counter and register file are usually 32 bits. This slows down the hardware. If you have 256 lines of microcode, then you only need an 8-bit program counter. A processor core has a register file that might have 32 32-bit registers. A processor has an ALU packed with every opcode possible. With the Hotstate machine, you add just the right amount of RTL hardware you need and let the state machine control that hardware. That might mean hundreds of adders and multipliers and registers, or a single adder and FIFO controls, all controlled via software.

Basic Types

There are 4 different types supported by the Hotstate compiler: bool, char, int and void

bool led = 1; //led is assumed to be an output state
These are the output states that will drive things like clock enables, mux controls, input to adders and any other control type signals. The states are in microcode memory.

State Outputs

To control state outputs stimulatingly use the comma operator.

LED0 = 1, LED2 = 0; // This is 100% C

These states will toggle at the same time leaving all other states quiescent.

The state outputs are qualified by the corresponding mask bit and latched if appropriate. The mask bit and state bits are combined into the state output.

state[i] = mask[i]?new_state[i]:old_state;

The number of possible number of state outputs at any one address during run time is

$$2^{(n-m)}$$

where n is the total number of states and m is the number of states used in that line of code

In the traditional algorithmic state there is one (1) state vector per address

Variable inputs

bool a0; //a0 is assumed to be an input variable
These are the variables that can be used in expressions:
if (expression) else (expression)
while (expression)
do while (expression)
Expressions can be any legal combination of C binary operators
<, <=, >, >=, ==, !=, &, |, ^, &&, ||
(a == 1 || b < 1 & c==1) is a legal expression.

To evaluate expressions the compiler will create a giant lookup table called the uberLUT. The compiler finds the eigenLength, which is 2^n, where n is the number of input variables. Each input variable is assigned an eigenLUT of eigenLength. Then, each eigenLUT for that variable is given its eigenvalue. For example if you have 3 inputs a, b, and c they each get assigned an eigenLUT value. The eigen-length is $2^3 = 8$.

eigenLUT for a	eigenLUT for b	eigenLUT for c
0	0	0
1	0	0
0	1	0
1	1	0
0	0	1
1	0	1
0	1	1
1	1	1

Figure 4: eigenLUT table for 3 variables

Each expression in your program is given an eigenLength LUT and these are all concatenated into the uberLUT. The different expressions are selected with the microcode bits varSel as the upper address in the uberLUT memory.

Case Expressions

char casein; // casein is a switch expression: The complier creates an address translation table for the switch statements. The address of the case statement is looked up using the case expression number. All cases for all switches are contained in the switch memory. This memory contains the addresses for the case statements. The default width for a switch expression is “char” which is 8-bits. This means that the address translation table will be 256 words deep, and jadr bits wide. You can use the –w option to reduce the size of this table.

If the number of bit variables is greater than 8 you must change the char to the appropriate C type to simulate. The variable must be char to go through the Hotstate compiler. Use the –w option to tell the Hotstate machine char is greater than 8. For example: –w 12.

switch (casein) {case 0: {continue} default: {continue}} This will loop forever.

For Loop Variables

int timer0; //timer0 is assumed to be used as a counter variable in “for” loops:
for (timer0=0; timer0< constant; timer0++) { }; // The only supported variant in this release. The counter/timer is a countdown counter.

void function (); // All functions must be declared “void”. All variables are global.
Functions can call other functions or recurse. There are no checks on overflow but you can set the stack depth with the –w <number> option

While there is no limit on the number of input, output, switches and timers or functions. Be careful as the machine can get very big, very fast. It's often better to have 2 or more machines for an algorithm. Each machine will run concurrently and you can have output states from one machine as inputs to others.

Microcode structure and short description of bit fields

The microcode bits control the behavior of the state machine.

```
struct MicroCode {

uint32_t state;
uint32_t mask;
uint32_t jadr;
uint32_t varSel;
uint32_t timerSel;
uint32_t timerLd;
uint32_t switch_sel;
uint32_t switch_adr;
uint32_t state_capture;
uint32_t var_or_timer;
uint32_t branch;
uint32_t forced_jmp;
uint32_t sub;
uint32_t rtn;
};
```

Bit field name	Bit field description
state	contains the state data, gets qualified by the mask
mask	mask determines if the state bit change
jadr	used for jump addresses, timer selection and switch selection
varSel	selects which expression in LUT is used
timerSel	selects which timer is in use
timerLd	loads the selected timer using timerSel
switch_sel	selects which case variable to use
switch_adr	selects case address from the switch look up table
state_capture	allow the state latches to capture state or not
var_or_timer	shared bit controls whether variables or for loops are in use
branch	branches if expressions are true or timers are done
forced_jmp	always branches to address in jadr
sub	push the stack and jump to jadr
rtn	pop the stack and return from subroutine

Figure 5: Table of bit field descriptions

Detailed bit field description

state: This field is as wide as the number of state bits in your code. They are assigned state numbers in the order in which they app in your code.

mask: This field is as wide as the state field. Each bit tells the output latches which state bits to capture.

jadr: This is a multi-use field. For any statement that has the branch or forced_jmp bit set this is the microcode address to jump to. If timerLd is set this points to the number in the timer memory that gets loaded into the countdown counter. If switch_sel is set this field holds the number of the switch statement translation field being accessed

varSel: Selects which expression will be addressed in the uberLUT. The uberLUT size is the number of expressions times the eigenLength. Suppose you have 2 input variables a0 and a1. The eigenLength will be 4. EigenLUT a0 = 0101 and a1 = 0011. Say your program looks like

void main () {

while (1) //varSel is 0

if (a0 != a1) state = 0; // varSel is 1

}

the uberLUT is (msb) 01101111 when varSel = 0 the uberLUT output is 1 and when varSel is 1 a0 and a1 follow 0110 so the uberLUT output is true when a0 = 0 and a1 = 1 or a0 = 1 and a1 = 0. This is the exclusive or (XOR) function.

timerSel: Each timer gets its own bit. If there are n timers this field and it is n bits wide. Only one timerSel bit is active at an address. The lsb is the first variable declared as “int”. When a bit is 1 the counter that count is decremented for that variable or, if timerLd is 1 the counter is loaded.

timerLd: The timerLd (load) field is the same width as timerSel. Like timerSel there is one bit for each “for” loop variable. When a timer’s timerLd bit is 1 the timer is loaded from the timer memory. The address is based on the jadr field.

switch_sel: Selects for the input switch expression to use in the uberSwitchmem. This signal is an output and is used in the Verilog templet to select input via a mux. Width depends on the numbers of switch variables.

switch_adr: This is a single bit that selects the address in the uberSwitchmem for the next address mux.

state_capture: Signals the state capture latches to activate. State output will only change when this bit is 1.

var_or_timer: Selects where the control block looks at the output of the uberLUT or looks for the timer done signal for its jump condition. The output of the uberLUT is the evaluation of expression in the code. The timer done signal is true when the timer counts down to zero.

branch: When branch is true, a combination of varSel, the input variables and the uberLUT are examined to determine the branch jump. If the expression is false then the state machine will jump to the address in the jadr field. If the expression is true the state machine will go to the next address.

forced_jmp: Always forces a jump to the address in the jadr field.

sub: This is used with the forced_jmp bit to push the current address onto the stack and jump to the address in the jadr field

rtn: This bit selects the address at the top of the stack as the next address. In the same clock it pops the stack.

Compiler options

The compiler will print out your comma code program so it can be debugged in any C/C++ debugger. The only difference is the compiler will set the input variables to 0 and it will promote switch variable from a char to an int if the –w option is greater than 8. The compiler uses the lack of initializers to infer that a variable is an input.

```
hotstate <opt> <file.c>
-A Print out analysis
-a Limit code printout to 32 chars during analysis
-O Optimizes by attempting to reduce code size
-T Output verilog testbench files with memory inits
-M Print out testbench and memory files
-m Print out memory files only
-H Printout hardware object file <file.hot>
-S Standalone - compile without run time loading
-o Output file name
-s <number> Set the stack_depth, default is 4
-t <number> Set the timer width, default is 32
-w <number> Set the switch width, default is 8
-v Version
-h Help
```

-A	Prints out the microcode. Each line of microcode has the comma code that line is derived from.
-a	Same as –A except the code output is limited to 32 characters as some expressions can be 1000s of characters long (see parser).
-O	Optimizes the code by merging lines. Use this most of the time
-T	Print out Verilog testbench where the different memory arrays are in the initial part of the testbench.
-M	Print out Verilog testbench like –T but outputs any of the needed mem files to initialize the memories. This testbench uses readmemh or readmemb to read in these files.
-m	Only print out the memory files needed by readmemh or readmemb.
-H	Print out the memory files in a format that can be compiled into a program.
-S	Standalone loads the memory files directly into the state machine. This eliminates all the runtime loading circuits. This is the fastest and smallest implementation and easiest way to integrate into your design.
-s	Set the stack depth. Default is 4.
-o	Output file name. This works for all other options.
-t	Sets the size of the counters. Default is 32 which is big (2^32). Making this smaller will speed up the design. This applies to all “for” loop counters in the program.
-w	Set the size of the switch expression. Default is 8 bits. Setting to a smaller size reduces the switch address translation tables.
-v	Print the version of the Hotstate compiler.
-h	Prints the help message.

Using the compiler without any options will print out a version suitable for emulation. This means that input variables will be initialized to 0 and #define bool _Bool will be inserted and, if –w <number> is greater than 8, the switch expressions will be promoted to “int” by the Hotstate compiler. To emulate the switch will have to be changed to a larger variable type and changed back to compile with the Hotstate compiler. This will be fixed in the next release.

Using the –o option will redirect the output to a file

hotstate –o emu.c program.c

Creates a file called emu.c that is ready for emulation.

hotstate –a –o analysis.txt program.c

The command above creates a file analysis.txt that will contain the output of the analysis described below.

If you use the compiler option –A (or –a) the compiler will print out an analysis of your program.

simple.c

```
bool LED0 = 0; /* state0 */
bool LED1 = 0; /* state1 */
bool LED2 = 1; /* state2 */

/* inputs */
bool  a0, a1, a2;

void
main()
{

/* main loop */
 while (1)
 {
  if(a0 == 0 && a1 == 1)
    LED0 = 1;
  else if((a1 == 0 || a2 == 1) & !a0)
    LED1 = 1;
  if(a0 == 1 && a2 == 0)
    LED2 = 1;
  if(a0 == 0 && a2 == 0 & !a2)
    LED0 = 0, LED1 = 0, LED2 = 0;
  if (a0 == a1) LED0 = 1;
 } /* end while */
}
```

```
State Machine Microcode derived from simple.c


                s s
                w w s     f
a               i i t t   o
d       v t     t t a i b r
d s     a i t c c t m r c
r t m j r m i h h e / a e
e a a a S S m s a C v n j s r
s t s d e e L e d a a c m u t
s e k r l l d l r p r h p b n
-------------------------------
0 4 7 0 0 x x x x 1 0 0 0 0 0   main (){
1 0 0 e 0 x x x x 0 0 1 0 0 0   while (1) {
2 0 0 5 1 x x x x 0 0 1 0 0 0   if ((a0 == 0) && (a1 == 1))
3 1 1 0 0 x x x x 1 0 0 0 0 0   LED0=1;
4 0 0 7 0 x x x x 0 0 0 1 0 0   else
5 0 0 7 2 x x x x 0 0 1 0 0 0   if ((a1 == 0) || (a2 == 1) & !(a0))
6 2 2 0 0 x x x x 1 0 0 0 0 0   LED1=1;
7 0 0 9 3 x x x x 0 0 1 0 0 0   if ((a0 == 1) && (a2 == 0))
8 4 4 0 0 x x x x 1 0 0 0 0 0   LED2=1;
9 0 0 b 4 x x x x 0 0 1 0 0 0   if ((a0 == 0) && (a2 == 0) & !(a2))
a 0 7 0 0 x x x x 1 0 0 0 0 0   LED0=0, LED1=0, LED2=0;
b 0 0 d 5 x x x x 0 0 1 0 0 0   if ((a0 == a1))
c 1 1 0 0 x x x x 1 0 0 0 0 0   LED0=1;
d 0 0 1 0 x x x x 0 0 0 1 0 0   }
e 0 0 e 0 x x x x 0 0 0 1 0 0   :exit

State assignments
state 0 is LED0
state 1 is LED1
state 2 is LED2

Variable assignments
var 0 is a0
var 1 is a1
var 2 is a2
```

Machine sizing

There are several options that control the size of the Hotstate machine. You can control the width of the counters and switch expressions. You can also set the stack.

hotstate –t <number> program.c

This option will set the width counters to <number>. The default is 32, which is quite large and creates 32-bit adders.

It might be the case in your code that you never have a “for” loop over 32. You would then set the counter width to 5.

hotstate –t 5 program.c

This will create 5-bit adders which are smaller and faster than 32-bit adders.

hotstate –w <number> program.c

This option sets the width of all switch expressions in the program to <number>. The default is 8. Switch expressions (switch (expression) {) are identified by declaring them type char this is 8-bits. The means the translation table has to be 256 words deep.

If you only have 10 cases and you can number them 1-10 you can compile with –w 4. This will make the input small and the table depth will go to 16. If you set the width to less than 8 this will show up in the testbench and templet but not the software emulation. If you set the width to greater than 8 the compiler will promote the 8-bit char to 32-bit int. In the testbench and templet you will see the number reflected in the port width.

hotstate –s <number> program.c

This option set the depth of the stack. The default is 4. If you don't have any functions calls (not including _user()) the complier will eliminate the stack. If you don't call functions from functions you only need the stack to be one deep (hotstate –s 1 program.c). If you call one function from another you need 2 deep if the called function does not call another function. You can have recursive functions and you can make the stack 100's deep but no program can infinitely recurse. A stack over flow is not monitored so be careful when calling functions from functions.

Software Debugging

Comma C is a subset of C. Each Comma C program can be run in any C debugger or for that matter compiled into a program to run on a CPU.

There is a special built-in function called _user(int n). You declare it as an extern void.

```
extern void _user();
```

The _user function takes a number for an argument. This number is to be used in a switch statement. If you _user in your program and there is no _user.c file the compiler will make one for you and put in the switch as well as all variables and states in your program. The Hotstate compiler will ignore the _user function when compiling to the state machine.

If you don't have a makefile the compiler will make one for you. The compiler will also make a skeleton _user.c file that you use to drive the input singles of your program. If you already have a makefile and you have not edited it just delete it and the compiler will make a new one. Or you can copy your makefile to makefile1, for example.

In the automatically generated makefile you can see emu.

```
make emu
```

This will try and call out the gdb gnu debugger.

Hardware Debugging

When you use either the –T or –M or –S option the compiler output is a Verilog testbench that uses the Hotstate machine open-source IP. The compiler will check to see if there is a file called user.v. If there is not it will generate user.v and place a simple Verilog module that will let you manipulate the input variables and monitor the output states of the state machine. Using 2 of these options together the last option has priority. –T –M = -M while –M -T = -T.

The Hotstate machine is written in SystemVerilog. There are 8 files that you need to have:

hotstate.sv
microcode.sv
control.sv
next_address.sv
variable.sv
timer.sv
switch.sv
stack.sv

The code parameters drive the size and speed of the state machine.

Other state machine signals

The state machine has several signals used at the hardware level.

clk: The single clock for the system.

rst: The reset will drive the internal address to zero.

hlt: The halt signal. This wraps the address around next address mux.

interrupt: This signal will route the interrupt_address to the address bus and jump to that address. The current address will be pushed on the stack and the machine will jump to interrupt_address. You can jump to any address in the address space, but do that at your own risk. You would want to jump into a function that has a return statement. If you call a function interrupt() the compiler will use that address in the verilog template. When the interrupt signal fires it is on the raising edge.

interrupt_address: see above.

ready: When all the files are loaded this signal will go high and stay there unless a reset is issued.

Running in hardware

Optimizing tips

state0 = 1, state1 = 0; // takes one clock

The switch and run at one symbol a clock if you only have one line of code in the switch statement if you use continue at the end of the case statement

case 5: {this=1, that = 0, something_else = 1; continue;} // takes one clock

state0 = 1; state1 = 0; // takes two clocks

Often you will want to toggle a bit for one clock

state0 = 1;

state0 = 0;

This will execute in one clock and set state0 = 1

state0 = 0, state0 = 1; // same as state0=1; behaves the same in gdb

Know problems

Every function must have a return statement only, “return;” is allowed

Every case and default statement must have braces “case 1:{state1 = 1; continue;}”

There is currently no software to load the Hotstate machine from a processor at runtime.

The command hotstate -H will output a .hot file. This has all the data to load the machine in an ASCII array suitable for compiling into programs. The testbench shows how to load the state machine in Verilog. You need to set the rst to 1 and then load the arrays, then release rst.

The Hotstate state machine