A Debugger for the b16 CPU

Bernd Paysan

EuroForth 2008, Vienna

Abstract

A debugging interface for the b16 CPU is shown. A few lines of Verilog and some small Forth programs are sufficient to add a classical debugging interface to this small CPU. Integration of other controls (like test equipment) is very easy to do.

1 Motivation

For the current project with the b16 core [1] inside, a few things are "unusual":

- Firmware programmer isn't a Forth expert (i.e. not me)
- Program in writable memory (first test chip: RAM, final chip: Flash or OTP)

Under these circumstances, it makes some sense to debug the firmware using a "classical" in–circuit–debugger. It will turn out that adding such a debugger to the hardware is a fairly trivial exercise, leaving writing the software as "main" challenge.

The features such a debugger should have are quite common:

- Interface the chip with a PC, so that the PC can control memory content (and memory mapped IO registers)
- The debugging window should show the source code, and jump with the cursor to the currently executed location (if the CPU is halted)
- Typical commands: Single step, multiple steps, run/stop, set/clear breakpoint
- Direct access to a memory location, dump of a consecutive memory block
- Optional: Forth console to mix debugging commands with other instructions (e.g. measurement and stimuli equipment driven by serial lines)

What's missing

- Classical command line for the embedded CPU

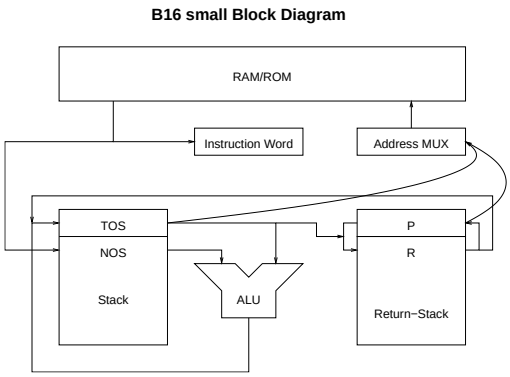

Figure 1: Block Diagram

1.1 Architectural Overview

Just to recap: The core components of the b16 are

- An ALU
- A data stack with top and next of stack (T and N) as inputs for the ALU
- A return stack with top R
- An instruction pointer P
- An instruction latch I

Figure 1 shows a block diagram.

2 Adding In–Circuit Debugging

From a previous project, we already had two important parts: The CPU in a shape that's useful for the project (multiplication and division, dropped back then, were added again), and a SPI–derived interface to directly access memory from outside. The interface uses only two pins, by sharing DI/DO, and interpreting activity on the clock line as chip select (with timeout). The device is not pad limited, but the package gets cheaper with less pins; a standard SPI interface that allows tristating DO can talk to this chip without problems. To interface with the PC, an FTDI module is used (bit–banging mode of serial port interface).

So the missing link was the actual debugger.

The registers were implemented in the order described here. This turned out as a not quite clever idea, but it was possible to work around the problem. The SPI interface can read multiple words in one go, by incrementing the address latch after each read access. This has the side effect that each read sequence ends with a read to the next memory location, even if this data is never used (it just has to be available on the next rising clock edge).

2.1 Implementation

For debugging purposes, all registers are memory read–writable. This requires an external bus master attached to the debugging interface. It's only active when the processor is stopped, so the processor itself can't access its own registers.

The debugging module offers the following registers as address space:

Address	*read*	*write*
$FFE0	P	P
$FFE2	T	T
$FFE4	R	R
$FFE6	I	I
$FFE8	state	state
$FFEA	stack[sp]	push+T
$FFEC	rstack[rp]	pushr+R
$FFEE	stop	start/step

The address $FFEE is special, since a read access to it stops the CPU. By writing to $FFEE, the debugger can either continue the program (write 1 there), or cause it to single step (write 0 there).

⟨*debugging read* 2a⟩≡
```
reg `L dout;

always @(daddr or dr or run or
         P or T or R or I or
         state or sp or rp or c)
if(!dr || run) dout <= 'hz;
else casez(daddr)
   3'h0: dout <= P;
   3'h1: dout <= T;
   3'h2: dout <= R;
   3'h3: dout <= I;
   3'h4: dout <= { run, 4'h0, c, state,
                   {4-sdep{1'b0}}, sp,
                   {4-rdep{1'b0}}, rp };
   3'h5: dout <= N;
   3'h6: dout <= toR;
   3'h?: dout <= 0;
endcase
```

⟨*debugging-ports* 2b⟩≡
```
input [2:0] daddr;
input dr, dw;
input `L din;
output `L dout;
```

⟨*debugging* 2c⟩≡
```
if(dw) casez(daddr)
   3'h0: P <= din;
   3'h1: T <= din;
   3'h2: R <= din;
   3'h3: I <= din;
   3'h4: { c, state, sp, rp } <=
           { din[10:8],
             din[sdep+3:4], din[rdep-1:0] };
   3'h5: { sp, T } <= { spdec, din };
   3'h6: { rp, R } <= { rpdec, din };
endcase
if(dr) casez(daddr)
   3'h5: sp <= spinc;
   3'h6: rp <= rpinc;
endcase
```

⟨*debugger* 2d⟩≡
```
module debugger(clk, nreset,
                addr, data, r, w,
                drun, dr, dw);
parameter l=16, dbgaddr = 12'hFFE;
input clk, nreset, r;
input [1:0] w;
input `L addr, data;
output drun, dr, dw;

reg drun, drun1;
wire dsel = (addr[l-1:4] == dbgaddr);
assign dr = dsel & r;
assign dw = dsel & |w;

always @(posedge clk or negedge nreset)
if(!nreset) begin
   drun <= 1;
   drun1 <= 1;
end else begin
   drun <= drun1;
   if((dr | dw) && (addr[3:1] == 3'h7)) begin
      drun <= !dr & dw;
      drun1 <= !dr & dw & data[0];
   end
end

endmodule
```

⟨*dbg senselist* 2e⟩≡
```
or run or dw or daddr
```

⟨*stack debugging* 2f⟩≡
```
if(!run && dw) casez(daddr)
   3'h5: dpush <= 1;
   3'h6: rpush <= 1;
endcase
```

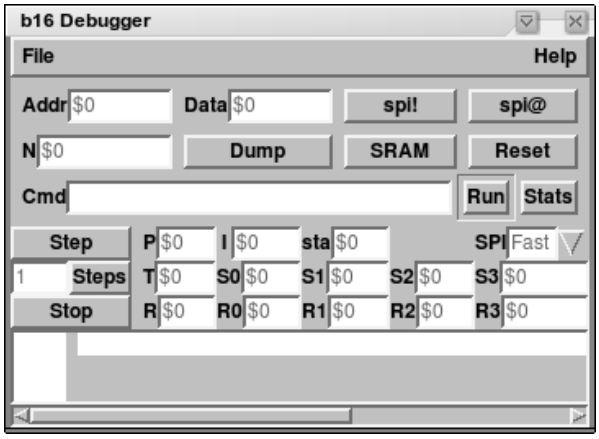

Figure 2: Debugging GUI

3 Debugging Software

The debugging GUI is just a MINOS window which shows the main states and opens a window to the source code (see figure 2).

The SPI interface code was already available from the last project [2] — it needed slight changes, though. First of all, there were two different bit orders of SPI interfaces availbable, both with consistent test environments, and the coworker in charge picked the little endian one[1].The SPI post–access read had a bad effect on the CPU status register read: This would also read the data stack, and as side effect increment the stack pointer. No good idea, especially, since the CPU status register also tells you if the CPU is running or halted by the debugger. A reordering of these registers would be a good idea.

What's worse is that the instruction register content changes the side–effect: Only NOPs really increment the stack pointers, other instructions may interfere with the commands from the debugger. So the workaround found was to first read the four registers P, I, T, and R, and then write back 0 (all NOPs) into the I register. This then allows to read the status plus the two stacks, and then again the two stacks until a full wrap–around of the stack pointers is achieved. Finally, restore the original content of the instruction register.

⟨*read registers 3*⟩≡
```
: load-regs ( - )
  DBG_P regs 4 spiw@s
  0 DBG_I spiw!
  \ clear instruction register to read stacks
  DBG_STATE regs 8 + 3 spiw@s
  stack 16 + stack 4 + DO
      DBG_S[] I 2 spiw@s
  4 +LOOP
  regs 6 + w@ DBG_I spiw! ...
```

[1]Certainly, this sort of software gets written short before the chip arrives from the fab — during device debugging, the proof that this software could be written is sufficient.

3.1 Breakpoints

The original idea how to implement breakpoints was to call the debugger status register. This plan was sabotaged by eliminating loops in the design, so the debugger status register is not accessible by the CPU itself. It won't halt the CPU then, as well. However, it turned out that this idea had been bad for another reason, as well: Calling the debugger status register wastes precious return stack space (20%!), and is not necessary at all. Instead, it's completely sufficient to replace the instruction where you want to break with an empty loop, and check the P register for the breakpoint addresses. The likelyhood that the CPU will be executing the current breakpoint is quite high under these circumstances (however, it's only 1/2, in the other case, the P register points to the next instruction).

If the debugger sees that a breakpoint has been reached, it will stop the CPU. It then has to single–step until the right state is reached (just before loading the instruction). For further execution in single–step mode, the original memory content is restored; only when the CPU goes to "run" mode, the breakpoints have to be restored (run from a breakpoint location then is done by single–stepping to the state where the instruction register has been loaded, then the effect of replacing that instruction by a "breakpoint" loop will not be recognized). Since empty loops are not possible at the last address of a 1k word block, the "workaround" is not fully functional. So far, the firmware is clearly below this 1k words total size limit, anyway.

3.2 Source Window

Looks fairly trivial: Just use the MINOS editor component, and load the source. Next to the editor component, there's a canvas, which can draw the addresses (obtained from the listing), and by clicking on an address, you set/clear a breakpoint. What's a bit less trivial was changing the assembler so that the listing contains meaningful information about the relation between cursor position and address+state of the CPU. So far, there are still a few bugs: `.org` statements don't write out the address into the listing stream (so that the start of the first instruction is not tagged), and the assembler doesn't expand tabs, while the editor window does. So code with tabs will not have the cursor at the right spot.

The source window currently is not an IDE window, i.e. changes won't result in anything. Adding this feature is possible, also adding the feature to automatically reload the source after it has been changed by another editor. However, with the typical Unix environment, people are happy to rerun assembler and restart the debugger after changes in

the source code. Remember: The user isn't expecting this kind of magic anyway, so don't deliver.

4 Integrating other Testing Equipment

After successful deployment of the debugger, coordination between it and other test equipment has been needed quite soon. E.g. to characterize the ADC, you'll want to load a test program that loops ADC conversions and stores them into RAM, and force a certain voltage into the input pin; iterate over this process through the entire input voltage range. Now we are happy that our debugger is nothing but a simple Forth program, and all you need is to add a few other simple Forth words to drive HP instruments over RS232 (nowadays using some USB to serial converters, optimally not from FTDI, to avoid conflicts). Warning: Confusion may arise when you reboot the machine or replug the USB adapters, because the number scheme of USB serial ttys is first come, first serve type. Unfortunately, Intel also forgot to specify a unique per–device ID for this kind of device, so you can't use an alternative naming scheme.

5 Lessons Learned

After a few days work, this debugger was satisfying the "customer" (the coworker doing firmware development and me doing other device testing). It's a fairly trivial program, and the hardware behind is also fairly trivial; trivial enough that the gate count is insignificant. One wonders why in earlier days CPUs with in–circuit debuggers used to be quite expensive; also the cost for an debugger (hardware plus software plus IP) for traditional 8051 clones still is very high — even though that's a product that doesn't go into just one device.

This is a fairly simple approach; for the final device, the breakpoint mechanism e.g. won't work; at least when it's in OTP (and reflashing entire sections to just add a breakpoint is also no good idea). So in the final device, there will be a fairly limited set of breakpoint address registers and comparators.

- If time permits, diverging modules like the SPI should be merged and made configurable
- The register order should be changed so that the stack access doesn't require special care (stack access first)
  - Read with side effect is evil, anyway
- Integrating the assembler into the debugger should be fairly trivial, and thereby it creates an IDE with little effort
- Further magic could allow to seamlessly insert code with just a small stop and restart of the CPU
- Adding some (further) interactivity with the target CPU is also fairly trivial
- Hot–plugged devices must have a unique serial ID (this is a hint to Intel!!!)

References

[1] EuroForth 2004, *b16-small — Less is More*, Bernd Paysan

[2] EuroForth 2007, *Audio GUI: MINOS@work,* Bernd Paysan