b16-small—Less is More

Bernd Paysan

July 9, 2006

Abstract

b16 is a Forth-CPU inspired by Chuck Moore's current work. It is intended at replacing state machines and other more sequential logic, including calculations. When going into the first SoC design, I shrank b16 down to what was needed for the project, creating an off-spring called b16-small. Surprisingly, the resulting architecture is closer to traditional Forth than the original b16. Uncommon sense results will include that a smaller design doesn't have less lines of code, that you don't need more stack space than 4 or 5 cells, that a 2-digit 7-segment is all you need for debugging, and that 1k of program ROM is enough.

Contents

Introduction

Minimalistic CPUs can be used in many designs. A state machine often is too complicated and too difficult to develop, when there are more than a few states. A program with subroutines can perform a lot more complex tasks, and is easier to develop at the same time. Also, ROM and RAM blocks occupy much less place on silicon than "random logic". That's also valid for FPGAs, where "block RAM" is—in contrast to logic elements—plenty.

The architecture is inspired by the c18 from Chuck Moore [1]. The exact instruction mix is different; it also differs from the b16 core. I left out multiplication and division step, but implemented Forth-typical logic operations. Also, this architecture is byte-addressed.

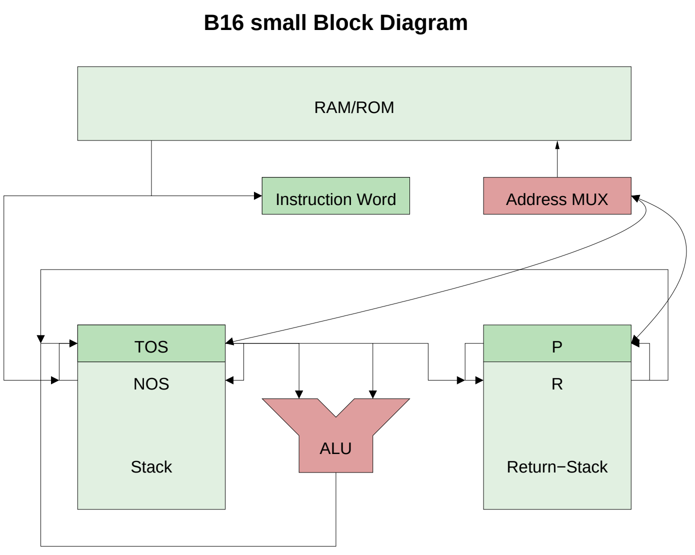

Figure 1: *b16-small Block Diagram*

A word about Verilog: Verilog is a C-like language, but tailored for the purpose to simulate logic, and to write synthesizible code. Variables are bits and bit vectors, and assignments are typically non-blocking, i.e. on assignments first all right sides are computed, and the left sides are modified afterwards. Also, Verilog has events, like changing of values or clock edges, and blocks can wait on them.

This article is an example of literate programming. This is the actual source code. This core is licensed unter the GPL. For more informations, see http://www.jwdt.com/~paysan/b16.html

1 Architectural Overview

The core components are

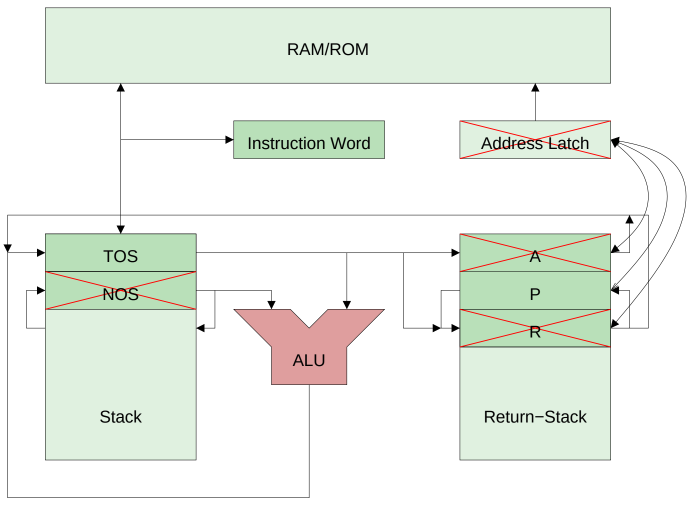

Figure 2: *b16 Block Diagram, changes crossed out*

- An ALU
- A data stack with top and next of stack (T and N) as inputs for the ALU
- A return stack
- An instruction pointer P
- An address mux `addr`, to address external memory
- An instruction latch I

Figure 1 shows the block diagram of b16-small.

Figure 2 shows the original block diagram, crossed out in red what I removed or replaced. I removed the A register. Furthermore, the R register now can't access memory, and is just part of the return stack SRAM block. N was a register in the b16, now it is part of the stack RAM block. This change is on the edge to trouble: Some FPGAs don't have the right asynchronous RAM needed for this. There, it is mandatory to keep N as register. Also, automatic test pattern generation (ATPG) is easier with N as register.

1.1 Register

In addition to the user-visible latches there are control latches for external RAM (`rd` and `wr`), stack pointers (`sp` and `rp`), and a carry `c`.

Name	*Function*
T	Top of Stack
I	Instruction Bundle
P	Program Counter
state	Processor State
sp	Stack Pointer
rp	Return Stack Pointer
c	Carry Flag

⟨*register declarations*⟩≡
```
reg [sdep-1:0] sp;
reg [rdep-1:0] rp;

reg ‘L T, I, P;

reg [1:0] state;
reg c;
```

2 Instruction Set

There are 32 different instructions. Since several instructions fit into a 16 bit word, we call the bits to store the packed instructions in an instruction word "slot", and the instruction word itself "bundle". The arrangement here is 1,5,5,5, i.e. the first slot is only one bit large (the more significant bits are filled with 0), and the others all 5 bits.

The operations in one instruction word are executed one after the other. Each instruction takes one cycle, memory operation (including instruction fetch) need another cycle. Which instruction is to be executed is stored in the variable `state`.

The instruction set is divided into four groups: jumps, ALU, memory, and stack. Table 1 shows an overview over the instruction set. Note: Some special characters indicate functions as follows: "`!`": "store", "`@`": "fetch", "`>`": "to" if before, "from" if afterwards.

Operations will be described using a "stack effect". This is a template for the stack elements before and after the operation, separated by a long dash. The names are listed in the order bottom to top, unchanged stack elements below are not listed.

Jumps use the rest of the instruction word as target address (except `ret`). The lower bits of the instruction pointer P are replaced, there's nothing added. For instructions in the last slot, no address remains, so they use T (TOS) as target.

⟨*instruction selection*⟩≡
```
// instruction and branch target selection
reg [4:0] inst;
reg ‘L jmp;

always @(state or I or data)
   case(state[1:0])
     2’b00: inst <= { 4’b0, data[15] };
     2’b01: inst <=   I[14:10];
     2’b10: inst <=   I[9:5];
     2’b11: inst <=   I[4:0];
   endcase // casez(state)

always @(state or I or P or T or data)
   case(state[1:0])
     2’b00: jmp <= { data[14:0], 1’b0 };
     2’b01: jmp <= { P[15:11], I[9:0], 1’b0 };
     2’b10: jmp <= { P[15:6], I[4:0], 1’b0 };
     2’b11: jmp <= { T[15:1], 1’b0 };
```

	0	1	2	3	4	5	6	7	*Comment*
0	`nop`	`call`	`jmp`	`ret`	`jz`	`jnz`	`jc`	`jnc`	
	`nop`	`exec`	`goto`	`ret`	`gz`	`gnz`	`gc`	`gnc`	*for slot 3*
8	`xor`	`com`	`and`	`or`	`+`	`+c`	`2/`	`c2/`	
10	`!+`	`@+`	`@`	`lit`	`c!+`	`c@+`	`c@`	`litc`	
	`!.`	`@.`	`@`	`lit`	`c!.`	`c@.`	`c@`	`litc`	*for slot 1*
18	`nip`	`drop`	`over`	`dup`	`>r`	`nop`	`r>`	`nop`	

Table 1: *Instruction Set*

```
    endcase // casez(state)
```

The instructions themselves are executed depending on `inst`:

⟨*instructions*⟩≡
```
  casez(inst)
    ⟨control flow⟩
    ⟨ALU operations⟩
    ⟨load/store⟩
    ⟨stack operations⟩
  endcase // case(inst)
```

```
5'b00010: begin
   P <= jmp;
   if(state == 2'b11) `DROP;
end
5'b00011: { rp, c, P } <=
          { rpinc, R[0], R[l-1:1], 1'b0 };
5'b001??: begin
   if((inst[1] ? c : zero) ^ inst[0])
      P <= jmp;
   `DROP;
end
```

2.1 Jumps

In detail, jumps are performed as follows: the target address is stored in the P register. The register P will later be set to the incremented value of the address multiplexer `addr`, after the instruction fetch cycle. Apart from `call`, `jmp` and `ret` there are conditional jumps, which test for 0 and carry. The lowest bit of the return stack is used to save the carry flag across calls. Conditional instructions do consume their test value, like Forth, unlike c18 or b16. This is based on an evaluation of the preliminary application program, which quite a lot of `IF drop` ... `ELSE drop`.

To make it easier to understand, I also define the effect of an instruction in a pseudo language:

nop (—)

call (— r:P) P $\leftarrow$ *jmp*; c $\leftarrow 0$

jmp (—) P $\leftarrow$ *jmp*

ret (r:a —) P $\leftarrow a \wedge$ \$FFFE; c $\leftarrow a \wedge 1$

jz (n —) **if**$(n = 0)$ P $\leftarrow$ *jmp*

jnz (n —) **if**$(n \neq 0)$ P $\leftarrow$ *jmp*

jc (x —) **if**(c) P $\leftarrow$ *jmp*

jnc (x —) **if**$(c = 0)$ P $\leftarrow$ *jmp*

⟨*control flow*⟩≡
```
  5'b00001: begin
    rp <= rpdec;
    P <= jmp;
    c <= 1'b0;
    if(state == 2'b11) `DROP;
  end // case: 5'b00001
```

2.2 ALU Operations

The ALU instructions use the ALU, which computes a result `res` and a carry bit from T and N. The instruction `com` is an exception, since it only inverts T—that doesn't require an ALU.

Ordinary ALU instructions just write the result of the ALU into T and c, and reload N. The original b16 had a double-wide multiplication and division step, which used the A register, too. These operations don't work without A, so I replaced them with two different divide by two instructions, one found in every Forth system (`2/`), the other intended for double-width shifts (`c2/`, shifts carry in). In the application code itself, only `2/` is used.

xor (a b — r) $r \leftarrow a \oplus b$

com (a — r) $r \leftarrow a \oplus$ \$FFFF, c $\leftarrow 1$

and (a b — r) $r \leftarrow a \wedge b$

or (a b — r) $r \leftarrow a \vee b$

+ (a b — r) c$, r \leftarrow a + b$

+c (a b — r) c$, r \leftarrow a + b +$ c

2/ (a — r) $r,$ c $\leftarrow r[15], r$

c2/ (a — r) $r,$ c $\leftarrow c, r$

⟨*ALU operations*⟩≡
```
  5'b01001: { c, T } <= { 1'b1, ~T };
  5'b0111?: { T, c } <= { inst[0] ? c : T[15], T };
  5'b01???: { sp, c, T } <= { spinc, carry, res };
```

2.3 Memory Instructions

Memory instructions use either T as address, and N as data (source or destination), or P as address, and T as destination (literals). The address is auto-incremented, except for instructions in the first slot which use T as address—this is to implement read-modify-write instructions (non-incrementing is written as `@.` or `!.` in the assembler, don't care as `@*` or `!*`). The traditional Forth `@`, which consumes the address, is here, too. For `!`, this can't work, since we would have to drop two stack cells at once.

!+ (n A — A') *mem*[*A*] ← *n*; $A' \leftarrow A + 2$

@+ (A — n A') *n* ← *mem*[A]; $A' \leftarrow A + 2$

@ (A — n) *n* ← *mem*[A];

lit (— n) *n* ← *mem*[P]; $P \leftarrow P + 2$

c!+ (c A — A') *mem.b*[A] ← *c*; $A' \leftarrow A + 1$

c@+ (A — c A') *c* ← *mem.b*[A]; $A' \leftarrow A + 1$

c@ (A — c) *c* ← *mem.b*[A];

litc (— c) *c* ← *mem.b*[P]; $P \leftarrow P + 1$

⟨*address control*⟩≡
```
  wire `L incaddr, toR, R, dataw;
  wire tos2r, tos2n;
  wire incby, addrsel, access, rd;
  wire [1:0] wr;

  assign incby = (inst[4:2] != 3'b101);
  assign access = (inst[4:3]==2'b10);
  assign addrsel = rd ? (access & (~&inst[1:0]))
                      : |wr;
  assign rd = (state==2'b00) ||
              (access && (|inst[1:0]));
  assign wr = (access && (~|inst[1:0])) ?
              { ~inst[2] | ~T[0],
                ~inst[2] | T[0] } : 2'b00;
  assign tos2r = (inst == 5'b11100);
  assign tos2n = (!rd | (inst[1:0] == 2'b11));
```

⟨*address handling*⟩≡
```
  assign addr = addrsel ? T : P;
  assign incaddr =
      { addr[l-1:1] + (incby | addr[0]),
                      ~(incby | addr[0]) };
  assign toR = tos2r ? T :
               { |state ? P[15:1]
                        : incaddr[15:1], c });
  assign toN = tos2n ? T : dataw;
  assign dataw = incby ? data :
                 { 8'h00, addr[0] ? data[7:0]
                                  : data[l-1:8] };
  assign dataout = { incby ? N[15:8]
                           : N[7:0], N[7:0] };
```

Memory access can't just be done word wise, but also byte wise. Therefore two write lines exist. For byte wise store the lower byte of N is also put on the higher byte bus part. Byte accesses are quite useful, since most controlled values (like DACs and ADCs) have at most 8 bits.

⟨*load/store*⟩≡
```
  5'b10?0?: begin
     if(nextstate != 2'b10) T <= incaddr;
     sp <= rd ? spdec : spinc;
  end
  5'b10?1?: T <= dataw;
```

In the original b16, memory accesses needed a separate cycle. It turned out that this was a waste of cycles, which is only reasonable if the memory is slower than the clock. And in this case, you better insert a wait state. Here the result of the memory access is handled:

⟨*load-store*⟩≡
```
  ⟨pointer increment⟩
  if(|state[1:0]) begin
     ⟨store afterwork⟩
  end else begin
     ⟨ifetch⟩
  end
```

⟨*debug*⟩≡
```
  $write("%b[%b] T=%b%x:%x[%x], ",
         inst, state, c, T, N, sp);
  $write("P=%x, I=%x, R=%x[%x], res=%b%x\n",
         P, I, R, rp, carry, res);
```

After the access is completed, the result for a load has to be pushed on the stack, or into the instruction register; for stores, the N is to be dropped.

⟨*store afterwork*⟩≡
```
  if(rd && { inst[4:3], inst[1:0] } != 4'b1010)
     sp <= spdec;
  if(|wr) sp <= spinc;
```

Furthermore, the incremented address may go back to the pointer.

⟨*pointer increment*⟩≡
```
  if(~|state[1:0] ||
     ((inst[4:3] == 2'b10) && (&inst[1:0])))
     P <= incaddr;
```

To shortcut a `nop` in the first instruction, there's some special logic. That's the second part of NEXT.

⟨*ifetch*⟩≡
```
  I <= data;
  if(!data[15]) state[1:0] <= 2'b01;
```

2.4 Stack Instructions

Stack instructions change the stack pointer and move values into and out of latches. With the 8 used stack operations, one notes that `swap` is missing. Instead, there's `nip`. The reason is a possible implementation option: it's possible to omit N as register, and fetch this value directly out of the stack RAM. This consumes more time, but saves space. We actually have two unused instruction slots, so implementing N as register could give us `swap` and we could also have `r@`[1].

nip (a b — b)

drop (a —)

over (a b — a b a)

dup (a — a a)

>r (a — r:a)

r> (r:a — a)

⟨*stack operations*⟩≡
```
  5'b11000: sp <= spinc;
  5'b11001: `DROP;
  5'b11010: { sp, T } <= { spdec, N };
  5'b11011: sp <= spdec;
  5'b11100: begin
     rp <= rpdec; `DROP;
  end // case: 5'b11100
  5'b11110: begin
     { sp, T } <= { spdec, R };
     rp <= rpinc;
  end // case: 5'b11110
```

3 Tradeoffs of the Shrink

Ruthless redesign is what Forthers are used to do. What did we learn from this shrink?

1. It's smaller. The logic element that costs most area is the flip-flop. Ruthlessly eliminating flip-flops is a good strategy to make hardware small. The original b16 has 6 16 bit registers which have to be implemented as flip-flops: T, N, R, P, A, and I. Strictly speaking, the address latch belongs to this set, but it was a design mistake that can be removed without further effect. Now we have only three 16 bit registers which absolutely have to be flip-flops: T, I, and P. As I said, it would be nice to have N as flip-flop, too, so we really are on the edge. We also have fewer logic paths, mostly due to A going away, and some instructions less. We have have of the stack size. Overall, the reduction in size is almost 50%.
2. It's slower. That's due to two reasons. First, with N being part of the stack SRAM block, all ALU operations have the SRAM in their critical path. Second, with memory accesses taking only one cycle, they are also in the critical path. READY provides a wait state logic here, so it's not that awful.
3. It's closer to Forth:
   (a) CHUCK's ideas with additional pointer registers (A and R) look nice at first sight, but turned out to be not very useful in this project. Controlling often means just storing values to memory locations (memory mapped IO), and it's difficult to get these memory locations into an order where you can make use of auto-increment. The traditional Forth `@` and `c@`—which consume their address—do the job in one instruction, where the A-based addressing would need two. And with `@+`/`!+`, auto-increment still is there when you need it.
   (b) The control flow is also closer to Forth, where IF consumes the flag.

[1] It would be more consistent to name this word `R`, but normal Forth convention is `r@`.

3.1 I/O Strategy

Real controller hardware needs I/Os and other special functions. Most of the I/O here were registers that control analog components. To the digital side, these registers look like memory, and therefore don't need special care. The remaining special functions are a timer, and a power control register. The timer is a simple one-shot timer that increments a 16 bit number until it overflows. The power control register replaces interrupt handling. It contains four bits for different external events:

1. Reset—a hardware reset woke up the CPU.
2. Timer—a timer event woke up the CPU.
3. Communication—a write into RAM via the external serial interface woke up the CPU.
4. Analog event—a periodic event from the analog side woke up the CPU

Simply looking at these four bits allows to dispatch events. I use four `IF`s to check for the events, since using a branch table would waste more space. The application priorize events, so you know what to do first, anyway.

Unlike a real interrupt, no stack space is needed—the event loop is the main loop, anyway. There's no background task to do here, so interrupts don't make sense. If the CPU clears all run bits in the power control register, operation stops, until another event arrives. To prevent race conditions with other events, the power control register has a special write mode, where specified bits are cleared, and other bits remain as they were. A read-modify-write access to this register would take several cycles, and could lead to clearing a bit that was set in the time between read and write.

3.2 The Software

Initially, the customer didn't know what he wanted. That's a very common behavior. Not being a programmer, he tried to express the logic as "state machine". It turned out that this wasn't what he wanted. That was anyway the main reason to use a CPU: to implement the logic in software was anticipating the customer's uncertainty.

The program can be divided into two parts: initialization, and event loop. The event loop is a combination of the originally state machine, all the possible exceptions that could happen, but weren't foreseen, and asynchronous events that need constant reaction no matter what the current state is.

The customer's uncertainty resulted in two things: frequent rewrites (which didn't take much longer than to understand what the customer wanted now), and more uncertainty on the customer side. The whole logic was prototyped in an FPGA, so the customer could see the behavior. What he didn't see was the state machine diagram, and he couldn't find the logic in the code. That's because the original logic wasn't in the code, and couldn't.

The interesting thing was that the program stayed a small bit below 1k all the time. I had foreseen 1k ROM at the project start, but that wasn't a fixed value. The size of the ROM wasn't needed until the software was completely frozen, and the ROM mask was ordered. However, it did work out.

Subroutines are a natural way to save space. Another one is when you have constant arguments a lot, e.g. for the timeout setting. I wrote a timeout word, which takes the next word in the instruction stream as data:

```
: TIMER! ( -- )  r> @+ >r TIMER # ! ;
```

The thing to struggle most with was the stack depth. I've stripped down the stack to 5 elements (TOS plus four), and the return stack to 4 elements. The limiting word was the 3 out of 4 redundancy calculation. It took all the stack, and but one of the return stack elements (and like above, it takes inline arguments):

```
: 3of4 ( #:addr,addr -- )
    r> @+ >r
    @+ @+ @+ @+ >r xor >r xor r> and
    r> -6 # + @+ @+ @+ >r xor swap
    r> -8 # + @. >r xor and or
    r> @+ @+ >r over xor dup >r com and
    r> r> @ and or swap ?err
    r> @+ >r ! ;
```

4 The Rest of the Implementation

First the implementation file with comment and modules.

⟨b16.v⟩≡

```
/*
 * b16 core: 16 bits,
 * inspired by c18 core from Chuck Moore
 *
⟨inst-comment⟩
 */

`define L [l-1:0]
`define DROP { sp, T } <= { spinc, N }
`timescale 1ns / 1ns

⟨ALU⟩
⟨Stack⟩
⟨cpu⟩
```

⟨inst-comment⟩≡

```
 * Instruction set:
 * 1, 5, 5, 5 bits
 *     0    1    2    3    4    5    6    7
 *  0: nop  call jmp  ret  jz   jnz  jc   jnc
 *  /3      exec goto ret  gz   gnz  gc   gnc
 *  8: xor  com  and  or   +    +c   u2/  c2/
 * 10: !+   @+   @    lit  c!+  c@+  c@   litc
 *  /1 !.   @.   @    lit  c!.  c@.  c@   litc
 * 18: nip  drop over dup  >r        r>
```

4.1 Top Level

The CPU consists of several parts, which are all implemented in the same Verilog module.

⟨cpu⟩≡

```
module cpu(clk, run, reset, addr, rd, wr,
           data, dataout, READY);
   ⟨port declarations⟩
   ⟨register declarations⟩
   ⟨instruction selection⟩
   ⟨ALU instantiation⟩
   ⟨address control⟩
   ⟨address handling⟩
   ⟨stack pushs⟩
   ⟨stack instantiation⟩
   ⟨state changes⟩

   always @(posedge clk or negedge reset)
      ⟨register updates⟩

endmodule // cpu
```

First, Verilog needs port declarations, so that it can now what's input and output. The parameter are used to configure other word sizes and stack depths.

⟨port declarations⟩≡

```
parameter show=0, l=16, sdep=2, rdep=2;
input clk, run, reset, READY;
output `L addr;
output rd;
output [1:0] wr;
input  `L data;
output `L dataout;
```

The ALU is instantiated with the configured width, and the necessary wires are declared

⟨*ALU instantiation*⟩≡
```
wire ‘L res, toN, N;
wire carry, zero;

alu #(l) alu16(res, carry, zero,
               T, N, c, inst[2:0]);
```

Since the stacks work in parallel, we have to calculated, when a value is pushed onto the stack (thus **only** if something is stored there).

⟨*stack pushs*⟩≡
```
reg dpush, rpush;

always @(state or inst or rd or READY)
  begin
     rpush <= 1’b0;
     dpush <= |state[1:0] & rd;
     casez(inst)
        5’b00001: rpush <= |state[1:0] | READY;
        5’b11100: rpush <= 1’b1;
        5’b11?1?: dpush <= 1’b1;
     endcase // case(inst)
  end
```

The stacks don’t only consist of the two stack modules, but also need an incremented and decremented stack pointer. The return stack even allows to write the top of return stack even without changing the return stack depth.

⟨*stack instantiation*⟩≡
```
wire [sdep-1:0] spdec, spinc;
wire [rdep-1:0] rpdec, rpinc;

stack #(sdep,l) dstack(clk, sp, spdec,
                       dpush, toN, N);
stack #(rdep,l) rstack(clk, rp, rpdec,
                       rpush, toR, R);

assign spdec = sp-{{(sdep-1){1’b0}}, 1’b1};
assign spinc = sp+{{(sdep-1){1’b0}}, 1’b1};
assign rpdec = rp-{{(rdep-1){1’b0}}, 1’b1};
assign rpinc = rp+{{(rdep-1){1’b0}}, 1’b1};
```

The basic core is the fully synchronous register update. Each register needs a reset value, and depending on the state transition, the corresponding assignments have to be coded. Most of that is from above, only the instruction fetch and the assignment of the next value of `incby` has to be done.

⟨*register updates*⟩≡
```
if(!reset) begin
   ⟨resets⟩
end else if(run) begin
   if(show) begin
      ⟨debug⟩
   end
   ⟨load-store⟩
   state <= nextstate;
   ⟨instructions⟩
end // else: !if(reset)
```

As reset value, we initialize the CPU so that it is about to fetch the next instruction from address 0. The stacks are all empty, the registers contain all zeros.

⟨*resets*⟩≡
```
state <= 2’b11;
P <= 16’h07FE;
T <= 16’h0000;
I <= 16’h0000;
c <= 1’b0;
sp <= 0;
rp <= 0;
```

The transition to the next state (the NEXT within a bundle) is done separately. That’s necessary, since the assignments of the other variables are not just dependent on the current state, but partially also on the next state (e.g. when to fetch the next instruction word).

⟨*state changes*⟩≡
```
reg [1:0] nextstate;

always @(inst or state)
   casez(inst)
      ⟨inst-nextstate⟩
   endcase // casez(inst[0:2])
```

⟨*inst-nextstate*⟩≡
```
5’b00000: nextstate <= state[1:0] + 2’b01;
5’b00???: nextstate <= 2’b00;
5’b?????: nextstate <= state[1:0] + 2’b01;
```

4.2 ALU

The ALU just computes the sum with possible carry-ins, the logical operations, and a zero flag. It would be possible to share common resources (the XORs of the full adder could also compute the XOR operation, and the carry propagation logic could compute OR and AND), but this optimization is left to the synthesis tool.

⟨*ALU*⟩≡
```
module alu(res, carry, zero, T, N, c, inst);
   ⟨ALU ports⟩

   wire       ‘L r1, r2;
   wire [l:0]  carries;

   assign #1 r1 = T ^ N ^ carries;
   assign #1 r2 = (T & N) |
                  (T & carries‘L) |
                  (N & carries‘L);
   assign #1 carries =
      prop ? { r2[l-1:0], (c | selr) & andor }
           : { c, {(l){andor}}};
```

```
    assign #1 res = (selr & ~prop) ? r2 : r1;
    assign #1 carry = carries[l];
    assign #1 zero = ~|T;
endmodule // alu
```

The ALU has ports T and N, carry in, and the lowest 3 bits of the instruction as input, a result, carry out, and test for zero as output.

⟨*ALU ports*⟩≡

```
parameter l=16;
input `L T, N;
input c;
input [2:0] inst;
output `L res;
output carry, zero;

wire prop, andor, selr;

assign #1 { prop, selr, andor } = inst;
```

4.3 Stacks

The stacks are modeled as block RAM in the FPGA. Therefore, they should have only one port, since these block RAMs are available even in small FPGAs. In an ASIC, this sort of stack is implemented with latches. Here it's possible to separate read and write port (also for FPGAs that support dual-ported RAM), and save the multiplexer for spset.

⟨*Stack*⟩≡

```
module stack(clk, sp, spdec, push, in, out);
    parameter dep=2, l=16;
    input clk, push;
    input [dep-1:0] sp, spdec;
    input `L in;
    output `L out;
`ifdef SYNCSTACK
// on FPGAs, SYNCSTACK is preferred
    reg `L stackmem[0:(1<<dep)-1];
    always @(posedge clk)
      if(push)
        stackmem[spdec] <= in;

    assign out = stackmen[sp];
`else
    wire [dep-1:0] #1 spset = spdec;
    wire #1 write = push & ~clk;
    wire `L #1 ind = in;
    reg `L stackmem[0:(1<<dep)-1];

    always @(write or spset or ind)
      if(write) stackmem[spset] <= ind;

    assign #1 out = stackmem[sp];
`endif
endmodule // stack
```

References

[1] *c18 ColorForth Compiler,* CHUCK MOORE, 17th EuroForth Conference Proceedings, 2001