A Debugger for the b16 CPU

Bernd Paysan[1]

[1]Diodes Zetex GmbH

EuroForth 2008

Outline

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.

Features

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)
- Missing: Classical command line for the embedded CPU

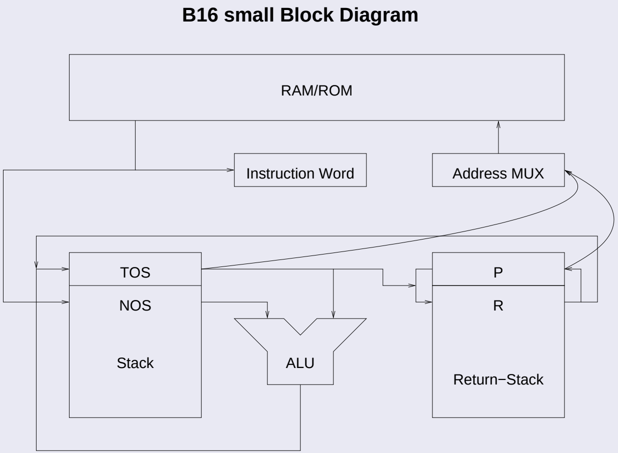
b16 Architecture Overview

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

b16 small Block Diagram

Block Diagram



Available Components

- CPU core (small change: add multiply and div step again)
- SPI interface
    - Two versions: Little and big endian
- Missing: Debugger

Register Structure

Debugging Registers

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

Read Registers

```
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
```

Write Registers

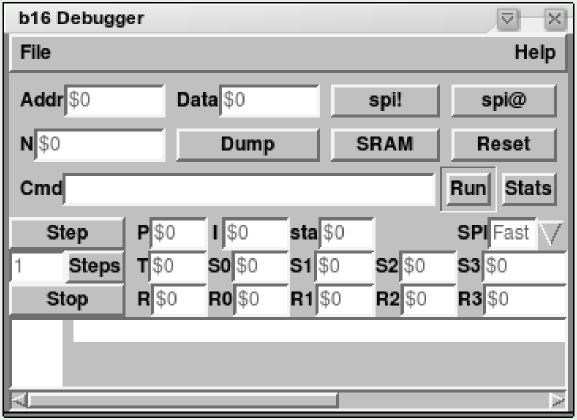
```
if(dw) casez(daddr)
   3'h0: P <= din;
   3'h1: T <= din;
   3'h2: R <= din;
   3'h3: I <= din;
   3'h4: { c, state, sp, rp } <=
           { din[10:8],
             din[sdep+3:4], din[rdep-1:0] };
   3'h5: { sp, T } <= { spdec, din };
   3'h6: { rp, R } <= { rpdec, din };
endcase
```

Debugger Core

```
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
```

Debugger GUI

Debugger GUI



Problems with Readout

- SPI post–access read makes status read problematic (will also modify stack pointer)
- Unexpected side effects of instruction loaded on stack readouts

Solution:

- First read the four registers
- Set I to 0
- Read status & stacks (stacks 4 times)
- Restore I

Readout Code

```
: 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! ...
```

Breakpoints

- Original idea: Call control register address $\longrightarrow$ stops CPU
- Doesn't work due to loop elimination in the design
- Turned out to be a bad idea, anyway (wastes 20% return stack space)
- Solution: Replace instruction by loop to itself.

Source Window

- MINOS editor component: Load the source into it
- Canvas on the left side displays address (for breakpoints)
- Change assembler so that listing contains enough information to translate address+state into cursor position
- No IDE at the moment (changes on the source go nowhere)

Integrating Test Equipment

- Other test equipment (from HP, driven via RS232) needs integration:
    - Voltage source
    - Measurement ADC
- Typical use: Apply voltage, measure with equipment, measure with chip (several times), collect data
- Problem: RS232 nowadays via USB, there's no easy way to know which interface is connected where

Lessons Learned

- 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!!!)

For Further Reading I

- EuroForth 2004, *b16–small — Less is More*, Bernd Paysan
- EuroForth 2007, *Audio GUI: MINOS@work*, Bernd Paysan