b16 — A Forth Processor in an FPGA

Bernd Paysan

July 9, 2006

Abstract

This article presents architecture and implementation of the b16 stack processor. This processor is inspired by Chuck Moore's newest Forth processors. The minimalistic design fits into small FPGAs and ASICs and is ideally suited for applications that need both control and calculations. The synthesizible implementation uses Verilog.

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. I traded `2*` and `2/` against division step and Forth-typical logic operations; these two instructions can be implemented as short macro. Also, this architecture is byte-addressed.

The original concept (which was synthesizible, and could execute a small sample program) was written in an afternoon. The current version is somewhat faster, and really runs on a Altera Flex10K30E on a FPGA evaluation board from Hans Eckes. Size and speed of the processor can be evaluated.

Flex10K30E About 600 LCs, the unit for logic cells in Altera[1]. The logic to interface with the eval board needs another 100 LCs. The slowest model runs at up to 25MHz.

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.

[1] A logic cell can compute a logic function with four inputs and one output, or a full-adder, and also contains a flip-flop.

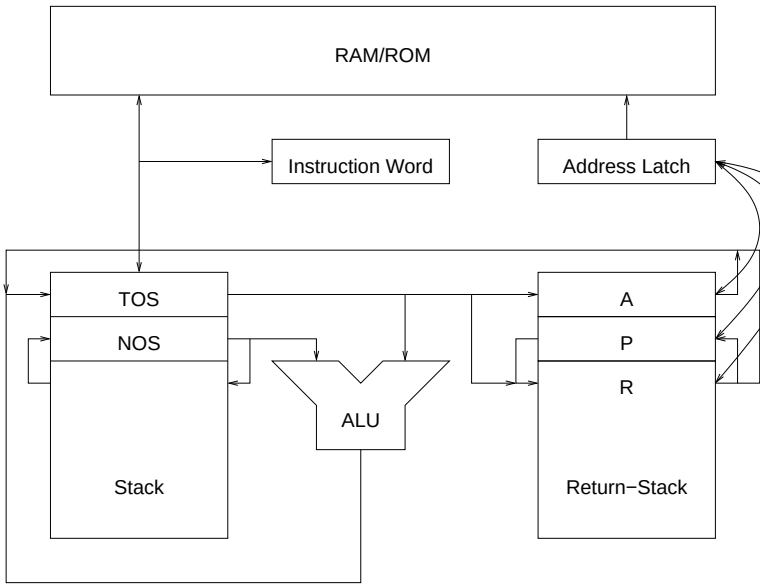

Figure 1: Block Diagram

1 Architectural Overview

The core components are

- An ALU
- A data stack with top and next of stack (T and N) as inp for the ALU
- A return stack, where the top of return stack (R) can used as address
- An instruction pointer P
- An address register A
- An address latch `addr`, to address external memory
- An instruction latch I

Figure 1 shows a block diagram.

1.1 Register

In addition to the user-visible latches there are control latches external RAM (`rd` and `wr`), stack pointers (`sp` and `rp`), a carr and the flag `incby`, by which `addr` is incremented.

Name	*Function*
T	Top of Stack
N	Next of Stack
I	Instruction Bundle
P	Program Counter
A	Address Register
addr	Address Latch
state	Processor State
sp	Stack Pointer
rp	Return Stack Pointer
c	Carry Flag
incby	Increment Address by byte/word

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

reg `L T, N, I, P, A, addr;

reg [2:0] state;
reg c;
reg incby;
reg intack;
```

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.

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)
   case(state[1:0])
     2'b00: inst <= { 4'b0000, I[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)
   case(state[1:0])
     2'b00: jmp <= { I[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 };
   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)
```

2.1 Jumps

In detail, jumps are performed as follows: the target addres stored in the address latch `addr`, which addresses memory, no the P register. The register P will be set to the incremented v of `addr`, after the instruction fetch cycle. Apart from `call`, and `ret` there are conditional jumps, which test for 0 and ca The lowest bit of the return stack is used to save the carry across calls. Conditional instructions don't consume the te value, which is different from Forth.

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

nop (—)

call (— r:P) P ← *jmp*; c ← 0

jmp (—) P ← *jmp*

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

jz (n — n) **if**$(n = 0)$ P ← *jmp*

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

jc (—) **if**(c) P ← *jmp*

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

Table 1: Instruction Set

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

⟨*control flow*⟩≡

```
5'b00001: begin
   rp <= rpdec;
   addr <= jmp;
   c <= 1'b0;
   if(state == 3'b011) `DROP;
end // case: 5'b00001
5'b00010: begin
   addr <= jmp;
   if(state == 3'b011) `DROP;
end
5'b00011: begin
   { c, addr } <= { R[0], R[l-1:1], 1'b0 };
   rp <= rpinc;
end // case: 5'b01111
5'b001??: begin
   if((inst[1] ? c : zero) ^ inst[0])
      addr <= jmp;
   if(state == 3'b011) `DROP;
end
```

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.

The two instructions `*+` (multiplication step) and `/-` (division step) shift the result into the A register and carry bit. `*+` adds N to T, when the carry bit is set, and shifts the result one step right.

`/-` also adds N to T, but also tests, if there is an overflow, or if the old carry was set. The result is shifted one to the left.

Ordinary ALU instructions just write the result of the ALU into T and c, and reload N.

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

***+** (a b — a r) **if**(c) $c_n, r \leftarrow a + b$ **else** $c_n, r \leftarrow 0, b$; r, A, c_n, r, A

/– (a b — a r) $c_n, r_n \leftarrow a + b + 1$; **if**(c $\vee\, c_n$) $r \leftarrow r_n$; c, r, A, r, A, c $\vee\, c_n$

⟨*ALU operations*⟩≡

```
5'b01001: { c, T } <= { 1'b1, ~T };
5'b01110: { T, A, c } <=
   { c ? { carry, res } : { 1'b0, T }, A };
5'b01111: { c, T, A } <=
   { (c | carry) ? res : T, A, (c | carry) };
5'b01???: begin
   c <= carry;
   { sp, T, N } <= { spinc, res, toN };
end // case: 5'b01???
```

2.3 Memory Instructions

CHUCK MOORE doesn't use the TOS as address any more, has introduced the A register. When you want to copy mem areas, you need a second address register, that's what he the top of return stack R for. Since P has to be incremer after each instruction fetch (to point to the next instruction), address logic must have auto increment. This will also be u for other accesses.

Memory instructions which use the first slot, and don't index P, don't increment the pointer; that's to realize read-modify-w instructions like `+!`. Write access is only possible via A, the other pointers can only be used for read access.

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

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

R@+ (— n) $n \leftarrow mem$[R]; R $\leftarrow$ R + 2

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

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

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

Rc@+ (— c) $c \leftarrow mem.b$[R]; R $\leftarrow$ R + 1

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

⟨*address handling*⟩≡
```
wire `L toaddr, incaddr, toR, R;
wire tos2r;

assign toaddr = inst[1] ? (inst[0] ? P : R) : A;
assign incaddr =
    { addr[l-1:1] + (incby | addr[0]),
                  ~(incby | addr[0]) };
assign tos2r = inst == 5'b11100;
assign toR = state[2] ? incaddr :
             (tos2r ? T : { P[15:1], c });
```

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 T is copied to the higher one.

⟨*load/store*⟩≡
```
5'b10000: begin
   addr <= toaddr;
   wr <= 2'b11;
end
5'b10100: begin
   addr <= toaddr;
   wr <= { ~toaddr[0], toaddr[0] };
   T <= { T[7:0], T[7:0] };
end
5'b10???: begin
   addr <= toaddr;
   rd <= 1'b1;
end
```

Memory accesses need an extra cycle. Here the result of the memory access is handled.

⟨*load-store*⟩≡
```
if(show) begin
   ⟨debug⟩
end
state <= nextstate;
⟨pointer increment⟩
rd <= 1'b0;
wr <= 2'b0;
if(|state[1:0]) begin
   ⟨store afterwork⟩
end else begin
   ⟨ifetch⟩
end
⟨next⟩
```

There's a special case for the instruction fetch (the NEXT of machine): when the current instruction is a literal, we must incaddr instead of P.

⟨*next*⟩≡
```
if(nextstate == 3'b100) begin
    { addr, rd } <= { &inst[1:0] ?
                      incaddr : P, 1'b1 };
end // if (nextstate == 3'b100)
```

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

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

⟨*store afterwork*⟩≡
```
if(rd)
   if(incby)
      { sp, T, N } <= { spdec, data, T };
   else
      { sp, T, N } <= { spdec, 8'h00,
        addr[0] ? data[7:0] : data[l-1:8], T };
if(|wr)
   `DROP;
incby <= 1'b1;
```

Furthermore, the incremented address may go back to the poin

⟨*pointer increment*⟩≡
```
casez({ state[1:0], inst[1:0] })
   4'b00??: P <= !intreq ? incaddr : addr;
   4'b1?0?: A <= incaddr;
// 4'b1?10: R <= incaddr;
   4'b??11: P <= incaddr;
endcase // casez({ state[1:0], inst[1:0] })
```

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

⟨*ifetch*⟩≡
```
intack <= intreq;
if(intreq)
  I <= { 8'h81, intvec }; // call $200+intvec*2
else
  I <= data;
if(!intreq & !data[15]) state[1:0] <= 2'b01;
```

Here, we also handle interrupts. Interrupts are accepted at instruction fetch. Instead of incrementing P, we load a call to the interrupt vector (addresses from $200) into the instruction register. The interrupt routine just has to save A (if needed), and has to balance the stack on return. Since three instructions can be executed without interrupt, there's no interrupt disable flag internally, only an external interrupt unit might do that. The last three instructions of such an interrupt routine then would be `a! >a ret`.

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, and fetch this value directly out of the stack RAM. This consumes more time, but saves space.

Also, CHUCK MOORE claims, that you don't need `swap` — if you don't have it, you help out with other stack operation, and there's nothing to do, there's still `>a >r a r>`.

nip (a b — b)
drop (a —)
over (a b — a b a)
dup (a — a a)
>r (a — r:a)
>a (a —) A ← *a*
r> (r:a — a)
a (— a) *a* ← A

⟨*stack operations*⟩≡
```
5'b11000: { sp, N } <= { spinc, toN };
5'b11001: 'DROP;
5'b11010: { sp, T, N } <= { spdec, N, T };
5'b11011: { sp, N } <= { spdec, T };
5'b11100: begin
   rp <= rpdec; 'DROP;
end // case: 5'b11100
5'b11101: begin
   A <= T; 'DROP;
end // case: 5'b11101
5'b11110: begin
   { sp, T, N } <= { spdec, R, T };
   rp <= rpinc;
end // case: 5'b11110
5'b11111:  { sp, T, N } <= { spdec, A, T };
```

If you don't want to live without `swap`, you can replace the implementation of `nip` in the first line by:

⟨*swap*⟩≡
```
5'b11000: { T, N } <= { N, T };
```

3 Examples

A few examples show, how to program this processor. M
plication works through the A register. There's one extra
necessary, since each bit first has to be shifted into the c
register. Since `call` clears carry, we don't have to do that he

⟨*mul*⟩≡
```
: mul ( u1 u2 - ud )
  >A 0 #
  *+ *+ *+  *+ *+ *+  *+ *+ *+
  *+ *+ *+  *+ *+ *+  *+ *+
  >r drop a r> ;
```

Division needs an extra step, too. Here, we need a real `swap`, since there is none, we first use `over` and accept that we hav use one extra stack item. Other than with `mul` we here nee clear the carry after `com`. And finally, we have to divide by and shift in the carry.

⟨*div*⟩≡
```
: div ( ud udiv - uqout umod )
  com >r >r >a r> r> over 0 # +
  /- /- /-  /- /- /-  /- /- /-
  /- /- /-  /- /- /-  /- /-
  nip nip a >r -cIF *+ r> ;
  THEN 0 # + *+ $8000 # + r> ;
```

The next example is even more complicated, since I emula serial interface. At 10MHz, each bit takes 87 clock cycles, to a 115200 baud fast serial line. We add a second stop bit, to a the other side to resynchronize, when the next bit arrives.

⟨*serial line*⟩≡
```
: send-rest ( c - c' ) *+
: wait-bit
  1 # $FFF9 # BEGIN  over +  cUNTIL  drop drop ;
: send-bit ( c - c' )
  nop \ delay at start
: send-bit-fast ( c - c' )
  $FFFE # >a dup 1 # and
  IF    drop $0001 # a@ or  a!+ send-rest ;
  THEN  drop $FFFE # a@ and a!+ send-rest ;
: emit ( c - ) \ 8N1, 115200 baud
  >r 06 # send-bit r>
  send-bit-fast send-bit send-bit send-bit
  send-bit send-bit send-bit send-bit
  drop send-bit-fast send-bit drop ;
```

Like in ColorForth, `;` is just an EXIT, and `:` is used as labe there's a call before `;`, this is converted to a jump. This s return stack entries, time, and code space.

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, N } <= { spinc, N, toN }
‘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   *+   /-
 * 10: A!+  A@+  R@+  lit  Ac!+ Ac@+ Rc@+ litc
 *  /1 A!   A@   R@   lit  Ac!  Ac@  Rc@  litc
 * 18: nip  drop over dup  >r   >a   r>   a
```

4.1 Top Level

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

⟨*cpu*⟩≡
```
module cpu(clk, reset, addr, rd, wr, data, T,
           intreq, intack, intvec);
   ⟨port declarations⟩
   ⟨register declarations⟩
   ⟨instruction selection⟩
   ⟨ALU instantiation⟩
   ⟨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 wh input and output. The parameter are used to configure o word sizes and stack depths.

⟨*port declarations*⟩≡
```
parameter show=0, l=16, sdep=3, rdep=3;
input clk, reset;
output ‘L addr;
output rd;
output [1:0] wr;
input  ‘L data;
output ‘L T;
input  intreq;
output intack;
input [7:0] intvec; // interrupt jump vector
```

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

⟨*ALU instantiation*⟩≡
```
wire ‘L res, toN;
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, wh value is pushed onto the stack (thus **only** if something is st there).

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

always @(clk or state or inst or rd)
  begin
     dpush <= 1'b0;
     rpush <= 1'b0;
     if(state[2]) begin
        dpush <= |state[1:0] & rd;
        rpush <= state[1] & (inst[1:0]==2'b10);
     end else
        casez(inst)
          5'b00001: rpush <= 1'b1;
          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 need an incremented and decremented stack pointer. The re stack even allows to write the top of return stack even wit changing the return stack depth.

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

stack #(sdep,1) dstack(clk, sp, spdec,
                       dpush, N, toN);
stack #(rdep,1) 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){(~state[2] | tos2r)}};
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(state[2]) begin
   ⟨load-store⟩
end else begin // if (state[2])
   if(show) begin
      ⟨debug⟩
   end
   if(nextstate == 3'b100)
      { addr, rd } <= { P, 1'b1 };
   state <= nextstate;
   incby <= (inst[4:2] != 3'b101);
   ⟨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 <= 3'b011;
incby <= 1'b0;
P <= 16'h0000;
addr <= 16'h0000;
A <= 16'h0000;
T <= 16'h0000;
N <= 16'h0000;
I <= 16'h0000;
c <= 1'b0;
rd <= 1'b0;
wr <= 2'b00;
sp <= 0;
rp <= 0;
intack <= 0;
```

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

⟨*state changes*⟩≡
```
reg [2:0] nextstate;

always @(inst or state)
   if(state[2]) begin
      ⟨rw-nextstate⟩
   end else begin
      casez(inst)
         ⟨inst-nextstate⟩
      endcase // casez(inst[0:2])
   end // else: !if(state[2]) end
```

⟨*rw-nextstate*⟩≡
```
nextstate <= state[1:0] + { 2'b0, |state[1:0] };
```

⟨*inst-nextstate*⟩≡
```
5'b00000: nextstate <= state[1:0] + 3'b001;
5'b00???: nextstate <= 3'b100;
5'b10???: nextstate <= { 1'b1, state[1:0] };
5'b?????: nextstate <= state[1:0] + 3'b001;
```

4.2 ALU

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

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

   wire        `L sum, logic;
   wire        cout;

   assign { cout, sum } =
          T + N + ((c | andor) & selr);
   assign logic = andor ?
                  (selr ? (T | N) : (T & N)) :
                  T ^ N;
   assign { carry, res } =
          prop ? { cout, sum } : { c, logic };
   assign 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, andor, selr } = 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=3, l=16;
   input clk, push;
   input [dep-1:0] sp, spdec;
   input `L in;
   output `L out;
   reg `L stackmem[0:(1<<dep)-1];
   wire [dep-1:0]  spset;

`ifdef BEH_STACK
   always @(clk or push or spset or in)
     if(push & ~clk) stackmem[spset] <= #1 in;

   assign spset = push ? spdec : sp;
   assign #1 out = stackmem[spset];
`else
   stackram stram(in, push, spdec, sp, ~clk, out);
`endif
endmodule // stack
```

4.4 Further Possible Optimizations

It would be possible to overlap memory accesses and operat on the stack, since there are separate pointer registers. The derstandability of the code would suffer, and the critical p would also be somewhat longer. With a guaranteed speed crease of 25% (the cycle to fetch instructions would vanish), a maximum acceleration by 100% (for memory-intensive appl tions), this could be worth the trouble — when there's eno space.

If there's lack of space, it is possible to implement most regis as latches. Only T needs to be a real flip-flop. For FPGAs, is not an option, flip-flops are cheaper there.

4.5 Scaling Issues

Two approaches allow to adopt the b16 to own preferences: w width and stack depth. The stack depth is easier. The cho depth of 8 is sufficient for the boot loader, but could cause p lems for more complex applications. Simpler applications how should fit with a smaller stack.

The word width can be adopted for the application, too. A sion reduced to 12 bit (and also with a modified instruction is used in a project at my employer Mikron AG. This requ to change the decoding of the instructions within the slot, adopt the logic to step over the first `nop`.

Furthermore, you can replace individual instructions. For th bit version, it was found that bit operations occur very frequen and byte accesses are completely irrelevant.

5 Development Environment

I could present a longer listing here, this time in Forth. Howe I'll just describe the functions. All three programs are put one file, and allow interactive use of simulator and target.

5.1 Assembler

The assembler resembles a bit CHUCK MOORE's ColorFo There are no colors, just normal punctation, as common in fo The assembler after all is coded in Forth, and therefore exp Forth tokens.

Labels are defined with `:` and `|`. The first one automatically call on reference, but can be put on stack with '. The last one more resemble an interactive `Create`. Labels are only resolved backwards. Literals must be taken from the stack explicitely with `#` or `#c`. The assembler takes care of the ordering within the slots. A `ret` is normally compiled with a `;`, preceeding calls are converted to a `jmp`. You can define macros (`macro: ... end-macro`).

Also the well-known control structures from Forth can be used (must be used for forward branches). `IF` becomes a `jz`, `jnz` is reached with `-IF`. `cIF` and `-cIF` correspond `jnc` and `jc`. Similar prefixes are available for `WHILE` and `UNTIL`.

5.2 Downloader

A piece of block RAM in the FPGA is occupied by a small gram, the boot loader. This small program drives the LE and waits for commands from the serial line (115.2KB-aud, 8 no handshake). There are three commands, starting with AS signs:

0 *addr, len, <len∗data>:* Programs memory from *addr* with data bytes

1 *addr, len:* Reads back *len* bytes from memory starting at *a*

2 *addr:* Execute the word at *addr*

These three commands are sufficient to program the b16 inte tively. On the host side, a few instructions are sufficient, too

comp Compile to the end of line, and send the result to evaluation board

eval Compile to the end of line, send the result to the eva tion board, call the code, and set the RAM pointer of assembler back to the original value

sim Same as `eval`, but execute the result with the simul instead of using the evaluation board

check (addr u —) Reads a memory block from the evalua board, and display it with `dump`

6 Outlook

More material is available from my home page [2]. All sou are available under GPL. Data for producing a board is availa too. HANS ECKES might make one for you, if you pay for it. if someone wants to use the b16 commercially, talk to me.

References

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

[2] *b16 Processor,* BERND PAYSAN, Internet H page, http://www.jwdt.com/ paysan/b16.h http://www.jwdt.com/~paysan/b16.html