

THE REFERENCE MANUAL FOR THE HOTSTATE MACHINE

A runtime loadable microcoded algorithmic state
machine (RLMASM)

Revision 1.1.0

Copyright © Hotwright Inc. 2022-2023

2/25/2023

Contents

Introduction.....	1
Software Architecture	3
Hardware Architecture	4
Basic Types.....	5
State Outputs	5
Variable inputs	5
Case Expressions.....	6
For Loop Variables	6
Microcode bits	7
Microcode structure and short description of bit fields	7
Detailed bit field description	8
Compiler options.....	9
Hardware Object Technology (HOT)	11
Machine sizing.....	11
Software Debugging.....	12
Hardware Debugging.....	13
Other state machine signals	13
Runtime loading and the Hotstate machine.....	15
Example design	17

Figures and Tables

Figure 1: Algorithm data and control planes.....	1
Figure 2: Supported C control statements	3
Figure 3: Hotstate block diagram	4
Figure 4: eigenLUT table for 3 variables.....	5
Figure 5: Table of bit field descriptions.....	7
Figure 6: Parser control flow	17
Figure 7: Parser data flow	18
Figure 8: Xilinx hardware manager	20
Figure 9: ILA wave forms.....	20
Figure 10: Xilinx IP integrator layout.....	21
Figure 11: The Hotstate machine	21

Introduction

Every program can be divided into two building blocks: 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

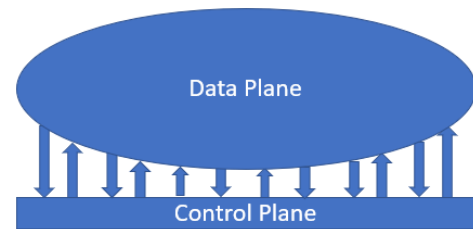


Figure 1: Algorithm data and control planes

The Hotstate RLMASM is an enhanced Moore style state machine, with some twists, that implements the control plane of an algorithm. The enhancements are a stack for function calls, a switch indirection table, a timer table and a CAM for evaluating inputs. One twist is that the possible output states are exponentially greater than any other state machine. (State Outputs) Another twist is that it is programmed in a subset of “C” instead of some GUI tool. All the C control statements and some limited use of functions are supported.

The motivation for the Hotstate machine is the observation that processor systems use an ALU, register file and a fixed hardware controller to divide the data and control planes into small pieces called “instructions.” The fixed hardware controller implements instruction decode of a fixed ISA and data orchestration.

The Hotstate machine performs the same function as the hardware controller of a processor for an arbitrary hardware architecture. Using one or more Hotstate machines in a hardware design enables programmable concurrency control.

The arbitrary data plane is created by the user. The user could have hundreds of different hardware components all controlled by one or more of the Hotstate machine RLMASMs. The behavior of the hardware can be changed by reloading the microcode at runtime. Changing the control plane microcode is thousands of times faster than partial reconfiguration and much easier to do.

The Hotstate machine and a handful of logic can replace softcore CPUs in an FPGA for many applications saving time, money, and power. The Hotstate machine delivers only the resources needed for the program. The machine is smaller and faster than any softcore CPU. The Hotstate machine saves area, which is valuable, and the hardware can run faster to get the job done in less time.

The Hotstate machine is strictly sequential in that there is only 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 to execute a program. This is programming at a lower level than HLS but a higher than HDL.

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 be loaded into counter/timers
- 5) a verilog template with parameters to build a state machine to exactly fit the code
- 6) a testbench that can used for unit test
- 7) a user.v file where user hardware test code goes
- 8) a _user.c file where software test code goes
- 9) a makefile that contains common commands
- 10) code capable of being debugged with gdb or any other C debugger

Software Architecture

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

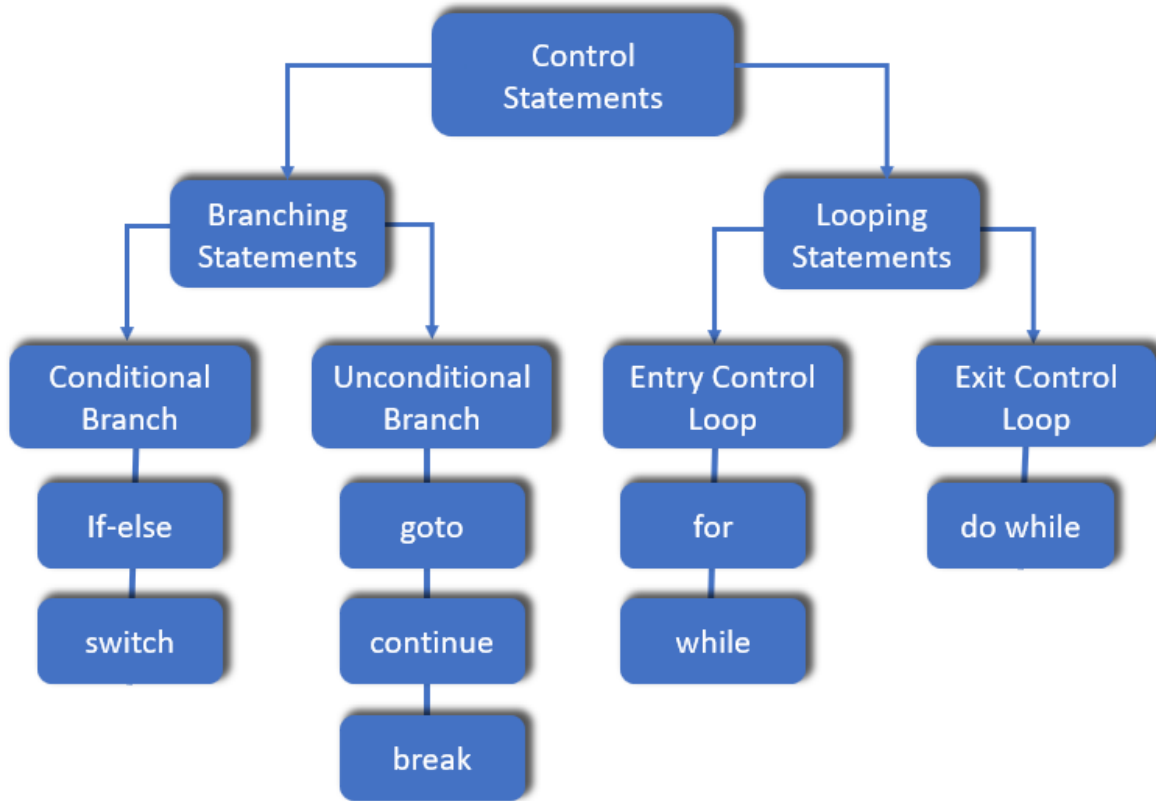


Figure 2: Supported C control statements

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 are supplied by the user. They could be connected to hardware status bits or data for the front-end CAM.

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 and create a stateful network of RLMASMs. In addition to C control statements there is a limited subroutine capability.

Since all variables are global, there is no need to pass parameters to functions except the built-in debugging function `_user(int)`. Functions can be called from other functions and recursive functions are supported. The state machine has a stack so it can be overflowed. There is an option to make the stack deeper. Later revisions will 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 compiler.

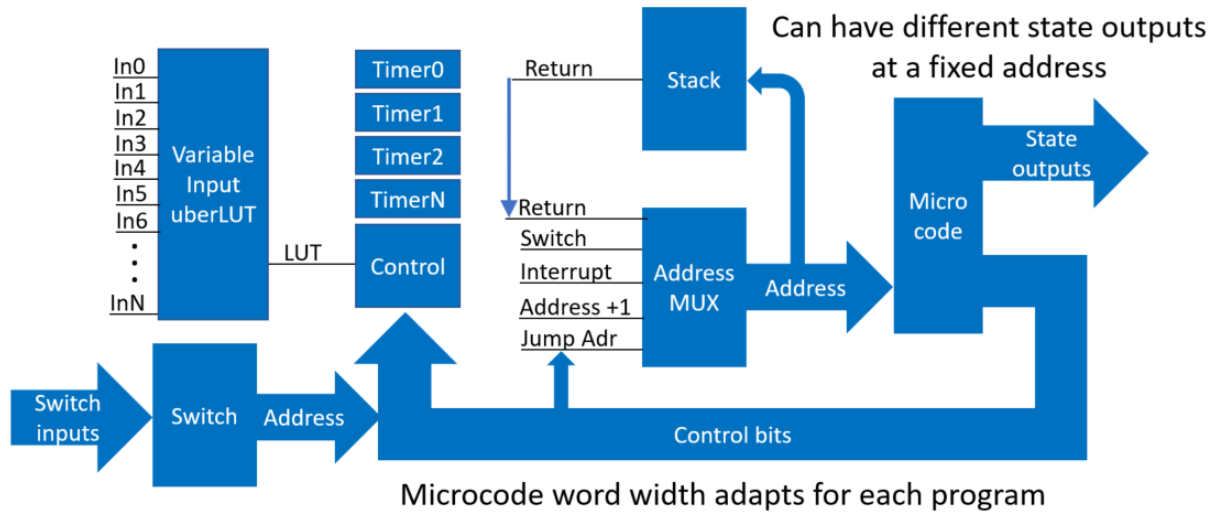


Figure 3: Hotstate block diagram

The Hotstate compiler extracts parameters from the code, passes those parameters to the System Verilog hardware, 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 switch translation tables. These memories may or may not be generated depending on which statements are used. If supported features are not used, “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 there are 256 lines of microcode, then only an 8-bit program counter is needed. A processor core has a register file that might have 32 32-bit registers. A processor has an ALU packed with every possible opcode. With the Hotstate machine, the user adds the right amount of hardware needed and lets the state machine control that hardware. This might mean hundreds of adders, multipliers, and registers, or a single mux and FIFO controls, all controlled via software.

Basic Types

There are 4 different types of variables 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 encoded in microcode memory.

State Outputs

To control state outputs simultaneously, 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 captured if appropriate. The mask bit and state bits are used to create the new state output.

$$\text{state}[i] = \text{new_state}[i] \text{ if } \text{mask}[i] = \text{true} \text{ else } \text{old_state}[i];$$

The number of possible 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. This means that the Hotstate can express more complexity than a standard state machine. In the traditional algorithmic state machine, 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 there are three 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

Outputs can be wrapped around to inputs allowing the state machine to create a persistent state. Each expression in a program is given an eigenLength LUT. 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 compiler 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. The -w option to reduce the size of this table.

If the number of bit variables is greater than 8 change the “char” to the appropriate C type to debug in software. The variable must be “char” to go through the Hotstate compiler. Use the -w option to tell the Hotstate machine if “char” is not equal to 8. For example: -w 12 or -w = 4

```
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.
```

Functions

```
void function (); // All functions must be declared “void”. All variables are global.  
Functions can call other functions or recurse. There are no checks for a stack overflow but the stack depth can be set with the -s <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 Hotstate machines for an algorithm. Each machine will run concurrently and can have output states from one machine as inputs to others.

Microcode bits

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_jump;  
    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_jump	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 the code. They are assigned state numbers in the order in which they appear in the 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_jump bit set this is the microcode address to jump to. If the timerLd bit is set this points to the address in the timer memory that gets loaded into the countdown counter. If switch_sel is set this field holds the address 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. For two input variables a0 and a1. The eigenLength will be four. EigenLUT a0 = 0101 and a1 = 0011.

```
void main () {  
while (1) //varSel is 0  
if (a0 ^ a1) state = 0; // varSel is 1  
}
```

The uberLUT for this program is 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: This is a variable sized field whose width is equal to the number of “for” loops used in a program. Each timer gets its own bit. If there are n timers, then this field is n bits wide. Only one timerSel bit is active at any one address. The lsb is the first variable declared as “int”. When a bit is 1, the timerSel field loads that number from the timer memory and decrements that number.

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 switchmem. This signal is an output and is used in the Verilog templet to select input via a mux. Width depends on the number of switch variables.

switch_adr: This is a single bit that selects the address in the switchmem 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 Hotstate machine will jump to the address in the jadr field. If the expression is true the Hotstate machine will go to the next address.

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

sub: This is used with the forced_jump 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 the 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 an existing 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 the compiler option -A (or -a) is used the compiler will print out an analysis of the program.

simple.c	hotstate -A simple.c
<pre> 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 */ } </pre>	<pre> State Machine Microcode derived from simple.c s s w w s f a i i t t o d s 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 </pre>

Hardware Object Technology (HOT)

Hardware Object Technology (HOT) is software that takes the data used to configure an FPGA (or in this case the Hotstate machine) and turns that into a format that can be compiled and used in a program. We are applying this philosophy to the Hotstate machine microcode data.

When the -H option is used the compiler will create a <program>.hot file.

This file will contain all the state machine memories and a function to load them and get them ready to be downloaded at runtime. That function is called <program>_init().

Machine Sizing

There are several options that control the size of the Hotstate machine. These options can control the width of the counters, switch expressions, and the stack depth.

```
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 that there is no “for” loop that goes over 32. To save area and increase speed set the counter width to five.

```
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. This means the translation table has to be 256 words deep.

If there were only 10 cases in a switch statement, the statements would be numbered 1-10 and compiled with -w 4. This will make the input small and the table depth will go down to 16. If the width is set to less than 8 this will show up in the testbench and templet but not the software emulation. If the width is set to greater than 8 the compiler will promote the 8-bit char to 32-bit int. In the testbench and templet the number will be reflected in the port width.

```
hotstate -s <number> program.c
```

This option sets the depth of the stack. The default is 4. If there are no function calls (excluding _user()) the compiler will eliminate the stack. If functions are not called from functions the stack can be set to be one deep (hotstate -s 1 program.c). If one function is called from another the stack must be at least two deep. Recursive functions are legal, and the stack can be 100's deep, but no program can recurse infinitely. A stack overflow 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 on 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)`. It is declared 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 `_user()` is used in the program and there is no `_user.c` file, the compiler will make one and put in the switch as well as all variables and states in the program. The Hotstate machine compiler will ignore the `_user()` function when compiling to the Hotstate machine.

If there is no makefile in the directory the compiler will create one. The compiler will also create a skeleton `_user.c` file, if `_user(int)` is used, that is used to drive the input signals of the program. If there is already a makefile and it has not been edited, just delete it and the compiler will make a new one. Or copy the makefile to `makefile1`, for example.

In the automatically generated makefile there is an entry for `emu`.

```
make emu
```

This will try and call out the `gdb` gnu debugger.

Hardware Debugging

When using either the `-T` or `-M` or `-S` option, the compiler will create a Verilog simulation testbench that uses the Hotstate machine source code IP. The compiler will check to see if there is a file called `user.v`. If there is not, the compiler will generate a `user.v` file and with a simple Verilog module in it. The `user.v` file is where the input variables are manipulated to test the Hotstate machine. Monitor the output states for expected behavior. When using 2 or more 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 are needed:

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

Parameters extracted from the code drive the size and functionality of the Hotstate machine.

The Kria parserRL example (`hotstate/examples/parserRL`) is a working example of how to debug the Hotstate machine in hardware. Each FPGA vendor has a different set of tools. Our first examples support Xilinx directly however the hardware is portable. Contact us for consulting services to port these examples to different vendors' tool sets.

Other Hotstate machine signals

The Hotstate 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`. The Hotstate machine can jump to any address in the address space but there are some considerations to minimize risk. Jumping into a function that has a return statement is safe as this will pop the stack and return the program counter to the address the Hotstate machine was at when the interrupt activated. If a function is called `interrupt()` the compiler will use that address in the verilog template. The interrupt fires on the raising edge of the signal and will jump to the address on the `interrupt_address` port in one clock cycle.

`interrupt_address`: see above.

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

Optimizing tips

`state0 = 1, state1 = 0; // assignment statement takes one clock`

If there is only one line of assignments in the case statement, and there is a continue at the end of the case statement, the switch will run at one symbol per clock.

`case 5: {state0=1, state1 = 0, state2 = 1; continue;} // takes one clock`

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

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`

If the Hotstate machine is too large consider creating two or more Hotstate machines instead of one large machine.

Runtime loading and the Hotstate machine.

Runtime loading is accomplished through the API that loads the machine as well as resets and halts the machine.

The header file `hot.h` contains the hot structure and function prototypes.

```
#ifndef _HOT
#define _HOT

typedef struct hot {
    uint32_t smdata_length;
    uint32_t smdata_width;
    uint32_t uberLUT_length;
    uint32_t uberLUT_width;
    uint32_t timermem_length;
    uint32_t timermem_width;
    uint32_t switchmem_length;
    uint32_t switchmem_width;
    uint64_t *smdata;
    uint64_t *uberLUTdata;
    uint64_t *timermemdata;
    uint64_t *switchmemdata;
} hot_t;

extern void loadsmdata(volatile uint64_t *base_address, hot_t *hot);
extern void loaduberLUT(volatile uint64_t *base_address, hot_t *hot);
extern void loadtimermem(volatile uint64_t *base_address, hot_t *hot);
extern void loadswitchmem(volatile uint64_t *base_address, hot_t *hot);

extern void hot_start (volatile uint64_t *);
extern void hot_stop (volatile uint64_t *);
extern void hot_reset (volatile uint64_t *);
#endif
```

These functions are in `hotstate/lib/hotlib.c`. The functions run on the hardened processor of the SoC and talk to the programmable logic.

Here is a typical hot file. It contains the data needed to load the Hotstate machine. To program the Hotstate machine include the hot file (`#include "simple.hot"`) and call out the initialization function `simple_init()`; Multiple hot files can be included in a program. If Hotstate machine is reload the state machine the parameters for length and width must be the same for each hot file. If they don't match take the hot file with the largest parameters and match the smaller ones to the larger one. This can be done by padding the data fields with zeros and adjusting the parameters to match.

```
// Copyright (c) 2021-2023 Hotwright Inc.  
// machine generated do not edit
```

```
// derived from simple.c  
// Created Mon Feb 27 01:28:16 2023
```

```
#include <hot.h>
```

```
hot_t simple;  
void simple_init () {  
    simple.smdata_length      = 15;  
    simple.smdata_width      = 23;  
    simple.uberLUT_length    = 48;  
    simple.uberLUT_width     = 1;  
    simple.timermem_length   = 0;  
    simple.timermem_width    = 0;  
    simple.switchmem_length  = 0;  
    simple.switchmem_width   = 0;  
  
    uint64_t smdata[15] = {  
        0x02003c,0x080380,0x080540,0x020008,  
        0x1001c0,0x0809c0,0x020012,0x080e40,  
        0x020024,0x0812c0,0x020038,0x081740,  
        0x020009,0x100040,0x100380};  
  
    simple.smdata = (uint64_t *) calloc(15, sizeof (uint64_t));  
    memcpy(simple.smdata, smdata, sizeof(smdata));  
  
    uint64_t uberLUTdata[1] = {  
        0x000055050a5144ff};  
  
    simple.uberLUTdata = (uint64_t *) calloc(1, sizeof(uint64_t));  
    memcpy(simple.uberLUTdata, uberLUTdata, sizeof(uberLUTdata));  
  
};
```

Here is how it would be used in a program.

```
simple_init();  
simple1_init();  
  
hot_reset(mem);  
loadsmdata (mem1,&simple);  
loaduberLUT(mem2,&simple);  
  
hot_reset(mem);  
loadsmdata (mem1,&simple1);  
loaduberLUT(mem2,&simple1);
```

Example design

The example design is a simple ascii parser. Given a set of ascii characters, filter out words.

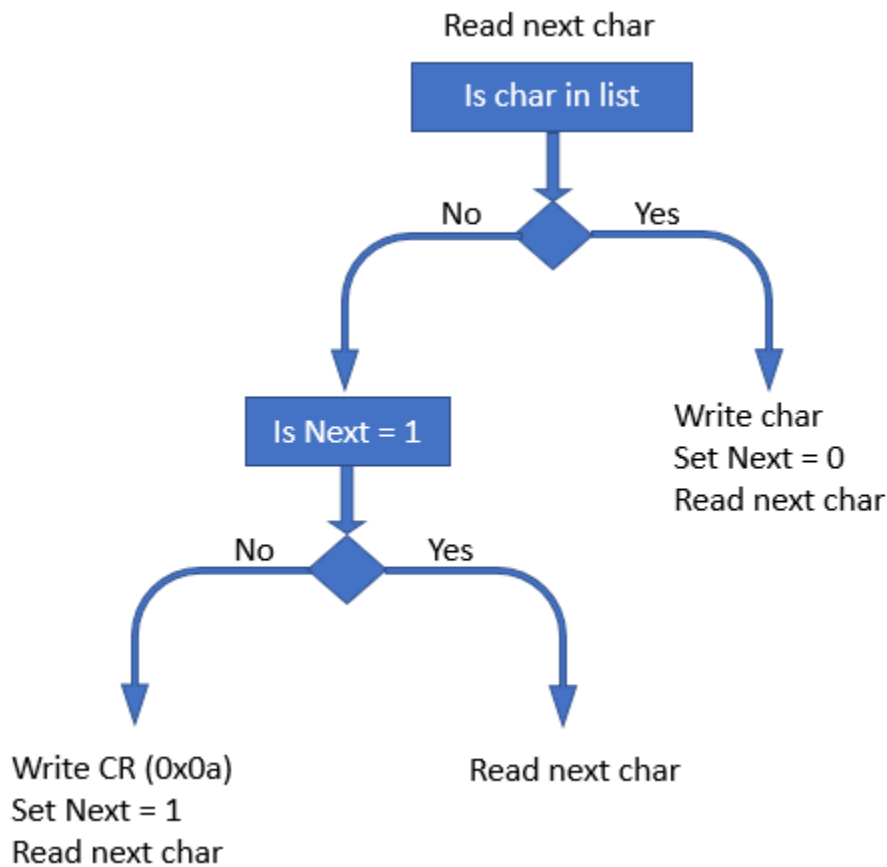


Figure 6: Parser control flow

The code to do this

```
//Copyright © Hotwright Inc. 2022-2023
#include "ascii_defs_in.h"

// for debugging in software
extern void _user();

#define bool _Bool

bool pop= 0;
bool push = 0;
bool cr_data = 0;

bool din0;
bool din1;
bool din2;
bool din3;
bool din4;
bool din5;
bool din6;
bool din7;
bool cr_wrap_around;
bool valid;

void main() {
    _user(0); // set input
    //get rid of non-letters in front
    while (!(lower_case | upper_case)) {
        if (valid == 1) {
            pop = 1;
            pop = 0;
        }
        else pop = 0, push = 0, cr_data = 0;
    }
}
```

```
// do the rest
while(1) {
    if (valid == 1) {
        _user(1); // set input
        if ((lower_case | upper_case)) {
            push = 1, pop = 1, cr_data = 0;
            push = 0, pop = 0;
        }
        else if (cr_wrap_around == 0) {
            cr_data = 1, push = 1;
            push = 0;
        }
        else {
            //pop = 1, push = 0, cr_data = 0;
            pop = 1, push = 0;
            pop = 0;
        }
    }
    else pop = 0, push = 0, cr_data = 0;
}
```

The Hotstate machine compiler uses the C preprocessor to read header files. One such file is `ascii_defs_in.h`

```
#define _A ((din0 == 1) & (din1 == 0) & (din2 == 0) & (din3 == 0) &\n          (din4 == 0) & (din5 == 0) & (din6 == 1) & (din7 == 0))
```

Note that the variables names in the include file match the input variables names in the program.

```
#define lower_case (_a|_b|_c|_d|_e|_f|_g|_h|_i|_j|_k|_l|_m|_n|_o|_p|_q|_r|_s|_t|_u|_v|_w|_x|_y|_z)
```

The `parserRL` example will check if an `ascii` character is in the set every clock cycle. It does not matter how many characters in the set the Hotstate machine will check against every character in the set in one clock cycle.

There are two programs in the `parserRL` example. `parserRL.c` and `parserRL1.c`. When these two programs compiled with the `-H` option a

In the `parserRL` example there is a directory called `parserRL-app`. This contains the code to talk to the Kria board. It runs on the ARM and loads the microcode into the Hotstate machine. The microcode is in `parserRL.hot` and `parserRL1.hot`.

The main data path of the hardware design consists of two FIFOs. The input FIFO takes a 64-bit word from the Arm and presents an 8-bit word to the parser. If the character is part of the set the parser pushes the data into the output FIFO and pops more data from the input FIFO. If not the parser pops the input FIFO. The FIFOs have registers that hold how much data is in the FIFOs and this data is put in a 64-bit status register used by the software to manage reads and writes to the hardware.

The application creates two software threads. One that writes to the hardware and one that reads from the hardware.

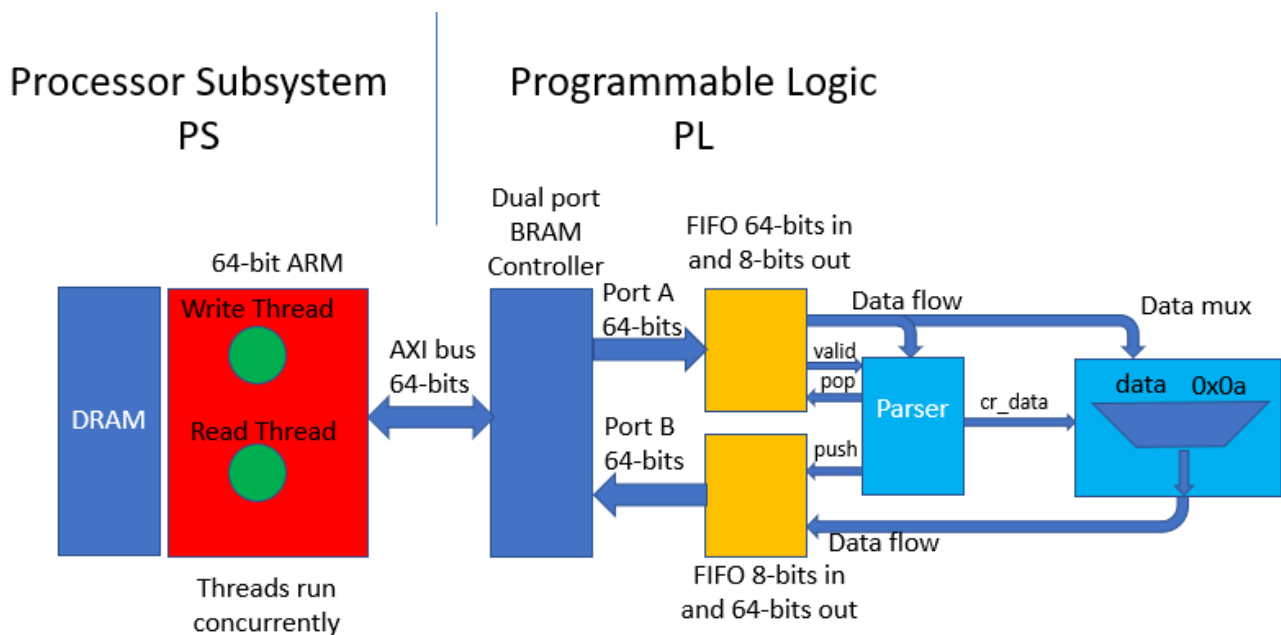


Figure 7: Parser data flow

```

// Hotwright Inc. All rights reserved (c) 2021-2023
// Read and write to kria board via threads
// hotstate machine structure for runtime loading.
#ifndef _KRIA
#define _KRIA

extern void *read_kria(void *);
extern void *write_kria(void *);

typedef struct READ_PARAMS {
    char      *data_out;
    uint64_t volatile *baseaddress;
    uint64_t volatile *statusbase;
    uint64_t words_read;
    uint64_t flush_char;
    uint64_t *write_done;
} ReadParams_t;

typedef struct WRITE_PARAMS {
    char      *data_in;
    uint64_t volatile *baseaddress;
    uint64_t volatile *statusbase;
    uint64_t size_in_bytes;
    uint64_t fifo_size;
    uint64_t *write_done;
} WriteParams_t;

#endif

ReadParams_t *ReadParams = (ReadParams_t *) aligned_alloc(512,
sizeof(ReadParams_t));
ReadParams->data_out = (char *) test_data;
ReadParams->baseaddress = mem0;
ReadParams->statusbase = mem1;
ReadParams->words_read = 0;
ReadParams->write_done = &write_done;
ReadParams->flush_char = 'A';
WriteParams_t *WriteParams = (WriteParams_t *) aligned_alloc(512,
sizeof(WriteParams_t));
WriteParams->data_in = data;
WriteParams->baseaddress = mem0;
WriteParams->statusbase = mem1;
WriteParams->size_in_bytes = size;
WriteParams->fifo_size = 0x200;
WriteParams->write_done = &write_done;

pthread_t ReadThread, WriteThread;
pthread_create(&WriteThread, NULL, write_kria, (void *) WriteParams);
pthread_create(&ReadThread, NULL, read_kria, (void *) ReadParams);
pthread_join(ReadThread, NULL);
pthread_join(WriteThread, NULL);

```

The functions `write_kria` and `read_kria` are in `hotstate/lib`. This library needs to be compiled on the Arm. Gcc may have to be installed.

The parserRL design is at `hotstate/examples/parserRL/parserRL`. Once there click on `parserRL.xpr`. This is a Xilinx/AMD design, and the user must have the proper licenses to run it.

Create a bitstream and download it to the Kria board from the hardware manager. There are five integrated logic analyzers (ILAs) in the design. Open the hardware manager and download the bitstream to device. Run all the triggers on each of the ILAs. Run the program, parserRL-app, and the ILAs will be triggered and collect data as shown below in figure 10.

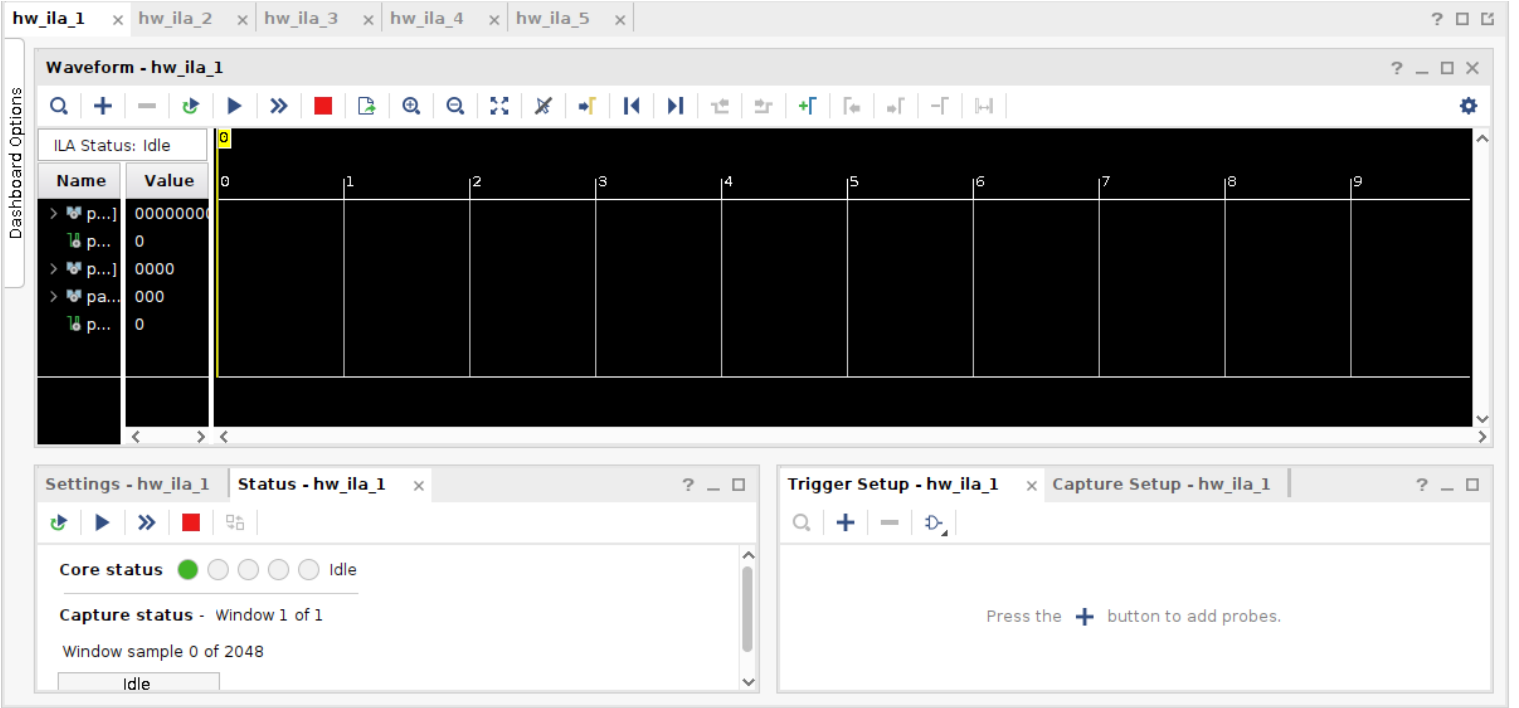


Figure 8: Xilinx hardware manager

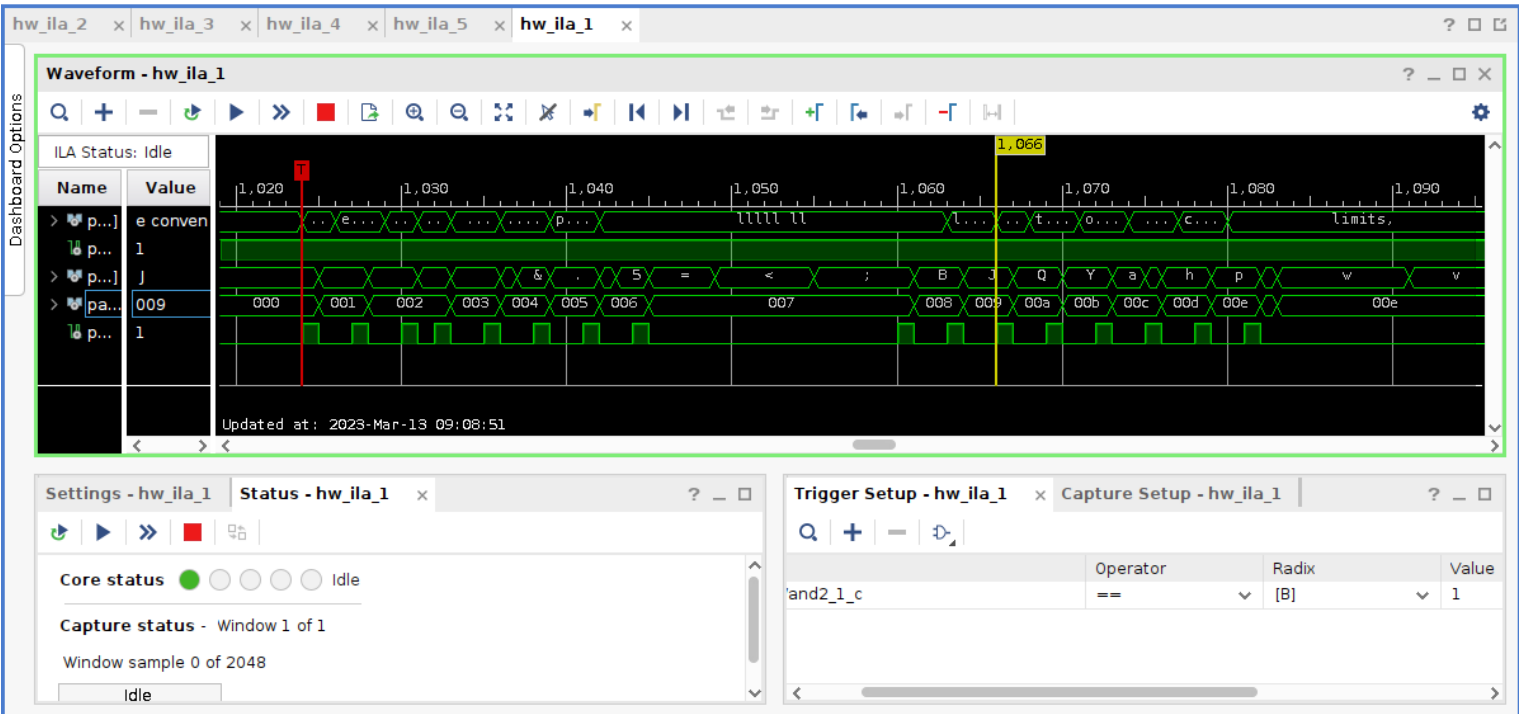


Figure 9: ILA wave forms

Know problems.

There is only one form of “for” loop supported for(start ;stop ;i++) It’s really a count down counter in hardware.

The diagram illustrates the control unit of the RISC-V processor, showing the internal structure and the flow of control signals and data. The control unit is composed of several blocks: Microcode, Variable, Branch, Control, Stack, and Next_addr. The inputs and outputs are labeled on the left and right sides of the diagram. The control unit is designed to manage the execution of instructions, including the fetching, decoding, and execution phases. The diagram shows the flow of control signals and data between these blocks, with various inputs and outputs labeled on the left and right sides of the diagram.

21