b16 Documentation

Bernd Paysan

February 25, 2011

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 factor is shifted towards control to save space. 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; it also differs from the standard b16 core. Also, this architecture is byte-addressed.

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 Architectural Overview

The core components are

- 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 a block diagram.

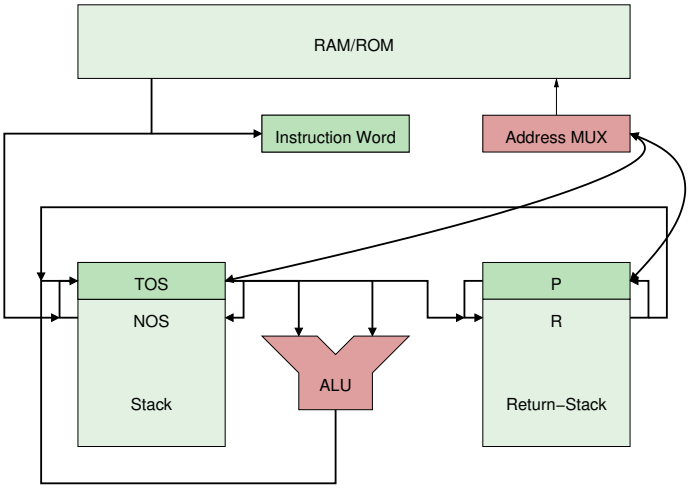

Figure 1: Block Diagram

1.1 Register

In addition to the standard Forth machine registers there are control registers for external RAM (`rd` and `wr`), stack pointers (`sp` and `rp`), and a carry `c`. For consistency with Chuck Moores' nomenclature, violating most coding style guidelines, the Forth machine registers are single-letter variables in upper case. Since the source code is a LyX document, you can use the "search whole word" mode to find them easily, and they also show up on top of the signal list during simulation.

Name	*Function*
T	Top of Stack
I	Instruction Bundle
P	Program Counter
R	Top of Returnstack
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, R;
reg [1:0] state;
reg c;
```

	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	!+ !.	@+ @.	@ @	lit lit	c!+ c!.	c@+ c@.	c@ c@	litc litc	 *for slot 1*
18	nip	drop	over	dup	>r		r>		

Table 1: Instruction Set

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"

@ "load",

> "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
wire [4:0] inst, rwinst;
reg ‘L jmp;

assign inst = { 4’b0000, data[15], I[14:0] }
              >> (5*(3-state[1:0]));
assign rwinst = { 5’b00000, I[14:0] }
                >> (5*(3-state[1:0]));

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 };
   endcase // casez(state)
```

The instructions themselves are executed depending on `inst`:

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

2.1 Jumps

In detail, jumps are performed as follows: the target address is stored in the address latch `addr`, which addresses memory, not in the P register. The register P will be set to the incremented value of `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 don't consume the tested value, which is different from Forth.

To make it easier to understand, I also define the effect of an instruction in a pseudo language. Every instruction has a stack effect (before—after) with top of stack on the right, "r:" prefix indicating return stack, and register assignments:

nop (—)

call (—r:P) P ← *jmp*; c ← 0

jmp (—) P ← *jmp*

ret (r:a—) P ← *a* ∧ $FFFE; c ← *a* ∧ 1

jz (n—) **if**(*n* = 0) P ← *jmp*

jnz (n—) **if**(*n* ≠ 0) P ← *jmp*

jc (x—) **if**(*c*) P ← *jmp*

jnc (x—) **if**(*c* = 0) P ← *jmp*

⟨*control flow*⟩≡
```
5'b00001: begin // call
   rp <= rpdec;
   R <= { ~|state ? incaddr[15:1] : P[15:1], c };
   P <= jmp;
   c <= 1'b0;
   if(state == 2'b11) `DROP;
end // case: 5'b00001
5'b00010: begin // jmp
   P <= jmp;
   if(state == 2'b11) `DROP;
end
5'b00011: // ret
          { rp, c, P, R } <=
          { rpinc, R[0], R[1-1:1], 1'b0, toR };
5'b00100, 5'b00101, 5'b00110, 5'b00111:
begin // conditional jmps
   if((inst[1] ? c : zero) ^ inst[0])
      P <= jmp;
   `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.

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

xor (a b—r) *r* ← *a* ⊕ *b*

com (a—r) *r* ← *a* ⊕ $FFFF, c ← 1

and (a b—r) *r* ← *a* ∧ *b*

or (a b—r) *r* ← *a* ∨ *b*

+ (a b—r) c, *r* ← *a* + *b*

+c (a b—r) c, *r* ← *a* + *b* + c

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

/− (a b—a r) $c_n, r_n \leftarrow a + b + 1$; **if**(c ∨ c_n) $r \leftarrow r_n$; c, *r*, R ← *r*, R, c ∨ c_n

⟨*ALU operations*⟩≡
```
5'b01001: // com
   { c, T } <= { 1'b1, ~T };
5'b01110: // *+
   { T, R, c } <=
   { c ? { carry, res } : { 1'b0, T }, R };
5'b01111: // /-
   { c, T, R } <=
   { (c | carry) ? res : T, R, (c | carry) };
5'b01000, 5'b01010, 5'b01011, 5'b01100, 5'b01101:
   // xor, and, or, +, +c
   { 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-incremeting is written as @. or !. in the assembler, don't care as @* or !*).

!+ (n A—A') *mem*[*A*] ← *n*; A′ ← A + 2

@+ (A—n A') *n* ← *mem*[A]; A′ ← A + 2

@ (A—n) *n* ← *mem*[A];

lit (—n) *n* ← *mem*[P]; P ← P + 2

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

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

c@ (A—c) *c* ← *mem.b*[A];

litc (—c) *c* ← *mem.b*[P]; P ← P + 1

⟨*address handling*⟩≡
```
wire ‘L incaddr, dataw, datas;
wire tos2r, tos2n;
wire incby, bswap, addrsel, access, rd;
wire [1:0] wr;

assign incby = (rwinst[4:2] != 3’b101);
assign access = (rwinst[4:3]==2’b10);
assign addrsel = rd ?
      (access & (rwinst[1:0] != 2’b11)) : |wr;
assign rd = (state==2’b00) ||
            (access && (rwinst[1:0]!=2’b00));
assign wr = (access && (rwinst[1:0]==2’b00)) ?
            { ~rwinst[2] | ~T[0],
              ~rwinst[2] | T[0] } : 2’b00;
assign addr = addrsel ? T : P;
assign incaddr = addr + incby + 1;
assign tos2n = (!rd | (rwinst[1:0] == 2’b11));
assign toN = tos2n ? T : dataw;
assign bswap = ~incby ^ addr[0];
assign datas = bswap ? { data[7:0], data[l-1:8] }
                     : data;
assign dataw = incby ? datas
                     : { 8’h00, datas[7:0] };
assign dataout = bswap ? { N[7:0], N[l-1:8] }
                       : N;
```

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, 5’b10001, 5’b10100, 5’b10101:
begin        // !+, @+, c!+, c@+
   if(nextstate != 2’b10) T <= incaddr;
   sp <= rd ? spdec : spinc;
end
5’b10010, 5’b10011, 5’b10110, 5’b10111:
   T <= dataw;  // @, lit, c@, litc
```

Memory accesses need an extra cycle. 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 TOS 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 program pointer.

⟨*pointer increment*⟩≡
```
if(~|state ||
   ({ inst[4:3], inst[1:0] } == 4’b1011))
   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;
```

2.3.1 Peripherals

Peripherals should only use address bits [15:1], read a whole word, and select the bytes written to based on the two write bits (bit 1 for most significant byte, bit 0 for least significant byte).

2.4 Stack Instructions

Stack instructions change the stack pointer and move values into and out of latches. With the 6 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.

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;                // nip
5’b11001: ‘DROP;                      // drop
5’b11010: { sp, T } <= { spdec, N }; // over
5’b11011: sp <= spdec;                // dup
5’b11100: begin                       // >r
   R <= T; rp <= rpdec; ‘DROP;
end // case: 5’b11100
5’b11110: begin                       // r>
   { sp, T, R } <= { spdec, R, toR };
   rp <= rpinc;
end // case: 5’b11110
default ;                             // noop
```

3 The Rest of the Implementation

First the implementation file(s) with comment and modules. You can either have all in one file (`b16.v`), or each module in a file with the same name as the module—the defines will go to `b16-defines.v` for central manipulation of the defines.

⟨*header*⟩≡
```
/*
 * b16 core: 16 bits,
 * inspired by c18 core from Chuck Moore
 * (c) 2002-2011 by Bernd Paysan
 *
 * ⟨gpl-header⟩
 */
```

⟨*defines*⟩≡
```
`define L [1-1:0]
`define DROP { sp, T } <= { spinc, N }
`define DEBUGGING
`define FPGA
// `define BUSTRI
```

⟨*b16.v*⟩≡
```
⟨header⟩
/*
⟨inst-comment⟩
 */
⟨defines⟩

⟨ALU⟩
⟨Stack⟩
⟨cpu⟩
⟨debugger⟩
```

⟨*b16-defines.v*⟩≡
```
⟨defines⟩
```

⟨*alu.v*⟩≡
```
⟨header⟩
`include "b16-defines.v"

⟨ALU⟩
```

⟨*stack.v*⟩≡
```
⟨header⟩
`include "b16-defines.v"

⟨Stack⟩
```

⟨*cpu.v*⟩≡
```
⟨header⟩
/*
⟨inst-comment⟩
 */
`include "b16-defines.v"

⟨cpu⟩
```

⟨*debugger.v*⟩≡
```
⟨header⟩
`include "b16-defines.v"

⟨debugger⟩
```

⟨*gpl-header*⟩≡
```
This program is free software; you can redistribute it
it under the terms of the GNU General Public License a
the Free Software Foundation; version 2 of the License

This program is distributed in the hope that it will b
but WITHOUT ANY WARRANTY; without even the implied war
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
GNU General Public License for more details.

This is not the source code of the program, the source
literate programming style article.
```

⟨*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: !+   @+   @    lit  c!+  c@+  c@   litc
 *  /1 !.   @.   @    lit  c!.  c@.  c@   litc
 * 18: nip  drop over dup  >r        r>
```

3.1 Top Level

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

⟨*cpu*⟩≡
```
module cpu(clk, latclk, run, nreset, addr, rd, wr, dat
           dataout, gwrite
`ifdef DEBUGGING,
           dr, dw, daddr, din, dout, bp`endif);
   ⟨port declarations⟩
   ⟨register declarations⟩
   ⟨instruction selection⟩
   ⟨ALU instantiation⟩
   ⟨address handling⟩
   ⟨stack pushs⟩
   ⟨stack instantiation⟩
   ⟨state changes⟩
   ⟨debugging read⟩

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

endmodule // cpu
```

First, Verilog needs port declarations, so that it can know what's input and output. The parameter are used to configure other word sizes and stack depths. The CPU is not fully scalable, e.g. the instruction decoder or the byte swap operation for byte access depends on 16 bit word size, but those parts of the CPU that are scalable can be scaled by changing that parameter—the others need manual intervention.

⟨*port declarations*⟩≡
```
parameter rstaddr=16'h3FFE, show=0,
          l=16, sdep=4, rdep=4;
input clk, latclk, run, nreset, gwrite;
output `L addr;
output rd;
output [1:0] wr;
input  `L data;
output `L dataout;
⟨debugging-ports⟩
```

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

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

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

Since the stacks work in parallel, we have to calculate 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 run ⟨dbg senselist⟩)
  begin
     rpush = 1'b0;
     dpush = (|state[1:0] & rd) |
             (inst[4] && inst[3] && inst[1]);
     case(inst)
        5'b00001: rpush = |state[1:0] | run;
        5'b11100: rpush = 1'b1;
        default ;
     endcase // case(inst)
     ⟨stack debugging⟩
  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(latclk),
                       .sp(sp),
                       .spdec(spdec),
                       .push(dpush),
                       .in(toN),
                       .out(N),
                       .gwrite(gwrite));
stack #(rdep,l) rstack(.clk(latclk),
                       .sp(rp),
                       .spdec(rpdec),
                       .push(rpush),
                       .in(R),
                       .out(toR),
                       .gwrite(gwrite));

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(!nreset) begin
   ⟨resets⟩
end else if(run) begin
`ifdef REPORT_VERBOSE
   if(show) begin
      ⟨debug⟩
   end
`endif
   ⟨load-store⟩
   state <= nextstate;
   ⟨instructions⟩
end else begin // debug
   ⟨debugging⟩
end // else: !if(nreset)
```

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 <= rstaddr;
T <= 16'h0000;
I <= 16'h0000;
R <= 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*⟩≡
```
wire [1:0] nextstate;

assign nextstate = ((~|inst) || (|inst[4:3])) ?
                   state[1:0] + 2'b01 : 2'b00;
```

3.2 Debugging

For debugging purposes, all registers are memory read–writable. This requires an external bus master attached to the debugging interface. The debugging interface is configured with the DEBUGGING flag. 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	stack[sp++]	push+T
$FFE2	rstack[rp++]	rpush+R
$FFE4	bp	bp
$FFE6	state+stop	state
$FFE8	P	P
$FFEA	T	T
$FFEC	R	R
$FFEE	I	I

The stacks and the state register change state when being read, so be careful!

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

reg drun, drun1;
reg `L bp;
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;
   bp <= 16'hffff;
end else begin
   if(cpu_addr == bp && cpu_r)
      { drun, drun1 } <= 0;
   else if(run) drun <= drun1;
   if((dr | dw) && (addr[3:1] == 3'h3)) begin
      drun <= !dr & dw;
      drun1 <= !dr & dw & data[12];
   end
   if(dw && addr[3:1] == 3'h2) bp <= data;
end

endmodule
`endif
```

⟨*debugging*⟩≡
```
`ifdef DEBUGGING
if(dw) case(daddr)
   3'h0: { sp, T } <= { spdec, din };
   3'h1: { rp, R } <= { rpdec, din };
   3'h3: { c, state, sp, rp } <=
           { din[10:8],
             din[sdep+3:4], din[rdep-1:0] };
   3'h4: P <= din;
   3'h5: T <= din;
   3'h6: R <= din;
   3'h7: I <= din;
   default ;
endcase
if(dr) case(daddr)
   3'h0: sp <= spinc;
   3'h1: rp <= rpinc;
   default ;
endcase
`endif
```

⟨*debugging read*⟩≡
```
‘ifdef DEBUGGING
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 or N or toR or bp)
if(!dr || run) dout = 'h0;
else case(daddr)
   3'h0: dout = N;
   3'h1: dout = toR;
   3'h2: dout = bp;
   3'h3: dout = { run, 4'h0, c, state,
                  {4-sdep{1'b0}}, sp,
                  {4-rdep{1'b0}}, rp };
   3'h4: dout = P;
   3'h5: dout = T;
   3'h6: dout = R;
   3'h7: dout = I;
endcase
‘endif
```

⟨*debugging-ports*⟩≡
```
‘ifdef DEBUGGING
   input [2:0] daddr;
   input dr, dw;
   input ‘L din, bp;
   output ‘L dout;
‘endif
```

⟨*dbg senselist*⟩≡
```
‘ifdef DEBUGGING
or run or dw or daddr
‘endif
```

⟨*stack debugging*⟩≡
```
‘ifdef DEBUGGING
if(!run && dw) case(daddr)
   3'h0: dpush = 1;
   3'h1: rpush = 1;
   default ;
endcase
‘endif
```

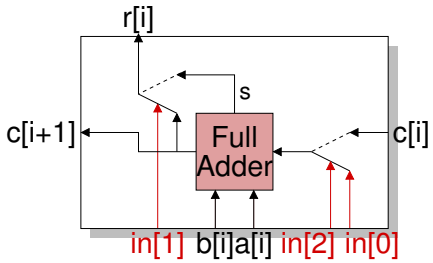

Figure 2: ALU bit slice

3.3 ALU

The ALU just computes the sum with possible carry-ins, the logical operations, and a zero flag. It reuses the same logic (essentially what comprises a full adder) to do both sums and logic. Figure 2 illustrates the logic that processes one bit of the ALU operation: Two multiplexers and one full adder (or the equivalent logic) per bit is sufficient to implement an ALU. The carry works as an AND gate if the carry in is 0 (both a and b input must be 1 to create a carry out), an OR gate if the carry in is 1 (both a and b input must be 0 to not create a carry out), and the sum is an XOR of a and b without carry in, and an XNOR with carry in. The XNOR operation of the ALU is not used. When the carry is propagated, a normal sum is generated; in this case, the result r selected is always the sum.

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

   wire        ‘L r1, r2;
   wire [1:0]  carries;

   assign r1 = T ^ N ^ carries;
   assign r2 = (T & N) |
               (T & carries‘L) |
               (N & carries‘L);
// This generates a carry *chain*, not a loop!
   assign carries =
        prop ? { r2[1-1:0], (c | selr) & andor }
             : { c, {(1){andor}}};
   assign res = (selr & ~prop) ? r2 : r1;
   assign carry = carries[1];
   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 { prop, selr, andor } = inst;
```

3.4 Stacks

The stacks are modelled as block RAM in the FPGA. In an ASIC, this is implemented with latches. The block RAM (or register file) needs one read and one write port.

⟨*Stack*⟩≡
```
module stack(clk, sp, spdec, push, gwrite, in, out);
   parameter dep=2, l=16;
   input clk, push, gwrite;
   input [dep-1:0] sp, spdec;
   input `L in;
   output `L out;

   reg `L stackmem[0:(1<<dep)-1];

`ifndef FPGA
   reg [dep-1:0] i;

   always @(clk or push or gwrite or spdec or in)
      if(~clk)
         if(gwrite)
            for(i=0; i<(1<<dep); i=i+1)
               stackmem[i] <= in;
         else if(push) stackmem[spdec] <= in;
`else
   always @(posedge clk)
      if(push)
         stackmem[spdec] <= in;
`endif

  assign out = stackmem[sp];

endmodule // stack
```

References

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