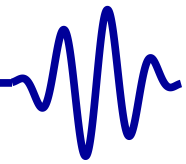
Gisselquist
Technology, LLC

Lessons Learned while Verifying the ZipCPU

Daniel E. Gisselquist, Ph.D.

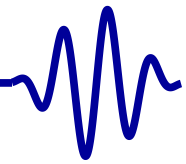
Overview



- What is formal verification?
- My expectations before starting
- Initial Formal Properties / Lessons Learned
  - Verifying the cache(s)
  - Instruction Decoder
  - Debug Port
  - Abstraction and the Multiplier
  - Aggregation
  - Multi-Pass Verification
- Bugs found and fixed

We'll be discussing lessons learned along the way

Formal Properties

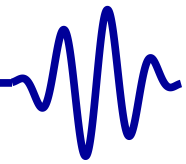


Three types of properties:

1. **assume**(): Limits the search space
2. **assert**(): Should never happen
   The solver *wins* if it can find a way to break an assertion
   A trace is created if an assertion can be made to fail
3. **cover**(): Prove something can happen
   A trace is provided with every success

Given that my assumptions hold, prove that my assertions hold

Formal Properties

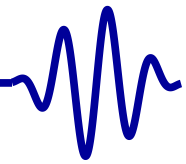


Three types of properties:

1. **assume**(): Limits the search space
2. **assert**(): Should never happen
   The solver *wins* if it can find a way to break an assertion
   A trace is created if an assertion can be made to fail
3. **cover**(): Prove something can happen
   A trace is provided with every success

Given that my assumptions hold, prove that my assertions hold
"The solver is quite a bastard isn't he?" Yes he is.

Formal Properties

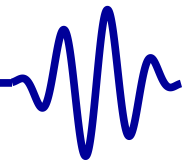


Three types of properties:

1. **assume**(): Limits the search space
2. **assert**(): Should never happen
   The solver *wins* if it can find a way to break an assertion
   A trace is created if an assertion can be made to fail
3. **cover**(): Prove something can happen
   A trace is provided with every success

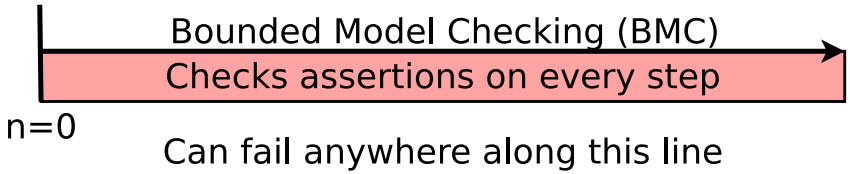
Given that my assumptions hold, prove that my assertions hold
"The solver is quite a bastard isn't he?" Yes he is.
For reasoning with time

- **$past**(x): Value of X one clock ago

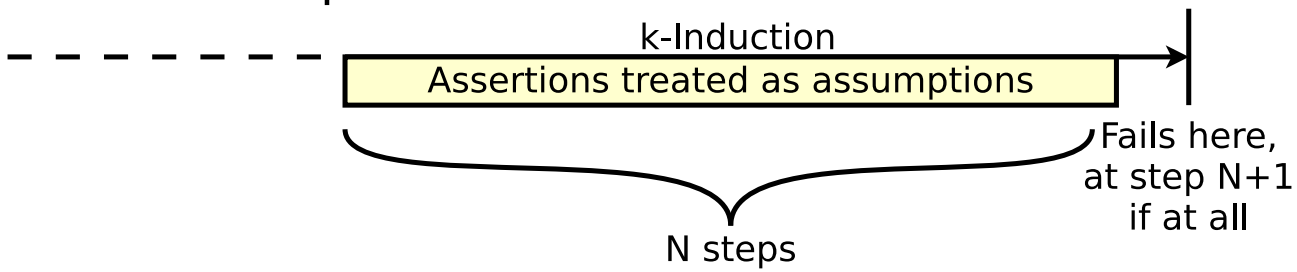


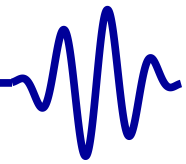
BMC vs Induction

- BMC, the base case for induction



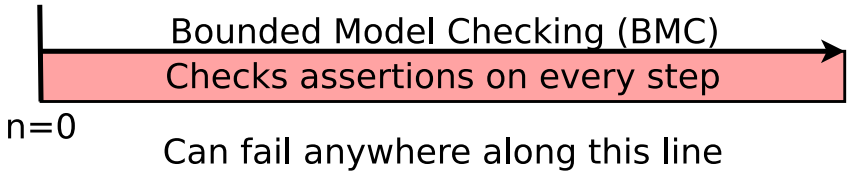
- Induction step



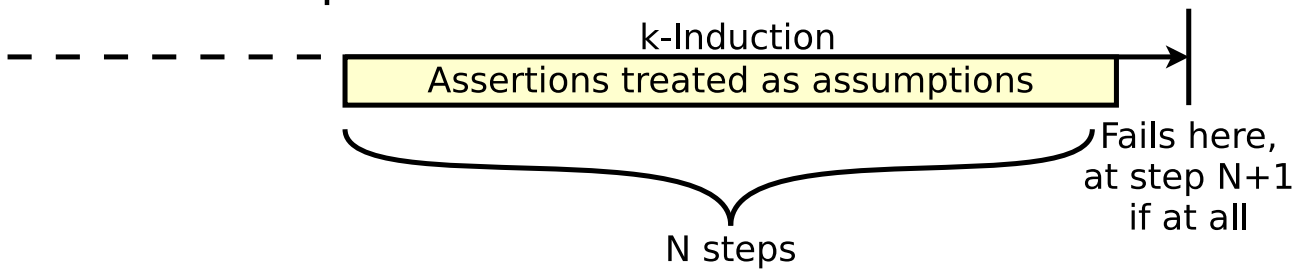


BMC vs Induction

- BMC, the base case for induction

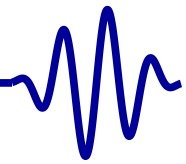


- Induction step



BMC	Induction
Simple Properties	Invasive properties
Black Box possible	White Box
Riscv-formal	Specialized (ZipCPU) proof
Finds failures	Proves success

Expectations



Before starting

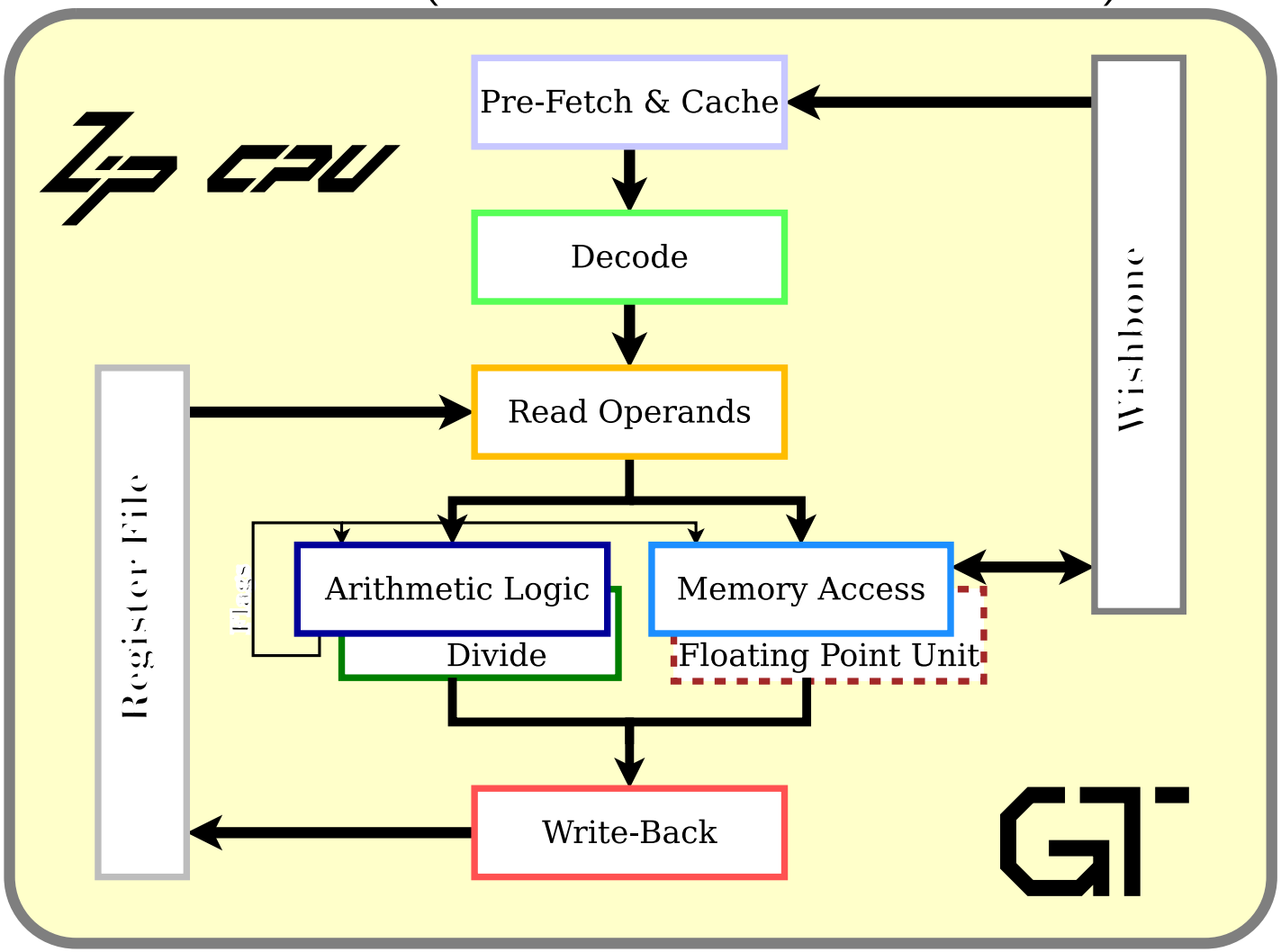
- Formally verified all the components
  - Core: Prefetch, I-cache, Decoder, ALU, divide, memory unit
  - Peripherals: timer, counter, interrupt controller
  - Others: bus arbiter(s), delay, etc.
- Only the top level remained

Initially, concerned with pipeline bugs

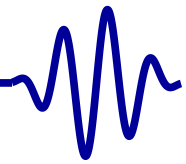
- Vanishing instructions
- Duplicated instructions
- Register forwarding bugs

Properties

4 Exectution units (ALU, MPY, DIV, MEM, DBG)



Expectations

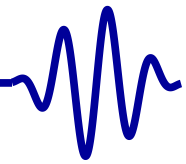


4 Exectution units (ALU, MPY, DIV, MEM, DBG)

- Only one write to register file at a time

```
wire     [2:0] valid_wires;
assign valid_wires = { alu_valid,
                       div_valid, mem_valid };
always @(*)
        assert((valid_wires == 0)
                ||($onehot(valid_wires)));
```

Properties

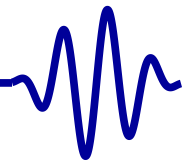


4 Exectution units (ALU, MPY, DIV, MEM, DBG)

- Only write to register file at a time
- Memory operations cannot be rolled back

```
always @(*)
if (mem_rdbusy)
        // No branches allowed
        // No traps allowed
        assert(!new_pc);
```

Properties



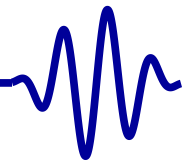
4 Exectution units (ALU, MPY, DIV, MEM, DBG)

- Only write to register file at a time
- Memory operations cannot be rolled back
- Operands going into the execute units *must* match the current register state
- Wishbone interactions must follow interface properties
- Instructions from PF are constant until accepted

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(pf_valid))
                  &&($past(dcd_stalled)))
        assert(($stable(pf_valid))
                  &&($stable(pf_instruction)));
```

This was a common criteria for several stages

Instruction Cache

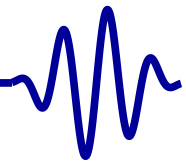


Three necessary properties

- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

Instruction Cache



Three necessary properties

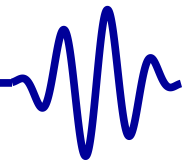
- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

1. **assume**() the bus response

```
always @(*)
if ((i_wb_ack)&&(returned_address == f_addr))
    assume(i_wb_data == f_data);
```

Instruction Cache



Three necessary properties

- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

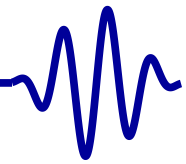
1. **assume**() the bus response

```
always @(*)
if ((i_wb_ack)&&(returned_address == f_addr))
    assume(i_wb_data == f_data);
```

2. **assert**() the cache holds the correct value

```
always @(*)
if (address_is_in_the_cache)
    assert(cache[f_addr[CS-1:0]] == f_data);
```

Instruction Cache



Three necessary properties

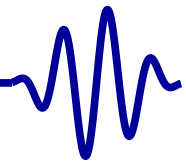
- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

1. **assume**() the bus response
2. **assert**() the cache holds the correct value
3. **assert**() the cache return

```
always @(*)
if ((pf_valid)&&(pf_instruction_pc == f_addr))
    assert(pf_instruction == f_data);
```

Instruction Cache



Three necessary properties

- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

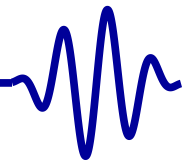
1. **assume**() the bus response
2. **assert**() the cache holds the correct value
3. **assert**() the cache return

```
always @(*)
if ((pf_valid)&&(pf_instruction_pc == f_addr))
    assert(pf_instruction == f_data);
```

Lesson learned:

- Verifying cache components is *really easy*!

Instruction Cache



Three necessary properties

- Pick an arbitrary address and its value

```
(* anyconst *) reg [31:0] f_addr, f_data;
```

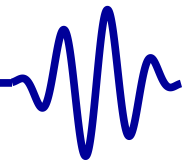
1. **assume**() the bus response
2. **assert**() the cache holds the correct value
3. **assert**() the cache return

```
always @(*)
if ((pf_valid)&&(pf_instruction_pc == f_addr))
    assert(pf_instruction == f_data);
```

Lesson learned:

- Verifying cache components is *really easy*!
- Easier than building the cache in the first place

Instruction Cache



The **cover**() statement is *very* powerful

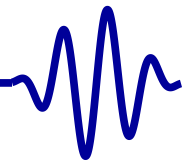
```
always @(posedge i_clk)
        cover(pf_valid);
```

What must happen to make this true?

- Reset
- Instruction Request
- Cache miss
- Fill the cache line
- Return a value from the cache

All returned in a trace.

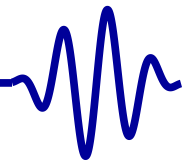
Data Cache



Applied the same methods to the data cache

- Developed using Formal Methods
- Still had one bug in simulation
- . . . but only one bug

Data Cache



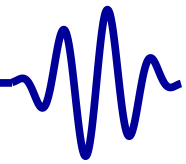
Applied the same methods to the data cache

- Developed using Formal Methods
- Still had one bug in simulation
- . . . but only one bug

Lesson learned:

- Formal methods find the most bugs
- Tools can return quickly
- Resulting trace points directly to bug
- Minimum number of logic steps necessary
- Still needed simulation

Instruction Decoder



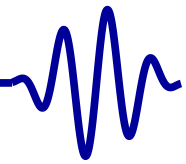
Mistake:

- Only tested the transitions

```
always @(posedge i_clk)
if ((f_past_valid)&&($past(dcd_ce)))
        assert(o_dcdR == $past(insn[30:25]));

assert property (@(posedge i_clk)
        dcd_ce
        |=> o_dcdR == $past(insn[30:25]));
```

- The check doesn't apply when the pipeline is stalled
- Invalid states not caught during induction
  - Ex: Might decode into divide *and* ALU op

Instruction Decoder



Lesson learned:

- Should have verified the outputs instead of the transitions

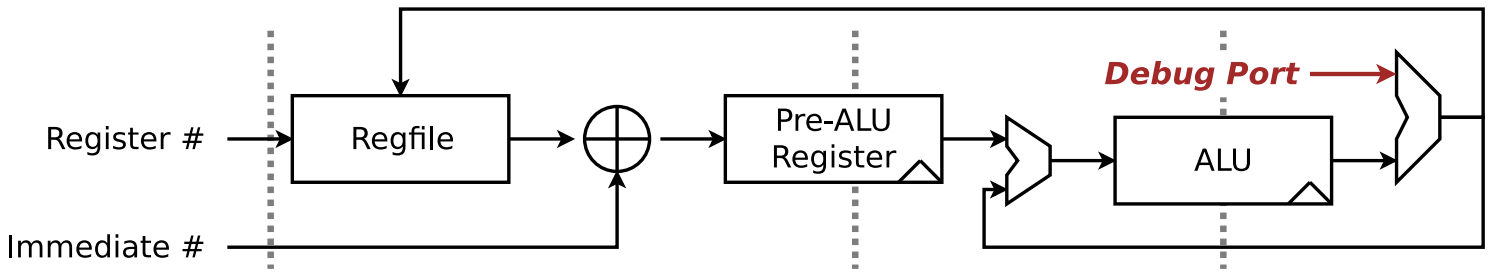
```
always @(posedge i_clk)
if (dcd_ce)
        f_last_insn <= pf_insn;

always @(*)
if (dcd_valid)
        assert(o_dcdR == f_last_insn[30:25]);
```

- This check is applied at all times
- Even when the pipeline is stalled

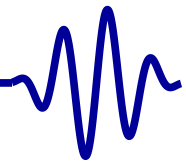
Debug Port

The ZipCPU has a debugging port

- Reset/halt CPU
- Read/set registers within the CPU

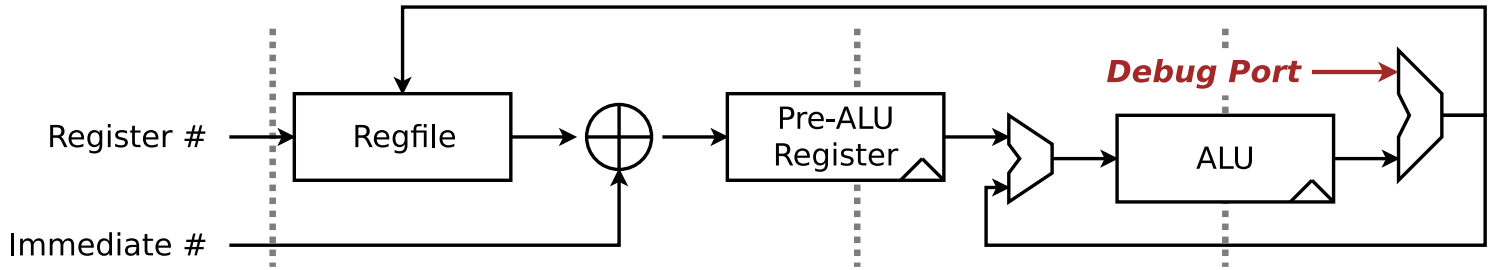


Debug Port



The ZipCPU has a debugging port

- Reset/halt CPU
- Read/set registers within the CPU



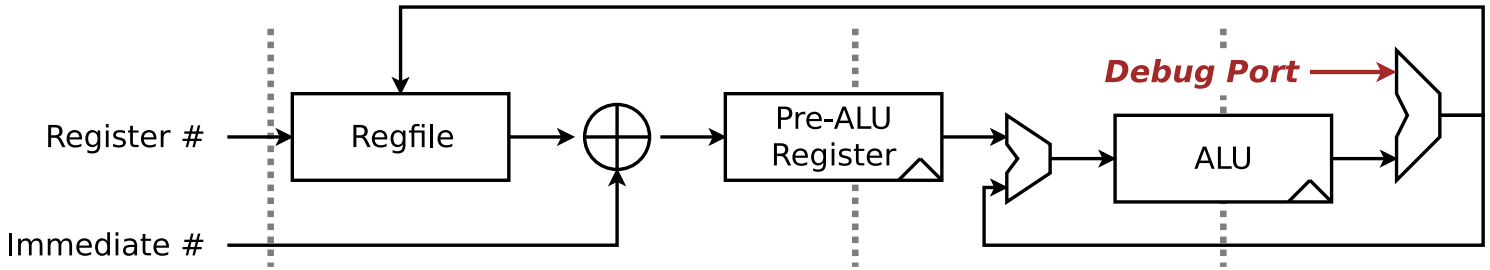
This simple interface caused no end of problems!

- At one time, I assumed no debug access just to keep focused
- Problem was the pipeline
- Solution was to reload the pipeline on any debug write

Debug Port

The ZipCPU has a debugging port

- Reset/halt CPU
- Read/set registers within the CPU



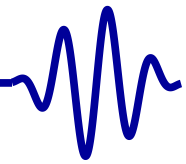
This simple interface caused no end of problems!

- At one time, I assumed no debug access just to keep focused
- Problem was the pipeline
- Solution was to reload the pipeline on any debug write

Lesson learned:

- Simple things aren't

Abstraction



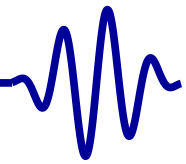
Ex: Multiply

- Returns an arbitrary value

```
module abs_mpy(i_a, i_b, o_r);
(* anyseq *) wire [W-1:0] result;
always @(*)
        if ((i_a == 0)||(i_b == 0))
                assume(result == 0);
        else
                assume(result != 0);
assign  o_r = result;
```

- Solver picks `result`
- Require: maintains signaling
- Prove CPU logic works

Abstraction



Ex: Multiply

- Returns an arbitrary value
- Solver picks `result`
- Require: maintains signaling
- Prove CPU logic works

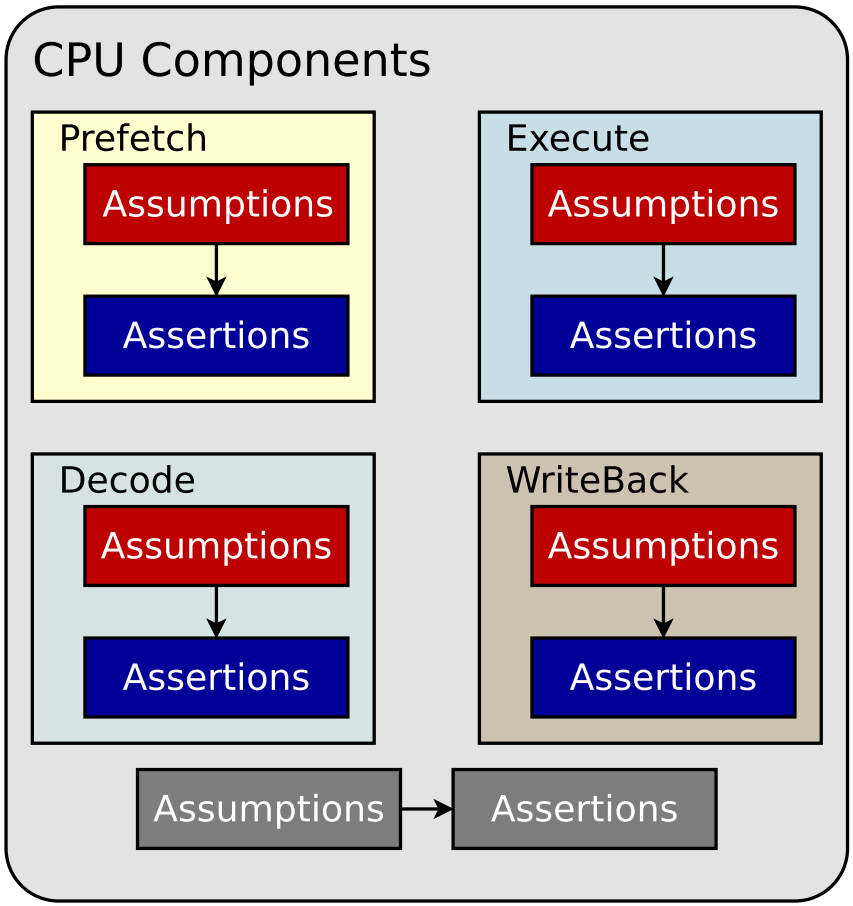
Reality:

- Top level CPU worked and proven
- Missed a bug in the actual multiplier

Lesson Learned:

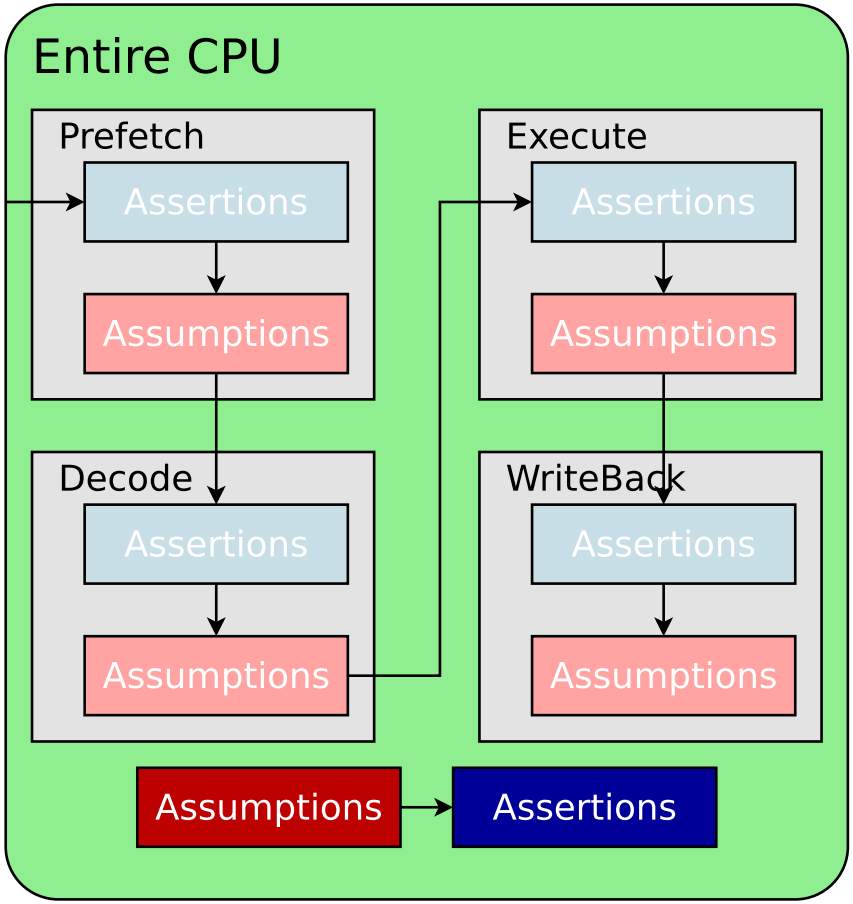
- Create a property file for each interface
  - Prefetch, decoder, ALU, memory unit

Aggregation



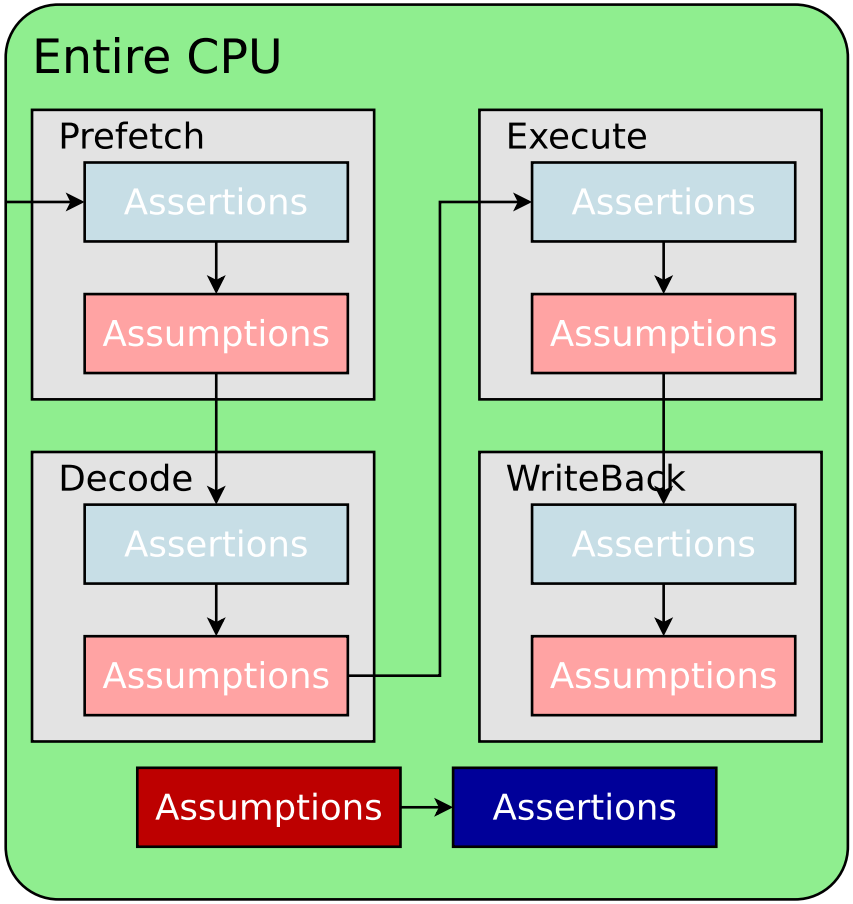
Prove every component before beginning

Aggregation



- Swap component assertions with assumptions
- Whole new set of CPU properties

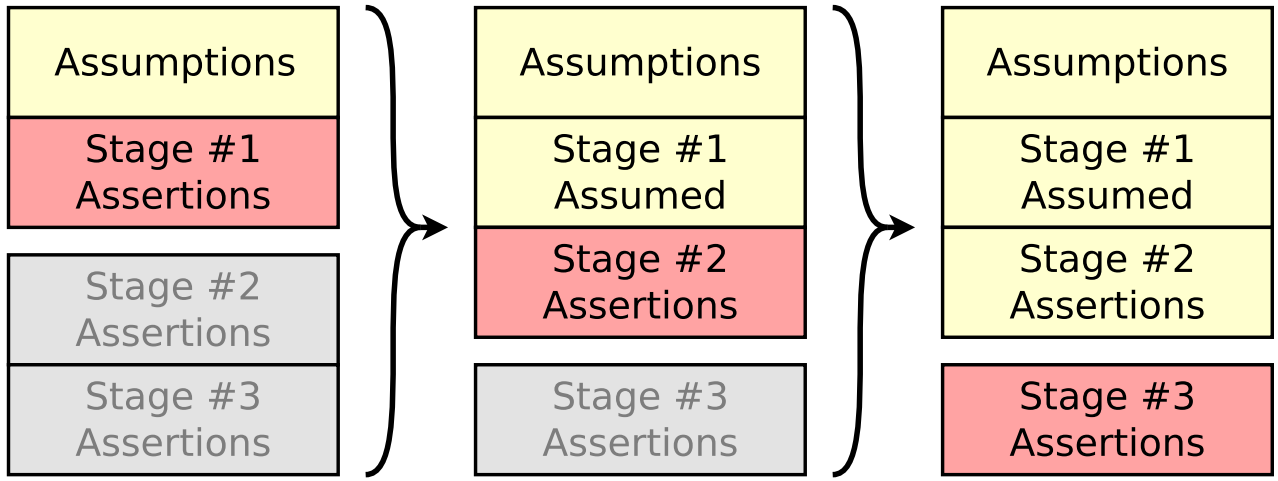
Aggregation



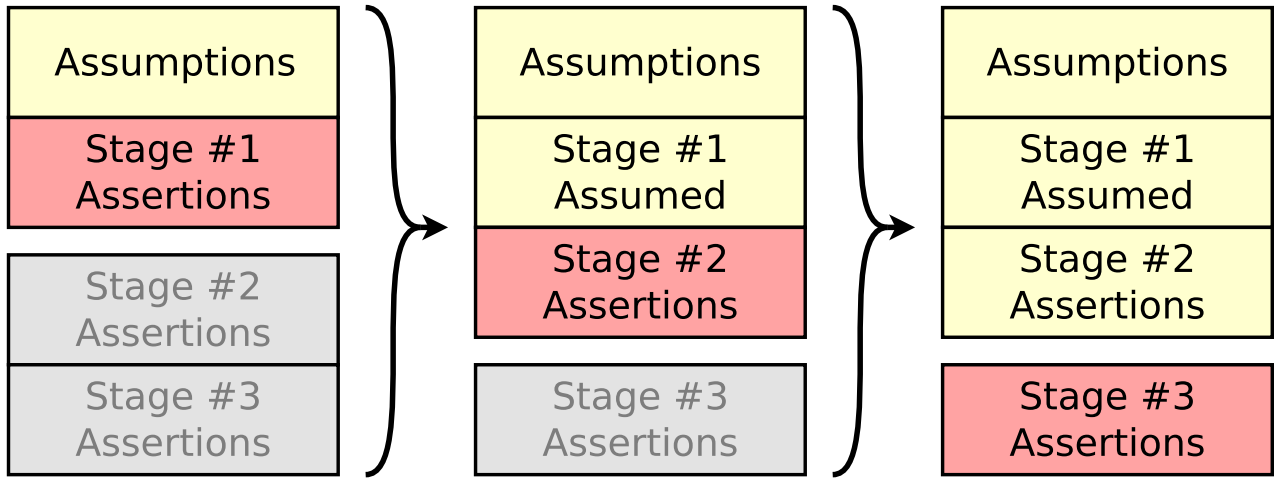
Lesson learned:

- Sub-module assumptions aren't given
- The need to be proven

Multi-Pass Verification



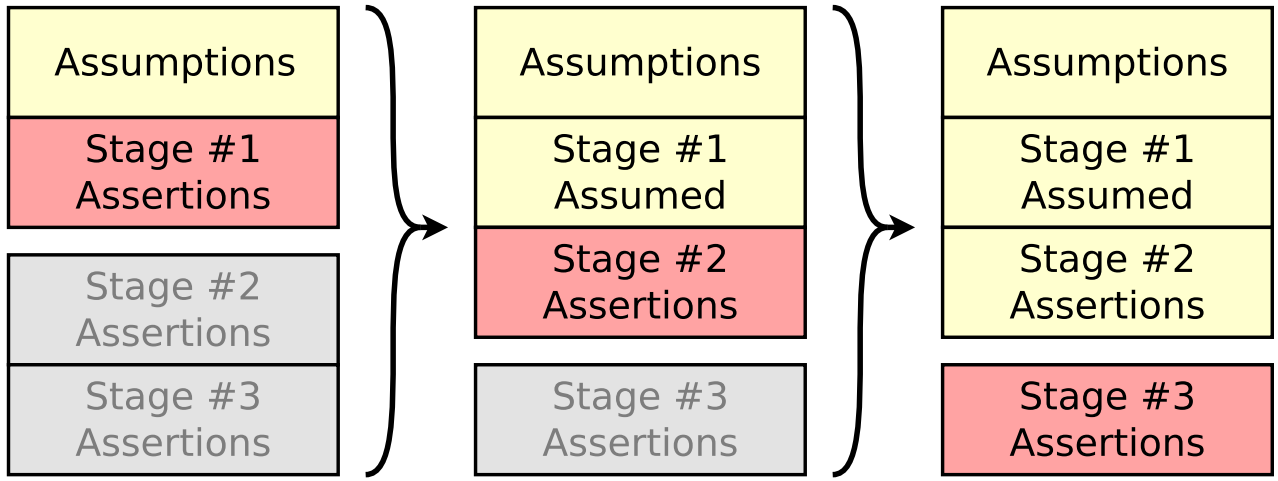
Multi-Pass Verification



Anything you've proved, . . .

- . . . can become assumptions to prove something more

Multi-Pass Verification



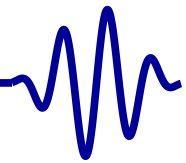
Anything you've proved, . . .

- . . . can become assumptions to prove something more

Must be done in order

- You can't *assume* stage #1 until you've first proven it via assertions.
- Any logic change will send you back to the beginning

Two-Pass Verification



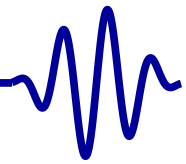
First pass

- Prove component assumptions
- Ad-hoc assertions
- Pipeline assertions

Second pass

- Assume a known instruction
- Verify its implementation

Two-Pass Verification



First pass

- Prove component assumptions
- Ad-hoc assertions
- Pipeline assertions

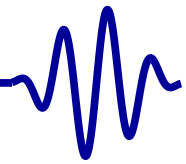
Second pass

- Assume a known instruction
- Verify its implementation

Lesson Learned:

- Assuming a known instruction was a waste of time
- First pass assertions were not trivial
- Most logic proved on the first pass
- One pass would've been easier and simpler

ZipCPU Bugs Fixed



- Bus error on instruction read might not halt CPU
- Memory reads into the program counter didn't stall the pipeline
- Interrupts might break compressed instruction words
- Debug register writes broke register values in the pipeline
- CPU might halt mid-compressed instruction pair
- Multicycle ALU operations (i.e. MPY's) set the wrong flags
- Divides would start before multiplies were finished
- Break instructions might get ignored
- Memory instructions might still be issued while an illegal instruction exception was pending
- Memory FIFO had no overflow protection
- CPU would switch to an interrupt state before completing memory operations

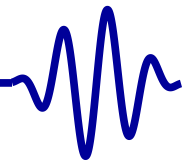
I was very glad I did it!

Lesson Learned

Before using formal methods

- Simulated many programs on the ZipCPU
- Applied the CPU to many FPGA Boards
- Debugging on an FPGA is difficult
- Simulation requires GB+ traces

Lesson Learned



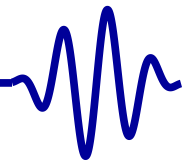
Before using formal methods

- Simulated many programs on the ZipCPU
- Applied the CPU to many FPGA Boards
- Debugging on an FPGA is difficult
- Simulation requires GB+ traces

With formal

- Simulation alone didn't cut it
- Even an incomplete proof is valuable
- What you don't prove, will surprise you
- Simulation requires GB of trace, formal 20-60kB
- Still needed simulation
- Can take a simulation symptom, and recreate it to fix it

Next time



If I had to do it over . . .

- I'd start with formal verification
- . . . even before Simulation
- . . . definitely before code bloat

Gisselquist
Technology, LLC

In all labour there is profit . . .

Prov 14:23a

CPU Survey

Feature	NiOS	μBlaze	ECO–32	RISC-V	OpenRISC	LM32	ZipCPU
Open Architecture?	**No**		Yes				
Number of Instructions	**86**	**129**	**61**	**50+**	**48+**	**62**	28+
OpCode Bits	6–17	6–11	6	10	6–32	6	5
Interrupt/Exception Vectors	1	6	2	9+	14	32	None
Register Indirect plus displacement (bits)	16			12	16		14
Immediate direct addressing (bits)	16, using R0=0						18
Relative branching (bits)	16		26 (28)	21	26	21	18
Conditional branching (bits)	16		16 (18)	13	26	16	18
Register Size (bits)	32			32 (Opt. 64 Exts.)		32	32-bits
Special Purpose Registers	**6**	**25**	**6**	**66+**	**65+**	**10**	1 (x2)
General Purpose Registers	32 (but R0=0, others are unusable, . . . 24)						14 (x2)
8–bit data	Yes						Yes
16–bit data	Yes						Yes
32–bit data	Yes						Yes
64–bit data	No			Yes, by extension		No	Yes
32–bit floats	Optional		No	Yes, by extension		No	Yes, not native
MMU	Yes, but optional						Verified
Instruction Cache	Yes, configurable						Optional
Data Cache	Yes, configurable						Optional