

OPEN VERILOG
INTERNATIONAL
USER GROUP MEETING

Open Verilog
International



PROCEEDINGS

MARCH 24-25, 1992

Synthesis and Verification of a GaAs Microprocessor from a Verilog Hardware Description

R. Brown, A. Chandna, T. Hoy, T. Huff,
R. Lomax, T. Mudge, D. Nagle,
K. Sakallah, R. Uhlig, and M. Upton

Advanced Computer Architecture Laboratory
Solid State Electronics Laboratory
The University of Michigan
Ann Arbor, Michigan

K. Olukotun

CIS, Stanford University
Stanford, California

D. Johnson

Cascade Design Automation Corporation
Bellevue, Washington

Abstract

The University of Michigan Gallium Arsenide MIPS Project is using Verilog in the design of a 250 MHz MIPS architecture microprocessor. The design system is based on a single Verilog model which is used for simulation, synthesis, and hardware verification. The model is composed of a mixture of Register Transfer Level (RTL) and behavioral descriptions. Datapaths are represented by RTL structural components, while the control logic has behavioral descriptions.

To simplify verification and test development, a number of operating system functions have been implemented using the Verilog PLI (Programming Language Interface). These functions allow the model to load and execute programs compiled for the DECstation 5000. To ensure the model's functional correctness, a verification tool compares simulation results against the execution of an physical MIPS processor. Any inconsistencies are flagged as errors. Once the model is deemed functionally correct, it is synthesized into a logic level implementation. Datapath logic, described at the register transfer level, is directly mapped into a netlist for automatic placement and routing. The control logic is translated to the Finesse logic synthesis language. The Finesse compiler then synthesizes each control block into a netlist which is passed to the physical design tools from Cascade Design Automation (CDA) for final layout.

The combination of front-end verification and back-end synthesis results in a very short design time. Our first chip, Aurora I, was completed by a team of 5 graduate students in about 5 months, including tool development and library cell layout.

1 Introduction

The primary objective of the GaAs MIPS group is to build an integrated GaAs microprocessor system consisting of a CPU, Memory Management Unit (MMU), Floating Point Unit, and two-level cache on a Multi-Chip Module[1]. However, given the difficulties involved in GaAs design, it would have been very time consuming to attempt the design without first establishing an automated CAD tool environ-

ment. Therefore, we have an integrated suite of commercial and locally developed tools that will take a design from the behavioral domain into the physical domain using the process flow outlined in Figure 1.

The CAD system is divided into two major areas: hardware description and physical implementation. At the top of the system lies the Verilog model of the architecture. The model uses structural and behavioral modules to describe the datapath and control logic, respectively. The other end of the system, corresponding to actual lay out, is centered on the Cascade physical design tools [2].

Connecting the two CAD domains is a translator developed at the University of Michigan. This translator, *VeriChip*, takes the top-level Verilog description and separates it into two components: 1) control blocks and 2) a datapath netlist. The control blocks and a library of standard cells are synthesized into a gate-level netlist using the Cascade state machine synthesis program, Finesse [3]. Then, the datapath cells corresponding to the Verilog structural blocks are combined with the control netlist and datapath standard cell library to create the core layout.

Together with the design system outlined above, we have developed a sophisticated Verilog environment which provides many of the operating system functions on the host machine. These functions include program loading, memory management and unimplemented instruction simulation. We have also implemented a verification system to compare the internal state of the Verilog model with the internal state of the MIPS workstation running the same executable code. The entire CAD system allows us to use a single model for the design, simulation, testing and synthesis of our GaAs processor.

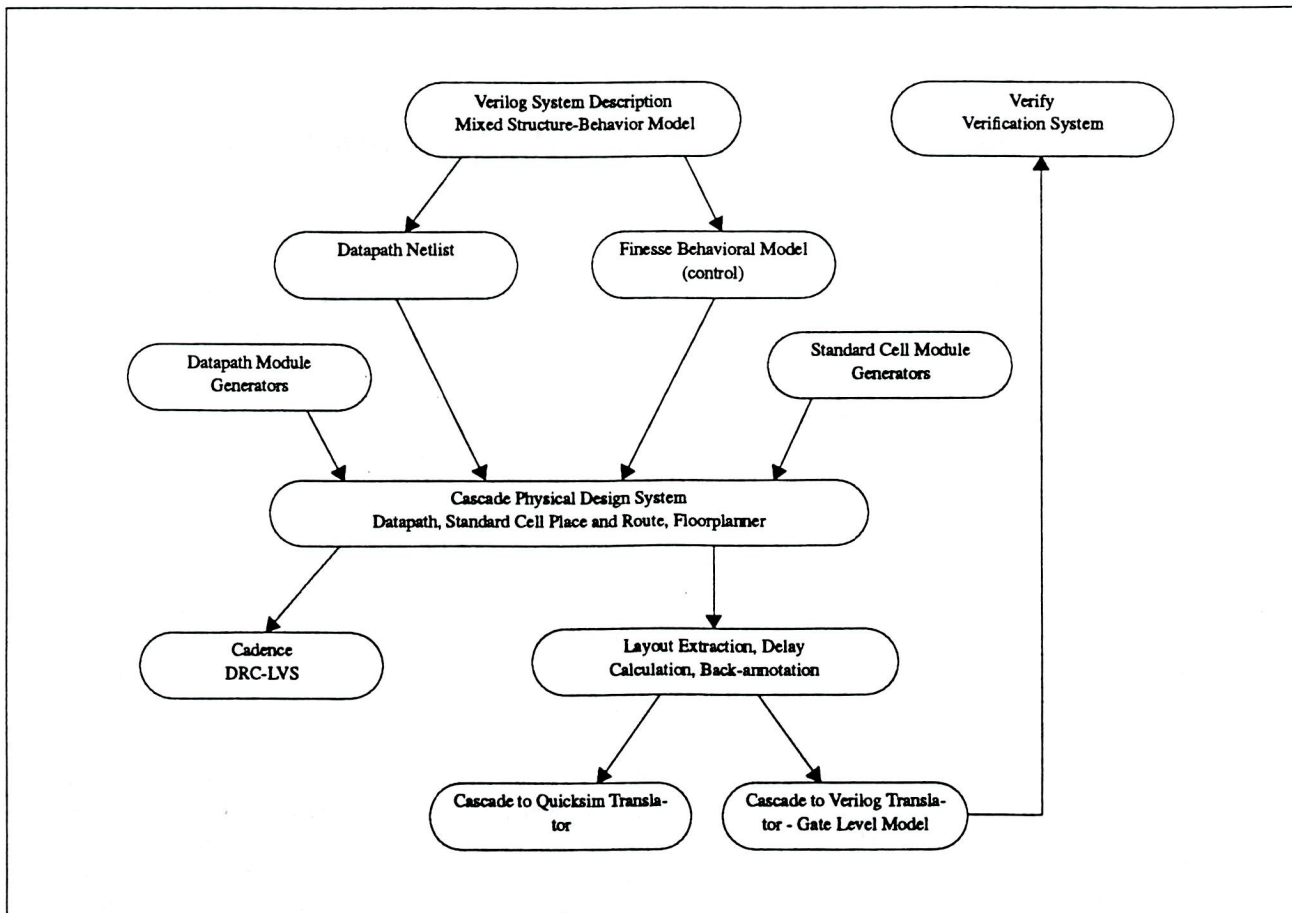


Figure 1: The CAD environment

2 Design Translation and Synthesis

The key to using Verilog as our design base is the ability to automatically translate the Verilog model into a form which is suitable for logic and layout synthesis by the Cascade tools. Therefore, we have developed a translator, VeriChip, that converts the Verilog model into Finesse input and Cascade netlist descriptions. This translation provides us with a single path between the high level Verilog description and the Cascade tools used to layout the chip.

Most of the translation between Verilog and Finesse involves a one-to-one conversion. However, the Verilog HDL is much richer than the Finesse logic synthesis language. Therefore, we have developed a modified form of the Verilog language, called *ChipVerilog*¹, which does not permit high level operators like +, *, or /. Instead, the model

1. Cascade Design Automation is incorporating support for Verilog into ChipCrafter 3.0.

must either explicitly instantiate a multiplier or divider that exists in the component library, or build a Verilog structural model from simpler, existing components.

ChipVerilog also incorporates several language extensions not found in Verilog. These extensions allow the translator to differentiate between four different module classes as outlined below:

- **PRIMITIVE:** gates.
- **FINESSE:** boolean equations, If-then-else statements, case statements, etc.
- **GENERATE:** variable bit-width components, especially datapath components.
- **HIERARCHY:** networks of gates, Finesse modules, Generate modules and other hierarchy modules.

The extensions used by VeriChip are identified by the “%” symbol. Consider the following example.

```

/*
Example of a primitive library module. The
only thing the translator needs is the I/O
definition. The simulation behavior is sepa-
rated and enclosed in translator comments.
*/
// 2 to 1 multiplexor library cell
module dpstwoto1nor (SEL, IN0, IN1, OUT);
//% class: PRIMITIVE
input SEL; input IN0,IN1; output OUT;
//%/* Verilog sim.; ignored by VeriChip
    wire SELBAR;
    wire t0,t1;
    not n1 (SELBAR,SEL);
    and a1 (t0,SEL,IN1);
    and a2 (t1,SELBAR,IN0);
    or o1 (OUT,t0,t1);
//%*/
endmodule

```

Example 1: Primitive class - a 2 to 1 MUX

To describe the module’s class, the token class:**PRIMITIVE** has been included in a special comment, “//%”. Because “//” denotes a single-line comment, the Verilog parser will ignore this entire line. VeriChip, however will look one character further and see the “%” symbol. This symbol tells VeriChip to cancel the “//” comment and continue parsing the line. Also, because this is a primitive module, VeriChip will not translate the Verilog description. Instead, it will instantiate a 2 to 1 multiplexor from our GaAs library into the netlist it creates. This is achieved by using the tokens “//%/*” and “//%*/”. To let VeriChip know the beginning and the end of the section that must not be translated. Because VeriChip ignores the Verilog description, the library designer much ensure identical functionality

The only ChipVerilog extension to **FINESSE** modules is their class specification. The rest of the module is translated into the Finesse language. Example 2 illustrates a **FINESSE** module.

```

/*
Example of behavioral descriptions, this mod-
ule performs a 3 to 8 decode, unless the
'allones' line is set, in which case its out-
put is set to all ones.
*/
// 3 to 8 decoder behavioral description
module decode( in, allones, out);

//% class : FINESSE

input [2:0] in;
input allones;
output [7:0] out; reg [7:0] out;

always @ (in or allones)
begin
    casex({allones,in})
        {1'b1,3'bxxx}: out = 8'b11111111;
        {1'b0,3'b000}: out = 8'b00000001;
        {1'b0,3'b001}: out = 8'b00000010;
        {1'b0,3'b010}: out = 8'b00000100;
        {1'b0,3'b011}: out = 8'b00001000;
        {1'b0,3'b100}: out = 8'b00010000;
        {1'b0,3'b101}: out = 8'b00100000;
        {1'b0,3'b110}: out = 8'b01000000;
        {1'b0,3'b111}: out = 8'b10000000;
        default: out = 8'h00;
    endcase
end
endmodule

```

Example 2: Finesse class - a 3 to 8 decoder

Because Verilog does not allow dynamic instantiation of modules, it is impossible to use Verilog to describe bit slice components. However we did not want to create a component library that contained separate descriptions for every possible module size. Instead, we embedded two descriptions into the **GENERATE** modules: the Verilog model, and a structural model described in the Verilog comments. This structural description is ignored by Verilog, but VeriChip recognizes the “%” symbol and translates the structural description.

```

/*
Example of a GENERATE module. This code will
create a variable width multiplexor. The
first section is used in simulation and is
ignored by the translator. The expand section
defines the interconnect of the primitive
library elements needed to generate the
netlist for the Cascade tools.
*/
// 2 to 1 mux with encoded select line
module mux2 (SEL, IO, I1, O);
//% class : GENERATE

parameter width = 32;

input SEL;
input [width-1:0] IO,I1;
output [width-1:0] O; reg [width-1:0] O;

integer i;

//%/* Verilog simulation; ignored by VeriChip
always @(SEL or I1 or IO)
begin
if( SEL == 1'b1) O=I1;
else O=IO;
end

//%*/

//% // VeriChip dynamic instantiation;
expand
for( i=0; i<width; i=i+1)
begin
%^ dpstwotolnor #(i) mux<%i%>
(.SEL(SEL), .INO( IO[i]), .IN1(I1[i]));
%^ assign O[i] = mux<%i%>.OUT;
end
%*/

endmodule

```

Example 3: Generate Class - a 2 to 1 MUX

When the translator encounters the **GENERATE** module, it ignores the Verilog description. Instead, it parses the structural description and creates a netlist with the correct number of bit slice components. Again, the module designer must ensure the Verilog model and structural description are equivalent.

After the Verilog model is translated into a Cascade netlist and Finesse input, Finesse performs logic synthesis using our GaAs library modules. Then, control logic's gate level netlist is merged with the datapath netlist generated by VeriChip. The complete description of control and datapath

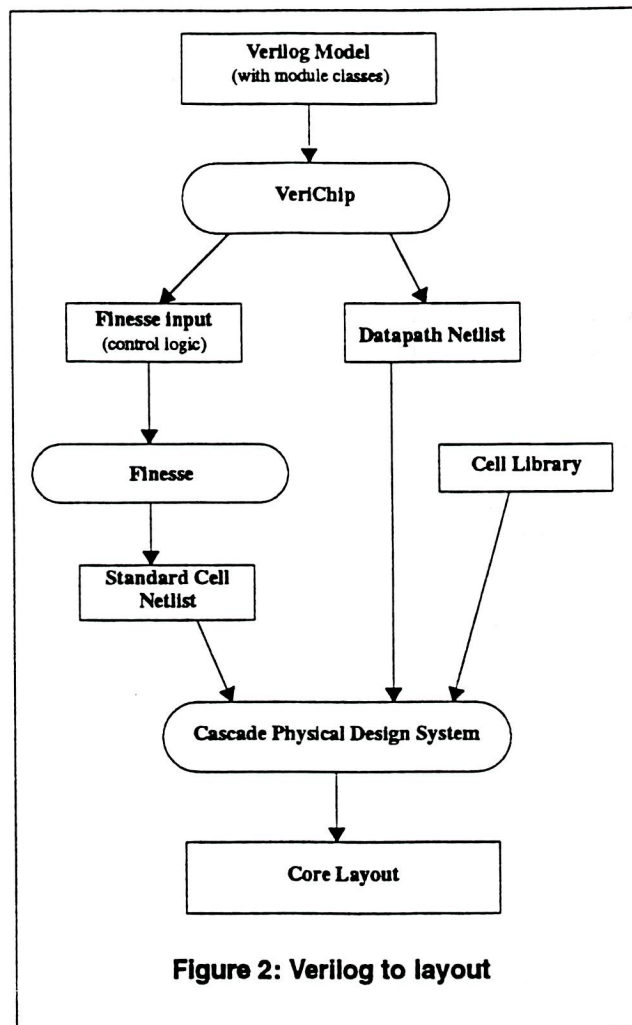


Figure 2: Verilog to layout

is then fed into the Cascade Physical Design System along with our standard cells. Finally, The Cascade tools produce the core layout. This process is depicted in Figure 2.

3 Emulation of an Operating System Environment

Because the Verilog model is used for both architectural development and processor design, we created an environment in which the model could load and run executable (binary) files compiled using the DECstation 5000 compiler. This allows us to use real programs to study the impact of various architectural changes and to debug the models. However, since the Verilog model currently only describes the processor, it is unable to support the memory management and system level functions required to load binaries and manage their run time memory requirements. Therefore, we have used the Verilog Programming Language Interface

Task Name	Description
<code>\$seed_mm(fileName)</code>	Seed a model of main memory with the executable image contained in <code>fileName</code> for future execution
<code>\$load_mm(address)</code>	Return the value stored at virtual memory location pointed to by <code>address</code>
<code>\$store_mm(address, datum)</code>	Store datum to virtual memory location pointed to by <code>address</code>

Table 1: Operating System Services (OSS tasks)

LD) to provide the operating system functions required to load and execute a program.

1 The PLI Operating System Services Interface

The minimal services required to execute a compiled program include the capability to load an executable image and run time memory management support. However, the processor model was incapable of providing these services because the first version of our processor (Aurora I) did not include any system level hardware models, (i.e. MMU, system bus, disk drives). Furthermore, we had to cope with compiled test programs that contained unimplemented instructions because Aurora I only implements a subset of the MIPS R3000 ISA (Instruction Set Architecture) [4].

To provide these missing functions, Verilog models of the I/O devices and memory management unit could have been built. Then, a “mini-operating system” that provides the minimal services and runs on these models could have been written. However, this approach would have required an unnecessary duplication of services that already exist in the simulator’s host operating system.

We adopted a different strategy. Using the Verilog PLI (Programming Language Interface), we wrote a series of routines that allow the processor model to directly access the host operating system services. We refer to this code as the Operating System Services (OSS) interface. It is composed of just three PLI tasks summarized in Table 1.

1.1 Loading Executable (Binary) Files

The first OSS interface function is `$seed_mm()`. A call to this task invokes several C routines through the PLI which perform the following sequence of steps:

The COFF file, `fileName`, is read to extract information specifying the start and length of the text, data and bss sections of the executable file. These operations are

performed using standard UNIX file I/O system calls provided by the host simulation machine.

- The appropriate amount of memory for each of these segments is allocated and then initialized with data contained in the executable image file. The memory is allocated with the standard C `malloc()` function supplied by the host operating system.
- Initial working space for the program stack and heap is allocated.
- Memory is allocated and filled with reset handling code which sets up the global pointer (gp) and stack pointer (sp) registers to point to the top of the program heap and stack. This reset code also contains a jump to the entry point of the program.

The reset code is placed at virtual memory location `0xbfc00000`, which is the R3000 reset vector address. The model begins executing code at this address after the processor’s reset signal is pulsed.

3.3 Virtual Memory

Once the processor model begins executing, it immediately requires access to memory for instructions and data. Since our Verilog model does not yet support an MMU (Memory Management Unit) to translate virtual addresses into physical addresses, the processor model expects to reference memory using full 32-bit virtual addresses. It is impractical to declare a 4 gigabyte Verilog array to represent this huge memory. Our solution implements a virtual memory space and gives the Verilog model access to this memory through `$load_mm()` and `$store_mm()`. These load and store tasks have direct access to the instructions and data from the executable file previously loaded by `$seed_mm()`.

Every invocation of `$load_mm()` causes the following steps to be performed through the PLI interface:

- The 32-bit value defined by **address** is compared against the text and data pages already in memory to see if the desired word is resident.
- If the desired word is in one of the resident pages, it is passed back to the Verilog model through the PLI.
- If the desired word is not resident, a new page of memory is allocated and an undefined value is passed back to the Verilog model. Stores with `$store_mm()` are handled in the same way as loads, except that an additional address check is performed to ensure that the store is not to a read-only segment (e.g. text segment). Writes to the kernel segment cause segmentation faults, as would happen in a real system. If no errors occur, the supplied **datum** is written to the specified **address**.

This technique of allocating memory only as needed is similar to "demand paging" in virtual memory systems. The main advantage is that it does not require a huge Verilog array to hold the entire contents of the memory space. It is also much simpler than real virtual memory page management because there is no need to explicitly page data out of memory when memory fills. This is because the memory allocation program runs in a UNIX process that has full access to a virtual address space itself. If this process begins to allocate too much memory, parts of its address space will be paged out automatically. In other words, the real complications of virtual memory are managed by the underlying operating system of the host machine. This is consistent with our strategy to "borrow" as many of the host OS services as possible.

3.4 Unimplemented Instruction Handling

The first version of our design only implements a subset of the R3000 ISA. This creates a problem when executing test programs that use unimplemented instructions

When hardware does not support all instructions in an ISA, a commonly used strategy is to generate an instruction exception so that the operating system can simulate execution of the unimplemented instruction with a trap handler. For example, a microprocessor-based system that does not have a hardware floating point unit can still define a set of floating point instructions and then implement them in software by trapping to the operating system.

We have extended this idea by implementing a a Verilog "monitoring" module that executes outside of the processor hierarchy. Its job is to probe into the processor model using hierarchical signal pathnames and watch for unimplemented instructions to appear in the instruction reg-

ister. Whenever an unimplemented instruction is detected, it function is simulated by a call to a C routine through the PLI. The routine is passed the required operands and the result are returned and injected into the model's datapath by using the Verilog **force** command.

This technique only allows programs to run on the Verilog *model* of the processor. The actual chip synthesized from the model will still be unable to perform any unimplemented instructions. However, this technique greatly extend the set of test programs that can be used to validate the model. A test program does not have to be ruled out simply because it contains a few unimplemented instructions.

Minor modifications to this scheme could implement more complex functions such as arbitrary system calls. For example, a **syscall** instruction could be detected, it arguments packaged up and sent through the PLI to a system call servicing routine. This routine would then make the real system call to the operating system of the simulation host machine. The results would be passed back to the Verilog model through the PLI. Such a scheme gives even the simplest processor model the ability to access any system services available through the standard UNIX system call mechanism. For example, test programs that require file I/O could easily be accommodated with this scheme.

4 Model verification

Because the Verilog model serves as the design basis for the GaAs MIPS processor, it is critical that the model accurately describe the processor's functionality. At first, verification was done by running opcodes through the model and manually validating the results. However, it soon became obvious that manual verification was consuming all of the developers' time. Therefore, we developed an automated verification system called *Verify*. *Verify* performs the verification by running DECstation executable binaries on both the model and the DECstation's R3000 processor. The model's internal state is compared against the R3000's state and any discrepancy is flagged as an error. This allows the designer to test the results of a model change and quickly locate errors. *Verify* will also run a complete suite of test programs and flag errors, freeing the designer to focus on design.

4.1 How *Verify* works

Two key factors helped in the design of *Verify*. First, our processor is a GaAs implementation of the standard R3000 architecture used by the DECstation. Second, our Verilog model can load and execute the same compiler generated binaries as the DECstation 5000. Therefore, we can compare

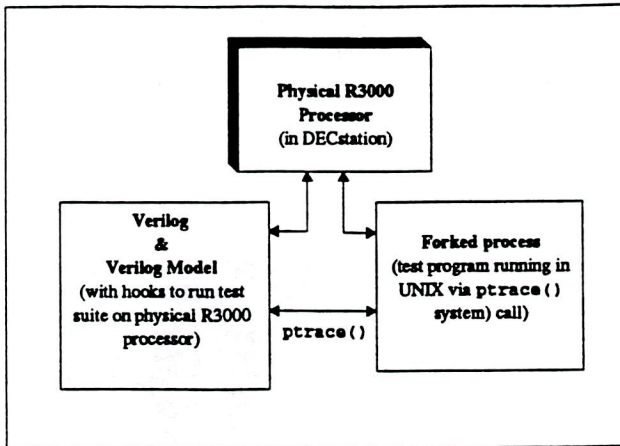


Figure 3: Verilog model and Verify running on a DECstation 5000

our model to the behavior of the physical CPU by running binaries on both the model and the workstation as shown in Figure 3.

Leveraging this capability, we embedded into Verilog a set of PLI primitives that allow us to concurrently execute the same binary on both the R3000 (DECstation 5000) and the Verilog model. These primitives are described in Table 2.

Using these primitives, we debug our model as follows:

1. Using `$fork(object_file)`, we create a process that executes the same binary file as our model.
2. Once the process is forked, we query the process for the

current state of its CPU registers using the `$return_reg()` primitive.

3. We load the model with its copy of the object code and force the model's registers to the same values as those returned from the physical CPU. At this point, both the model and the forked process (running on the actual CPU) are in identical states.
4. We execute one instruction on the model and single-step the forked process with the `$step(1)` task.
5. The forked process is queried for its current state which is the contents of its registers. These values are compared against the values in the model.
6. Steps 4 and 5 are repeated until any discrepancy between the physical CPU's registers and the model's registers is discovered. This discrepancy is reported back to the user.

The entire debugging process is automated through several Verilog tasks which allow the user to specify the binary file and start the debugging process. The verification system will run until an error is discovered or the program completes.

To help view the progress of the testing system, an X-window interface to Verify was written and embedded into Verilog. It allows the user to see the state of the model and physical CPU. Whenever a difference between the model and the physical CPU occurs, a flag appears on the display pointing to the location of the error. Both the Verilog and Verify displays are shown in Figure 4.

Task Name	Description
<code>\$fork(object_file)</code>	Forks off a process that will execute the binary in <code>object_file</code>
<code>\$step(num_of_instructions)</code>	Forces the forked process to execute the number of instructions specified as the parameter by making repeated calls to the unix system call <code>ptrace()</code>
<code>\$goto(program_counter)</code>	Forces the forked process to execute until the PC register equals <code>program_counter</code>
<code>\$return_reg(reg_num)</code>	Returns the current contents of register <code>reg_num</code>
<code>\$return_mem(address)</code>	Returns the contents of memory location <code>address</code>
<code>\$restart</code>	Restarts the forked process


Table 2: Verify Primitives

Support

This work is supported by the Defense Advanced Research Projects Agency under DARPA/ARO Contract No. DAAL03-90-C0028, by the U.S. Army Research Office under the URI Program, Contract No. DAAL03-87-K-0007.

References

- 1 T. Mudge, et. al., "The Design of a Microsupercomputer", *IEEE Computer*, January, 1991.
- 2 Finesse Reference Manual, Cascade Design Automation, 15
- 3 ChipCrafter Designer's Handbook, Cascade Design Automtion, 1990.
- 4 G. Kane, *Mips RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J., 1988.

☒ GaAs MIPS


MIPS and GaAs Comparison

Reg[0]	00000000	00000000	Reg[16]	00000001	00000001
Reg[1]	00000000	00000000	Reg[17]	00000001	00000001
Reg[2]	10000000	10000000	Reg[18]	10005714	10005714
Reg[3]	7FFFbc88	7FFFbc88	Reg[19]	101ab198	101ab198
Reg[4]	00000000	00000000	Reg[20]	100130e0	100130e0
Reg[5]	00000000	00000000	Reg[21]	1001f580	1001f580
Reg[6]	7FFFbcc8	7FFFbcc8	Reg[22]	7FFF74bc	7FFF74bc
Reg[7]	00000000	00000000	Reg[23]	00000000	00000000
Reg[8]	0000004b	0000004b	Reg[24]	8017a040	8017a040
Reg[9]	80a88128	80a88128	Reg[25]	00000000	00000000
Reg[10]	00000001	00000001	Reg[26]	00000000	00000000
Reg[11]	00000001	00000001	Reg[27]	00400218	00400190
Reg[12]	00000001	00000001	Reg[28]	100088e0	100088e0
Reg[13]	00000001	00000001	Reg[29]	7FFFbc60	7FFFbc60
Reg[14]	00000001	00000001	Reg[30]	00000000	00000000
Reg[15]	00000001	00000001	Reg[31]	00400158	00400158

next pc 0040021c nop
curr pc 00400218 lw r8,-32724(gp)

☒ REGS

REDRW	LINE	LINE	PAGE	PAGE	695
<<					>>

Stage I Instruction Fetch

Iaddr	00400218	PCinc	1	00400218	PCmux	00400218
Iin	00000000	PCadd	2	00400218		
IR2	10200007	PCjump	4		PC1	00400218
	10200007	PCvector	8	00000000		00400218
	10200007	PCregister	16	00000001		

Stage II Register Fetch

zero	00000000	r0	t0	0000004b	r8	s0	00000001	r16	k0	00000000	r26
at	00000000	r1	t1	80a88128	r9	s1	00000001	r17	k1	00400190	r27
v0	00000000	r2	t3	00000001	r11	s3	101ab198	r19	gp	100088e0	r28
v1	7FFFbc88	r3	t4	00000001	r12	s4	100130e0	r20	sp	7FFFbc60	r29
			t5	00000001	r13	s5	1001f580	r21	ra	00400158	r31
a0	00000000	r4	t6	00000001	r14	s6	7FFF74bc	r22			
a1	00000000	r5	t7	00000001	r15	s7	00000000	r23	RS	00000000	1
a2	7FFFbcc8	r6	t8	8017a040	r24	s8	00000000	r30	RT	00000000	0
a3	00000000	r7	t9	00000000	r25				RD	00000000	0

b3 1 00000001 b4 2 00000000 RSreg 0 00000000 RTreg 0 00000000 immed 8 *****

IR2 1 beq -1,r0,0x20 Smux 1 00000001 Tmux 4 00000000 Dmux 4 00000000

2 beq -1,r0,0x20 Sreg 1 00000000 Treg 1 00000000 D3reg 1 00000000

Stage III Execute

IR3 1 slt -1,r25,r8 X 00000000 Y 00000000

2 slt -1,r25,r8 Z 00000001

QCsel 0 Z 1

ALUsel 00 Z3reg 0 00000001 L3reg 1 00400218 D3reg 0 00000000

1 00000001 L4reg 1 00400218 MDreg 1 00000000

Stage IV Memory Access

IR4 1 ncp Daddr 00000000 Dwrite 0 Dread 0 Dout 00000000

2 ncp D1n 00000000

Stage V Write Back

Z4reg 0 00000000 L4reg 0 00400218 D4reg 0 00000000

☒ ornette:/tmp/bassoon/gaas/work

```

C249: $stop at simulation time 69
C250: $stop at simulation time 69
69 PC:00400224 IR:0328082a slt r
C251: $stop at simulation time 69
L407 "utilities": $stop at simula
C257 > ^Z
Stopped
ornette [58]xwd -root -out foo
    
```

Figure 4: The Verilog Model Display and Verify