Inside the Nut CPU

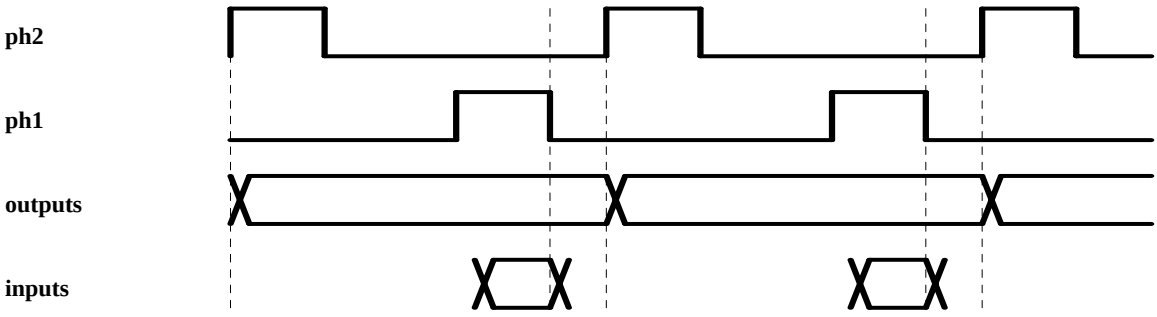
Monte Dalrymple

The HP-41 series has always been my favorite calculator. The machine fits my hand perfectly, and my original 41CV is still on my desk after 25 years of use. A couple of years ago I started a project to reverse-engineer the CPU board in my 41CV. As I have mentioned previously (at HHC2004), my initial goal was to show that moving a design to a new process didn't need to be expensive. During this process I have come to appreciate what an excellent example of engineering the HP-41 series really is. In this paper I will attempt to explain some of what I've found.

Before I go into details though, it is important to place the following comments into the context of the technology that was available at the time. In those days an IC designer might spend a day or more trying to eliminate a half dozen transistors from a design. And package pins were just as scarce, in addition to being a significant source of failures. So while an IC designer today usually has hundreds of pins available, and can afford to waste thousands of transistors to save a schedule day, things were much different when the ICs in the 41CV were first being designed. With this in mind, let me answer some of the questions that you may have had about why the CPU in an HP-41 works the way it does.

Why use a two-phase clock?

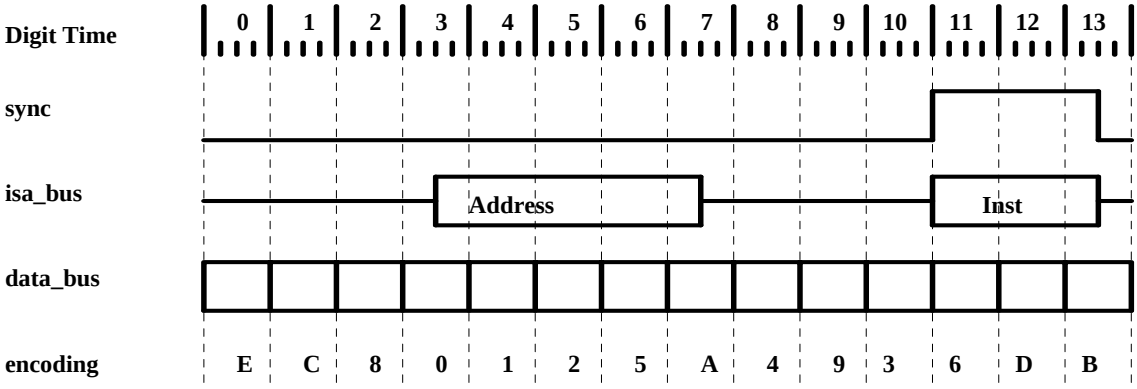
To understand this design decision, think about what a system using an HP-41 might look like. In addition to the mainframe, there might be plug-in modules, a printer, a barcode wand, an HPIL interface, or more. Which means that the bus (and the clock) are going to be distributed physically. Even though the operating frequency is fairly low, the problem of clock skew, where different parts of the circuit "see" the controlling clock edge at different times, is independent of frequency. So even though a two-phase clock means more pins and an extra bus signal, it eliminates any potential problem with clock skew. As shown in the figure below, signals change as a result of the rising edge of the ph2 signal, while signals are sampled by the ph1 signal.



With a two-phase clock it doesn't matter if different parts of the circuit drive or sample signals at slightly different times. In addition, circuitry inside the IC can be much simpler when two clocks with known phase relationships are available

Why the odd digit encoding?

The diagram below shows the bus timing for the CPU, including the encoding for each digit. At first glance the encoding makes little sense. Why not just use straight binary encoding?



The encoding makes perfect sense once you look at the circuit that generates it, which is a modified twisted ring counter shown in the Verilog HDL code below. It's basically just a shift register plus one exclusive-OR gate. Then the counter is further modified to shorten its count sequence to the 14 states required for the 14 digit times. This implementation requires fewer transistors than a straight binary counter. Even the choice of the unused states is optimized, requiring just two gates to shorten the sequence.

```
always @ (posedge ph2 or negedge rstb) begin
  if         (!rstb) state_reg <= 4'b1011;
  else if (ld_cycle) state_reg <= {(state_reg[2] || state_b), state_reg[1:0],
                                   ((state_reg[0] ^ !state_reg[3]) && !state_b)};
  end

assign state_b =  state_reg[3] && !state_reg[2] &&  state_reg[1] &&  state_reg[0];
```

Optimizing the state encoding isn't much use if it makes decoding the states difficult. The code below shows the decoding for the X (exponent), XS (exponent sign), M (mantissa) and MS (mantissa sign) digits. It's only a few gates, and would be even simpler if I had

chosen to take advantage of the unused states in the sequence. I'm sure the original design does take advantage of the unused states.

```
assign te_x    = state_reg[3] && (state_reg[2] || !state_reg[1]) && !state_reg[0];
assign te_xs   = state_reg[3] && ~|state_reg[2:0];
assign te_m    = !state_reg[3] || (!state_reg[1] && state_reg[0]) ||
                 (!state_reg[2] && state_reg[1] && !state_reg[0]);
assign te_ms   = state_reg[3] && !state_reg[2] && &state_reg[1:0];
```

Why does GOTOC use digits 3-4 of the C register?

This one is easy once you look at the bus timing. The GOTOC (Go To C) instruction will use two digits of the C register to load into the PC (Program Counter) to perform a jump to a new location. Since the entire machine is bit-serial, digits 3 and 4 will be the closest (in the time domain) to when the PC is doing its shifting, and so are the logical choice. But why is the PC shift (and address output on the ISA bus) done where it is? I suspect that the timing is a legacy from the original HP-35 timing, where I'm sure it made perfect sense.

How do we explain the CXISA timing?

The CXISA (C Exchange ISA) instruction reads the contents of a program location into digits 0 through 2 of the C register, using the digits 3 through 6 of the C register as the program address. At first glance, digits 11 through 13 seem to be good candidates to use for bringing in the data, until you realize how this data will usually be used. Like the LDI (Load Immediate) instruction, this data will usually be a small number, which is most convenient in digits 0 through 2. This is simple to accomplish by adding a recirculate function to the register that normally assembles the instruction, and uses the XS time enable.

Why does the stack exchange with digit 3-6 of the C register?

By now, this one should be obvious. This is the time slot where the PC is shifting, which means that it is also the time where the stack contents are shifting. Waste not, want not.

How does the auto subroutine return work?

This feature is actually quite clever. If the first instruction of a subroutine called by a JSB (Jump Subroutine) instruction is a NOP (No Operation), the CPU automatically performs a return. This feature protects the user in the case where a plug-in module might have been removed without being registered by the operating system. Since there is a weak pull-down on the ISA bus, jumping to a subroutine at a nonexistent memory location will always be interpreted as a NOP.

Implementing this feature is straightforward. The JSB sets an internal flip-flop that modifies the NOP instruction to act like a RTN (Return) instruction. This flip-flop is automatically cleared at the end of every instruction except JSB.

Why does BRN (Branch) use the current PC?

Most CPUs increment the PC during the current instruction so that it is ready for the next instruction fetch, but this CPU does not. Instead, it outputs and increments at the same time. This means that the BRN instruction has to use the current PC to add with the offset to form the target address.

There's nothing wrong with doing it this way since the machine is bit-serial. In a parallel machine doing a simultaneous increment and output would eat up too much of the memory access time for the increment. I suspect that this behavior is also a legacy from previous generations, and the only obvious impact it has on the design is that the PC is set all ones by reset instead of the usual all zeros.

How is the keyboard encoding generated?

At first glance, you might expect the keyboard scanner to just use the regular state machine to generate the column strobes, but I don't think that this is the case. In the original HP-35 implementation the keyboard scanner was on a separate chip from the ALU, and I think that this fact has influenced later generations.

The code below shows my implementation of the keyboard scanner. Despite the apparent complexity, it really is a minimum gate count design, again using a twisted ring counter.

```
assign keycol_nxt = {((!keycol_reg[3] && keycol_reg[2]) ||
                      (keycol_reg[3] && !keycol_reg[0])),
                     ((!keycol_reg[2] && keycol_reg[1]) ||
                      (keycol_reg[3] && !keycol_reg[0])),
                     ((!keycol_reg[2] && keycol_reg[0]) ||
                      (keycol_reg[2] && !keycol_reg[0])),
                     ((!keycol_reg[2] && keycol_reg[0]) ||
                      (keycol_reg[1] && !keycol_reg[0]))};

always @ (posedge ph2 or negedge rstb) begin
  if         (!rstb) keycol_reg <= 4'b0001;
  else if (ld_cycle) keycol_reg <= (ld_inst) ? 4'b0001 : keycol_nxt;
  end

assign kc0 = !(!keycol_reg[1] &&  keycol_reg[0]);
assign kc1 = !(!keycol_reg[2] &&  keycol_reg[1]);
assign kc2 = !(!keycol_reg[3] &&  keycol_reg[2]);
assign kc3 = !( keycol_reg[3] && !keycol_reg[2]);
assign kc4 = !( keycol_reg[2] && !keycol_reg[1]);
assign kc5 = !( keycol_reg[1] && !keycol_reg[0]);
assign kc6 = !( keycol_reg[3] &&  keycol_reg[0]);
```

The keyboard row decoder is also quite simple, with the apparent complexity a result of the Verilog HDL coding rather than the circuitry itself. This circuit in the technology of the time would have been quite compact, probably using pass transistor logic to do the priority encoding.

```
always @ (posedge ph2 or negedge rstb) begin
  if        (!rstb) keysmpl_reg <= 9'b111111111;
  else if (keytime) keysmpl_reg <= {por, kr7, kr6, kr5, kr4, kr3, kr2, kr1, kr0};
  end

always @ (keysmpl_reg) begin
  casex (keysmpl_reg) //synopsys parallel_case
    9'b0xxxxxxxx: key_row = 4'b1000;
    9'b10xxxxxxx: key_row = 4'b0111;
    9'b110xxxxxx: key_row = 4'b0110;
    9'b1110xxxxx: key_row = 4'b0101;
    9'b11110xxxx: key_row = 4'b0100;
    9'b111110xxx: key_row = 4'b0011;
    9'b1111110xx: key_row = 4'b0010;
    9'b11111110x: key_row = 4'b0001;
    9'b111111110: key_row = 4'b0000;
    default:      key_row = 4'b1111;
    endcase
  end
```

What's up with the TEF (Time Enable Field) when using the PQ option?

Using the PQ option for the TEF can be confusing from the description. If pointer Q is to the left of pointer P (pointing at a more significant digit), then the time enable is from digit P through digit Q, with the carry out of the Q digit. If pointer P is to the left of pointer Q, then the time enable is from digit P to the most significant digit. If the pointers are equal then the operation is on that digit. But it turns out that the straightforward implementation of the first (and normal) case leads directly to the other two cases. No extra logic was required to implement the two "abnormal" cases, as shown in the code below.

```
assign te_pq   = ~|(p_reg ^ state_reg) || tpq_reg;

always @ (posedge ph2 or negedge rstb) begin
  if (!rstb) begin
    teg_reg  <= 1'b0;
    tpq_reg  <= 1'b0;
    twpt_reg <= 1'b0;
    end
  else if (ld_cycle) begin
    teg_reg  <= ~|(ptr_reg ^ state_reg);
    tpq_reg  <= |(q_reg ^ state_reg) && !ld_inst && (~|(p_reg ^ state_reg) || tpq_reg);
    twpt_reg <= ld_inst || (|(ptr_reg ^ state_reg) && twpt_reg);
    end
  end
```

What's up with the odd cases when operating on the C and G registers?

The G register is an eight-bit register that communicates with the C register. There is the C=G (Load C From G) instruction, the G=C (Load G From C) instruction, and the CG EX (Exchange C and G) instruction. The digits of the C register that are to be used by these instructions are selected by the current pointer register, but there are a number of very strange boundary conditions that occur if the pointer is changed with the instruction immediately preceding one of these instructions. I won't detail them here, but they really do appear strange at first glance.

As one might expect, implementing the normal case, without regard for whether or not the pointer has changed, leads directly to the described behavior. The code below shows the generation of the time enable. Refer back to the previous code example for the register operation.

```
assign te_g    = teg_reg || ~|(ptr_reg ^ state_reg);
```

The David Assembler manual calls this behavior an errata, but I don't agree. It's just the behavior that falls out when implementing the normal case with the minimum amount of circuitry.

Why are some instructions not allowed to immediately follow others?

The original CPU documentation states that a number of instructions cannot immediately follow an arithmetic instruction, but it doesn't say why. The instructions with this restriction are the logical operations, instructions affecting or testing the status bits, and the pointer test instruction.

I suspect that the restriction has something to do with the operation of the C flag, but have never done enough investigation to find the root cause. It was easier just to do the design without this restriction.

Conclusion

Reverse-engineering a design is a very rewarding experience, particularly if you understand the constraints that the original designer had to deal with. This experience has taught me new tricks and led to a number of enjoyable "aha!" moments. I truly appreciate the skill of the original designers of the brains of my HP-41CV.