b16 — Ein Forth Prozessor im FPGA

Bernd Paysan

2. Mai 2004

Zusammenfassung

Dieser Artikel präsentiert Architektur und Implementierung des b16 Stack-Prozessors. Dieser Prozessor ist von Chuck Moores neusten Forth-Prozessoren inspiriert. Das minimalistische Design paßt in kleine FPGAs und ASICs und ist ideal geeignet für Applikationen, die sowohl Steuerung als auch Berechnungen benötigen. Die synthetisierbare Implementierung erfolgt in Verilog.

rev 1.0: Ursprüngliche Version

rev 1.1: Interrupts

Einleitung

Minimalistische CPUs können in vielen verschiedenen Designs benutzt werden. Eine State-Maschine ist oft zu kompliziert und zu aufwendig zu entwickeln, wenn es mehr als ein paar wenige States gibt. Ein Programm mit Subroutinen kann viel komplexere Aufgaben erledigen, und ist dabei noch einfacher zu entwickeln. Auch belegen ROM- und RAM-Blöcke viel weniger Platz auf dem Silizium als „Random Logic“. Das gilt auch für FPGAs, bei denen „Block RAM“ im Gegensatz zu Logik-Elementen reichlich vorhanden ist.

Die Architektur lehnt sich an den c18 von Chuck Moore [1] an. Der exakte Befehlsmix ist etwas anders, ich habe zugunsten von Divisionsstep und Forth-üblicher Logikbefehle auf `2*` und `2/` verzichtet; diese Befehle lassen sich aber als kurzes Makro implementieren. Außerdem ist diese Architektur byte-adressiert.

Das ursprüngliche Konzept (das auch schon synthetisierbar war, und ein kleines Beispielprogramm ausführen konnte) war an einem Nachmittag geschrieben. Die aktuelle Fassung ist etwas beschleunigt, und läuft auch tatsächlich in einem Alterea Flex10K30E auf einem FPGA-Board von Hans Eckes. Die Größe und Geschwindigkeit des Prozessors kann man damit auch abschätzen.

Flex10K30E Etwa 600 LCs, die Einheit für Logik-Zellen im Altera[1]. Die Logik zur Ansteuerung des Eval-Boards braucht nochmal 100 LCs. Im langsamsten Modell könnte man etwas mehr als 25MHz erreichen.

Xfab 0.6μ $\sim$1mm^2 mit 8 Stack-Elementen, das ist eine Technologie mit nur 2 Metal-Lagen.

TSMC 0.5μ $<$0.4mm^2 mit 8 Stack-Elementen, diese Technologie hat 3 Metal-Lagen. Mit einer etwas optimierten ALU kommt man mit der 5V-Library auf 100MHz.

Die ganze Entwicklung (bis auf das Board-Layout und Testsynthese für ASIC-Prozesse) ist mit freien oder umsonsten Tools geschehen. Icarus Verilog ist in der aktuellen Version für Projekte dieser Größenordnung ganz brauchbar, und Quartus II Web Edition ist zwar ein großer Brocken zum Downloaden, kostet aber sonst nichts (Pferdefuß: Windows NT, die Versionen für richtige Betriebssysteme kosten richtig Geld).

Ein paar Sätze zu Verilog: Verilog ist eine C-ähnliche Sprache, die allerdings auf den Zweck zugeschnitten ist, Logik zu simulieren, und synthetisierbaren Code zu geben. So sind die Variablen Bits und Bitvektoren, und die Zuweisungen sind typischerweise non-blocking, d.h. bei Zuweisungen werden zunächst erst einmal alle rechten Seiten berechnet, und die linken Seiten erst anschließend verändert. Auch gibt es in Verilog Ereignisse, wie das Ändern von Werten oder Taktflanken, auf die man einen Block warten lassen kann.

1 Übersicht über die Architectur

Die Kernkomponenten sind

- Eine ALU
- Ein Datenstack mit top und next of stack (T und N) als Inputs für die ALU

[1] Eine Logik-Zelle kann eine Logik-Funktion mit vier Inputs und einem Output berechnen, oder einen Voll-Addierer, und enthält darüber hinaus noch ein Flip-Flop.

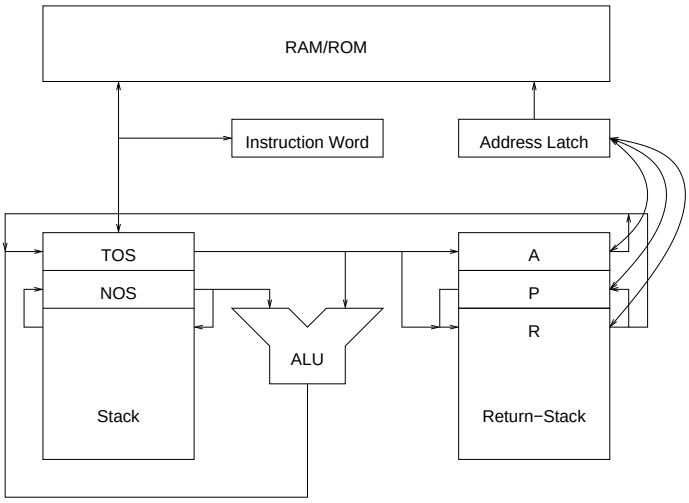

Abbildung 1: Block Diagram

- Ein Returnstack, bei dem der top of return stack (R) als Adresse genutzt werden kann
- Ein Instruction Pointer P
- Ein Adreßregister A
- Ein Adreßlatch `addr`, um externen Speicher zu adressieren
- Ein Befehlslatch I.

Ein Blockdiagram zeigt Abbildung 1.

1.1 Register

Neben den für den Benutzer sichtbaren Latches gibt es auch noch Steuerlatches für das externe RAM (`rd` und `wr`) und Stackpointer (`sp` und `rp`), Carry `c` und den Wert `incby`, um den `addr` erhöht wird.

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 Befehlssatz

Es gibt insgesamt 32 verschiedene Befehle. Da in ein 16-Bit-Wort mehrere Befehle 'reinpassen, nennen wir die einzelnen Plätze für einen Befehlwortes einen „Slot“, und das Befehlswort selbst „Bundle“. Die Aufteilung hier ist 1,5,5,5, d.h. der erste Slot ist nur ein Bit groß (die höherwertigen Bits werden mit 0 aufgefüllt), die anderen alle 5 Bit.

Die Befehle in einem Befehls-Wort werden der Reihe nach ausgeführt. Jeder Befehl braucht dabei einen Takt, Speicherzugriffe (auch das Befehlsholen) brauchen nochmal einen Takt. Welcher Befehl gerade an der Reihe ist, wird in der Variablen `state` gespeichert.

Der Befehlssatz teilt sich in vier Gruppen, Sprünge, ALU, Memory und Stack. Tabelle 1 zeigt eine Übersicht über die Befehle.

Sprünge verwenden den Rest des Befehlswort als Zieladresse (außer `ret` natürlich). Dabei werden nur die untersten Bits des Instruction Pointers P ersetzt, es wird nichts addiert. Für Befehle im letzten Slot bleibt da natürlich nichts mehr übrig, die nehmen dann T (TOS) als Ziel.

	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	

Tabelle 1: Instruction Set

⟨*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)
```

Die eigentlichen Befehle werden dann abhängig von `inst` ausgeführt:

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

2.1 Sprünge

In Einzelnen werden die Sprünge wie folgt ausgeführt: Die Sprungadresse wird nicht im P-Register gespeichert, sondern im Adreßlatch `addr`, das für die Adressierung des Speichers genutzt wird. Das Register P wird dann nach dem Befehlsholen mit dem inkrementierten Wert von `addr` gesetzt. Neben `call`, `jmp` und `ret` gibt's auch bedingte Sprünge, die auf 0 oder Carry testen. Das unterste Bit auf dem Returnstack wird genutzt, um das Carryflag zu sichern. Unterprogramme lassen also das Carryflag in Ruhe. Bei den bedingten Sprüngen muß man als Forther berücksichtigen, daß die den getesteten Wert nicht vom Stack nehmen.

Der Einfachheit beschreibe ich den Effekt eines jeden Befehls noch in einer Pseudo-Sprache:

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*

jnc (—) **if**$(c = 0)$ P ← *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-Operationen

Die ALU-Befehle nutzen die ALU, die aus T und N ein Ergebnis `res` und das Carry-Bit ausrechnet. Ausnahme ist der Befehl `com`, der einfach nur T invertiert — dazu braucht man keine ALU.

Die beiden Befehle `*+` (Multiplikationsschritt) und `/-` (Divisionsschritt) schieben das Ergebnis noch über das A-Register und das Carry-Bit. `*+` addiert N zum T, wenn das Carry gesetzt ist, und schiebt das Ergebnis eins nach rechts.

`/-` addiert auch N zum T, prüft aber, ob es dabei eine Überlauf gegeben hat, oder ob das alte Carry gesetzt war. Dabei schiebt es das Ergebnis eins nach links.

Normale ALU-Befehle nehmen einfach das Resultat der ALU in T und c, und laden N nach.

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 $\leftarrow c_n, r$, 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 $\leftarrow r$, 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 Speicher-Befehle

CHUCK MOORE benutzt nicht mehr den TOS als Adresse, sondern hat ein A-Register eingeführt. Wenn man Speicherbereiche kopieren will, braucht man noch ein zweites Adreßregister; dafür nimmt er den Top-of-Returnstack R. Da man den P nach jedem Zugriff erhöhen muß (auf den nächsten Befehl), ist in der Adressierungslogik schon ein Autoinkrement enthalten. Das wird dann auch für andere Zugriffe verwendet.

Speicher-Befehle, die im ersten Slot stehen, und nicht über P indizieren, inkrementieren den Pointer nicht; damit sind Read-Modify-Write-Befehle wie +! einfach zu realisieren. Speichern kann man nur über A, die beiden anderen Pointer sind nur zum Lesen gedacht.

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 });
```

Der Zugriff kann nicht nur wortweise, sondern auch byteweise erfolgen. Dazu gibt es zwei Write-Leitungen. Für byteweises Speichern wird das untere Byte in T ins obere kopiert.

⟨*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
```

Speicherzugriffe benötigen einen Extra-Takt. Dabei wird das Ergebnis des Speicherzugriffs verarbeitet.

⟨*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⟩
```

Eine kleine Besonderheit gibt's beim angesetzten Instruction-Fetch (dem NEXT der Maschine) noch: Wenn der aktuelle Speicherbefehl ein Literal ist, müssen wir `incaddr` statt P nehmen.

⟨*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);
```

Ist der Zugriff beendet, muß das Resultat abgearbeitet werden — bei Load-Zugriffen der Wert auf den Stack oder ins Instruction-Register geladen, bei Store-Zugriffen der TOS gedropt werden.

⟨*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;
```

Außerdem muß bei Bedarf die inkrementierte Adresse zurück in den entsprechenden Pointer geladen werden.

⟨*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] })
```

Damit der erste Befehl (nur `nop` oder `call`) keine unnötige Zeit verbraucht, wird ein `nop` hier einfach übersprungen. Das ist der zweite Teil des NEXTs.

⟨*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;
```

Hier werden auch die Interrupts abgearbeitet. Interrupts werden beim Instruction-Fetch akzeptiert. Statt P zu erhöhen, wird hier ein Call auf den Interruptvektor (Adressen ab $200) ins Befehlsregister geladen. Die Interruptroutine muß lediglich bei Bedarf A sichern, und den Stack so hinterlassen wie sie ihn vorgefunden hat. Da drei Befehle hintereinander ohne Unterbrenchung ausgeführt werden können, sehe ich keine Interruptsperrung vor, oder eine über die externe Interrupt-Unit verwaltete. Die letzten drei Befehle einer Interrupt-Routine währen dann `a! >a ret`.

2.4 Stack-Befehle

Die Stack-Befehle ändern den Stackpointer und schieben entsprechend die Werte in und aus den Latches. Bei den 8 benutzten Stack-Effekten fällt auf, daß `swap` fehlt. Stattdessen gibt es `nip`. Der Grund ist eine damit mögliche Implementierungs-Option: Man kann das separate Latch N einfach weglassen, und diesen Wert direkt aus dem Stack-RAM holen. Das dauert zwar länger, spart aber Platz.

Außerdem behauptet Chuck Moore, daß man `swap` gar nicht so nötig braucht — wenn es nicht verfügbar ist, behilft man sich mit den anderen Stack-Operationen, und wenn es gar nicht anders geht, gibt's ja immer noch `>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 };
```

Wer auf `swap` nicht verzichten möchte, kann einfach die Implementierung des `nip`s in der ersten Zeile ersetzen:

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

3 Beispiele

Ein paar Beispiele sollen zeigen, wie man den Prozessor programmiert. Die Multiplikation funktioniert wie gesagt über das A-Register. Es ist ein Extra-Schritt nötig, weil ja jedes Bit zunächst einmal ins Carry geschoben werden muß. Da `call` das Carry-Flag löscht, brauchen wir uns darum nicht zu kümmern.

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

Auch bei der Division muß ein Extra-Schritt eingelegt werden. Eigentlich bräuchten wir hier echt ein `swap`, da wir aber keines haben, nehmen wir zunächst `over` und nehmen in Kauf, daß wir ein Stackelement mehr brauchen als im anderen Fall. Anders als bei `mul` müssen wir hier nach dem `com` das Carry wieder löschen. Und am Schluß müssen wir noch den Rest durch zwei teilen, und den Carry nachschieben.

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

Das nächste Beispiel ist etwas komplizierter, weil ich hier eine serielle Schnittstelle emuliere. Bei 10MHz muß jedes Bit 87 Takte brauchen, damit die Schnittstelle 115200 Baud schnell ist. Erst hinter dem zweiten Stop-Bit haben wir Ruhe, die Gegenseite wird sich schon wieder synchronisieren, wenn das nächste Bit kommt.

⟨*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 ;
```

Der ; hat hier wie bei ColorForth die Funktion des EXITs, so wie der : nur ein Label einleitet. Steht vor dem ; ein Call, so wird der in einen Sprung umgewandelt. Das spart Returnstack-Einträge, Zeit und Platz im Code.

4 Der Rest der Implementierung

Zunächst einmal den Rumpf der Datei.

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

`define L [1-1:0]
`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 Toplevel

Die CPU selbst besteht aus verschiedenen Teilen, die aber alle im selben Verilog-Modul implementiert werden.

⟨*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
```

Zunächst braucht Verilog erst mal Port Declarations, damit es weiß, was Input und Output ist. Die Parameter dienen dazu, auch andere Wortbreiten oder Stacktiefen einfach einzustellen.

⟨*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
```

Die ALU wird mit der entsprechenden Breite instanziiert, und die nötigen Leitungen werden deklariert

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

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

Da die Stacks nebenher arbeiten, müssen wir noch ausrechnen, wann ein Wert auf den Stack gepusht wird (also **nur** wenn etwas gespeichert wird).

⟨*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
```

Zu den Stacks gehören nicht nur die beiden Stack-Module, sondern auch noch inkrementierter und dekrementierter Stackpointer. Beim Returnstack kommt erschwerend dazu, daß der Top of Returnstack manchmal geschrieben wird, ohne daß sich die Returnstacktiefe ändert.

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

stack #(sdep,l) dstack(clk, sp, spdec,
                       dpush, N, toN);
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){(~state[2] | tos2r)}};
assign rpinc = rp+{{(rdep-1){1'b0}}, 1'b1};
```

Der eigentliche Kern ist das voll synchrone Update der Register. Die brauchen einen Reset-Wert, und für die verschiedenen Zustände müssen die entsprechenden Zuweisungen codiert werden. Das meiste haben wir weiter oben schon gesehen, nur das Befehlsholen und die Zuweisung des nächsten States und des Wertes von `incby` bleibt noch zu erledigen.

⟨*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)
```

Als Reset-Wert stellen wir die CPU so ein, daß sie sich gerade den nächsten Befehl holen will, und zwar von der Adresse 0. Die Stacks sind alle leer, die Register enthalten alle 0.

⟨*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;
```

Der Übergang zum nächsten State (das NEXT innerhalb eines Bundles) wird getrennt erledigt. Das ist nötig, weil die Zuweisungen der anderen Variablen zum Teil nicht nur abhängig vom aktuellen State sind, sondern auch vom nächsten (z.B. wann das nächste Befehlswort geholt werden soll).

⟨*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

Die ALU berechnet einfach die Summe mit den verschiedenen möglichen Carry-ins, die logischen Operationen, und ein Zero-Flag. Zwar können hier gemeinsame Resourcen verwendet werden (die XORs des Volladdierers können auch die XOR-Operation machen, und die Carry-Propagation könnte OR und AND berechnen), dieses Quetschen von Logik überlassen wir aber dem Synthesetool.

⟨*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
```

Die ALU hat die Ports T und N, carry in und die untersten 3 Bits des Befehls als Input, ein Ergebnis, carry out und der Test auf 0 als 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

Die Stacks werden im FPGA als Block-RAM implementiert. Dazu sollten sie am besten nur einen Port haben, denn solche Block-RAMs gibt's auch in kleinen FPGAs. Im ASIC wird diese Art von Stack mit Latches implementiert. Dabei könnte man auch Read- und Write-Port trennen (oder für FPGAs, die dual-ported RAM können), und sich den Multiplexer für `spset` sparen.

⟨*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 Weitere mögliche Optimierungen

Eigentlich könnte man Speicherzugriffe und Berechnungen auf den Stacks überlappend ausführen. Durch die separaten Pointer-Register ist das möglich. Die Verständlichkeit des Prozessors würde darunter aber sicher leiden, und der kritische Pfad würde wohl auch länger. Angesichts einer garantierten Beschleunigung um 25% (der Zyklus zum Befehlsholen fällt weg) und einer maximalen Beschleunigung um 100% (bei speicher-intensiven Anwendungen) könnte es aber wert sein — wenn der Platz da ist.

Falls Platz knapp ist, kann man fast alle Register als Latches auslegen. Nur T muß ein echtes Flip-Flop bleiben. Für FPGAs ist das keine Option, Flip-Flips sind dort allemal günstiger.

4.5 Skalierbarkeit

Zwei mögliche Ansatzpunkte gibt es, den b16 schnell an eigene Wünche anzupassen: Die Wortbreite und die Stacktiefe. Die Stacktiefe ist dabei der einfachste Punkt. Die gewählte Tiefe 8 ist für den Bootloader ausreichend, kann aber für komplexere Applikationen Schwierigkeiten bereiten. Einfachere Applikationen sollten dagegen mit einem kleineren Stack zurechtkommen.

Die Wortbreite kann auch der Applikation angepaßt werden. So wird eine auf 12 Bit abgemagerte Version in einem Projekt bei meinem Arbeitgeber Mikron AG eingesetzt. Dabei muß man natürlich noch das Decodieren der einzelnen Befehle im Slot ändern, und die Logik zum Überspringen des ersten `nop`s anpassen.

Außerdem kann man natürlich einzelne Befehle auswechseln. So wird bei der 12-Bit-Version kein Byte-Zugriff auf den Speicher benötigt, aber relativ viele Bit-Zugriffe. Entsprechend werden die Byte-Zugriffe dann durch Bit-Operationen auf den obersten Teil des Speichers ersetzt, und das Register incby wegoptimiert (nur noch Wort-Zugriffe).

5 Entwicklungsumgebung

Hier könnte ich noch ein etwas längeres Listing präsentieren, diesmal in Forth. Ich will mich aber auf eine Funktionsbeschreibung beschränken. Alle drei Programme sind in einer einzigen Datei vereint, und erlauben damit eine interaktive Benutzung des Simulators und des Targets.

5.1 Assembler

Der Assembler ist leicht an CHUCK MOORE's ColorForth angelehnt. Es gibt aber keine Farben, sondern nur normale Interpunktion, wie in Forth üblich. Der Assembler ist schließlich in Forth geschrieben, und erwartet damit Forth-Tokens.

Labels definiert man mit `:` und `|`. Erstere geben automatisch einen Call, können aber mit `'` auf den Stack gelegt werden. Letztere entsprechen etwa einem interaktiven `Create`. Labels können nur rückwärts aufgelöst werden. Literals muß man explizit mit `#` oder `#c` vom Stack nehmen. Die Zuordnung in Slots nimmt der Assembler selber in die Hand. Ein `ret` compiliert man normal mit einem `;`, der vorangestellte Calls in einen `jmp` konvertiert. Man kann Makros definieren (`macro: ... end-macro`).

Auch die aus Forth bekannten Kontrollstrukturen können (bzw. müssen — für Vorwärtssprünge) verwendet werden. `IF` wird zu einem `jz`, `jnz` erreicht man mit `-IF`. `cIF` und `-cIF` entsprechen `jnc` und `jc`. Entsprechende Prefixes gibt es auch für `WHILE` und `UNTIL`.

5.2 Downloader

Im FPGA ist ein Stück Block-RAM mit einem Programm vorbelegt, dem Boot-Loader. Dieses kleine Programm läßt ein Lauflicht laufen, und wartet auf Kommandos über die serielle Schnittstelle (115.2kBaud, 8N1, kein Handshake). Es gibt drei Kommandos, die mit ASCII-Zeichen eingeleitet werden:

0 *addr, len, <len∗data>:* Programmiere den Speicherbereich ab *addr* mit *len* Datenbytes

1 *addr, len:* Lese *len* Bytes vom Speicherbereich ab *addr* zurück

2 *addr:* Führe das Wort an *addr* aus.

Diese drei Befehle reichen aus, um den b16 interaktiv zu bedienen. Auch auf der Host-Seite reichen ein paar Befehle aus:

comp Kompiliert bis zum Ende der Zeile, und schickt das Ergebnis an das Eval-Board

eval Kompiliert bis zum Ende der Zeile, schickt das Ergebnis ans Eval-Board, führt den Code aus, und setzt den RAM-Pointer des Assemblers zurück an den Ausgangspunkt

sim Wie `eval`, nur wird das Kompilat nicht vom FPGA ausgeführt, sondern vom Emulator

check (addr u —) Liest den entsprechenden Speicherbereich vom Eval-Board, und zeigt ihn mit `dump` an

6 Ausblick

Mehr Material gibt's auf meiner Homepage [2]. Alle Quellen sind unter GPL verfügbar. Wer ein bestücktes Board haben will, wendet sich am besten an HANS ECKES. Und wer den b16 kommerziell verwenden will, an mich.

Literatur

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

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