Klaus Kohl-Schöpe

Forth-Prozessor K1 und SmallForth auf MAX1000

*Ende 2018 hatte ich 4 günstige FFGA-Boards vorgestellt - entwickelt und vertrieben durch Arrow und Trenz. Schon damals war meine Idee, auf diesen Boards einen Forth-Prozessor im Stil von J1 zu realisieren. Jedoch gab mir erst der Lockdown der letzten Monate genügend Zeit für dieses Projekt und möchte die schon funktionierende Version vorstellen. Das entsprechende Quartus II v18.1 Projekt und das zugehörige SmallForth (kurz SMAF) kann von meiner Webseite **http://www.mcforth.net** heruntergeladen und getestet werden. Ich kann nur hoffen, dass ich für mein erstes eigenes FPGA-Projekt nicht gesteinigt werde.*

Vorgeschichte

Als Microcontroller-Spezialist arbeite ich aktuell fast ausschließlich mit C(++), jedoch mit einer langen Erfahrung in Assembler, Basic und Forth. Dabei hatte ich auch häufig intensiven Kontakt mit Forth-Prozessoren wie NC4000, RTX2000 oder FPR1600 und oft darauf eigene Forth-Versionen realisiert. Auch bei meiner letzten Implementierung mcForth wurde für den virtuellen Prozessor ein eigener Befehlssatz definiert und in Assembler simuliert.

Da ich mich aktuell auch wieder mehr mit FPGA‘s befassen darf und mir auch schon längere Zeit verfügbare IP‘s wie z.B. den J1 von James Bowman angesehen habe, nutzte ich die durch Corona verursachte Freizeit und Urlaub um mich sowohl in Verilog als auch in Prozessordesign auf FPGA einzuarbeiten. Was lag dabei näher, als diesen J1-Ansatz zu verwenden und auf einer der damals vorgestellten günstigen FPGA-Boards zu implementieren. Die Wahl fiel auf das MAX1000-Board, weil der MAX10 Flash-basierend ist und damit sofort startet, der Chip mit 8K LE und 42kByte RAM groß genug für das Forth-IP ist und genügend Speicher für das SmallForth hat. Außerdem erlauben Schnittstellen wie UART (über FDTI-Chip), 8 LEDs, Taster, Beschleunigungssensor und je 8MByte großes SDRAM und SPI-Flash viele Anwendungen.

Von J1 zu K1

Als „Geschädigter“ von Forth-Prozessoren versuche ich, Eigenheiten wie 16- oder 32-Bit pro Adresse oder weit vom aktuellen Standard abweichende Forth-Versionen ohne Prüfung von Kontrollstrukturen zu vermeiden. Deshalb war meine erste Aktivität die Überarbeitung des Befehlssatzes und des Speicherinterface.

Im einzelne gab es folgende Wünsche:
- Speicher ist als 8-Bit pro Adresse organisiert
- Befehle und Variablen sind 16-bit-aligned
- High-Endian (ist einfacher im Dump zu lesen)
- Literals können ±16k abdecken (Bit 15 <= Bit 14)
- Calls verwenden absolute Adressen (addr/2 + $8000)
- Sprünge sind relativ ±4kByte
- Zusätzlich ein Sprungbefehl für `FOR...NEXT`-Schleife
- Arithmetik ist erweitert und auch bei @ nutzbar
- Neue Befehle: `UM* * Divstep SP! EXECUTE GOTO`

Dazu wurde der Befehlssatz umgestellt, wie man im Vergleich von Bild 1 bis Bild 3 sehen kann. Im Bild 4 sind dann noch die verwendeten Abkürzungen aufgelistet.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	value															Literal (0..32K)
0	0	0	target													Sprung
0	0	1	target													Sprung wenn T=0
0	1	0	target													Call
0	1	1	;	ALU				>N	>R	!		rr		ss		Arithmetik

Bild 1: Befehlssatz J1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	literal															Literal (±16k)
1	0	addr														Call
1	1	0	0	relativ												Sprung
1	1	0	1	relativ												Sprung wenn T=0
1	1	1	0	relativ												Sprung wenn R<>0
1	1	1	1	0	;	>R	>N	ALU				BW	@!	ss		Speicherzugriff
1	1	1	1	1	;	>R	>N	ALU				rr		ss		Arithmetik

Bild 2: Befehlssatz K1

ALU	J1	K1	K1 special
0	T	Tx	R
1	N	N	RA (R mit Bit 0=0)
2	T+N	N+Tx	Tx+1
3	T and N	N-Tx	Tx-1
4	T or N	N and Tx	Tx+2
5	T xor N	N or Tx	Tx-2
6	~ T	N xor Tx	Carry (0 oder 1)
7	N = T ?	~Tx	INV14: Tx xor $4000
8	N < T ?	N ashift Tx	Tx/2
9	N rshift T	N rshift Tx	Tx/2 unsigned
11	R	N lshift Tx	Tx*2
10	T-1	Tx = 0 ?	Tx < 0 ?
12	[T]	N u< Tx ?	DSP
13	N lshift T	N < Tx ?	RSP
14	DSP	Divstep	SP!
15	N u< T ?	*	UM*

Bild 3: Vergleich ALU-Befehle J1 und K1

Abkürzung	Bedeutung
TOS oder T	Oberster Datenstack-Wert
Tx	TOS oder Memory
~T oder ~Tx	Invert T(x) = T(x) xor $FFFF
NOS oder N	Zweiter Datenstack-Wert
TOR oder R	Oberster Returnstack-Wert
RSP	Returnstackpointer
DSP	Datenstackpointer
;	Exit: PC <= TOR
>N	Kopiere TOS nach NOS
@!	Lesen (0) oder Schreiben (1)
>R	Kopiere TOS nach TOR
BW	Byte (0) oder 16-Bit-Wort (1)
value	Wert (Bit 15 wird 0)
literal	Wert (Bit 15 wird Bit 14)
addr	PC wird addr*2
relativ	PC wird PC+2±(relativ*2)
u<	Vergleich: Unsigned kleiner
INV14	Bit 14 invertieren (für Literal)
ashift	Shift rechts mit Bit 15=Bit 14
rshift	Shift rechts mit Bit 15=0
lshift	Shift links mit Bit 0=0

Bild 4: Abkürzungen

Der Speicher wurde auf 32KByte für Programme gelassen (war 16k*16), weil sowohl der interne Speicher des MAX10 als auch die verwendbaren Opcodes dies limitiert. Der restliche Speicher ab $8000 enthält die Schnittstellen zur Peripherie und evtl. in Zukunft noch Videospeicher. Beide Stacks verwenden auch ein 9Kb-Speicher des MAX10 und haben damit je 256 Einträge. Die Stackpointer sind aber 9 Bit, um Überläufe zu erkennen.

Bei Literal hat der J1 die unteren 15 Bits verwendet und damit 0…32767 realisieren können. Negative Zahlen konnten durch den INVERT-Befehl erreicht werden. Da in Anwendungen eher auch negative Zahlen und Adressen im $FFxx-Bereich vorkommen, habe ich definiert, dass bei Literals das Bit 14 auch in Bit 15 übernommen wird und damit -16364 (=49152) bis 16383 mit einem Takt realisiert wurde. Ansonsten invertiere ich vorher Bit 14 bei einem Literal und führe danach den `INV14`-Befehl aus.

Auch bei der Definition von Call und Sprüngen habe ich einiges geändert. Zum einen kann immer noch der gesamte 32KByte-Speicher bei Calls erreicht werden und da die Sprünge normalerweise nur in Bedingungen innerhalb einer :-Definition vorkommen, sollten ±4KByte ausreichen. Der zusätzliche Sprungbefehl für die `FOR` … `NEXT`-Schleife erniedrigt TOR um 1 und springt, solange der Wert vorher ungleich 0 war. Ist er 0, dann wird der Wert vom Returnstack entfernt und die Schleife verlassen. Gleichzeitig `>R` und `EXIT` sollte eigentlich nicht vorkommen. Deshalb habe ich dies im K1 genutzt, um TOS direkt als neue Befehlsadresse zu verwenden und damit `EXECUTE` (bei rr=%01 - rettet Rücksprungadresse), `GOTO` und sogar `PERFORM (ist @ EXECUTE)` zu implementieren.

Meist wird TOS oder/und NOS für Arithmetik verwendet. K1 verwendet jedoch beim Lesen vom Speicher nicht TOS (enthält die Adresse des Speicherzugriffes), sondern den vom Speicher gelesenen Wertes um z.B. Kombination von `@` mit `+` … zu erlauben. Multiplikation in Hardware kostet beim MAX10 nur eine DSP-Einheit, eine Zeile Verilog und auch nur einen Takt. Für Division gibt es den DivStep in Anlehnung an NC4000 oder RTX und braucht dann ca. 20 Takte. Allerdings wird dabei kein getrenntes MD-Register, sondern TOR als Teiler genutzt. Damit entfallen alle Klimmzüge zur Rettung von Register. Es gibt auch ein Carry-Flag für Addition, Subtraktion und Shiftbefehle. Dafür nutzt K1 das Bit 0 des PC, das ja nicht gebraucht, aber bei Calls ... gerettet wird. Es gab nur ein kleines Problem bei Inline-Parameter wie Strings, weshalb gibt es den Befehl `RA` (Returnstack-Adresse), welche die Rücksprungadresse aus TOR mit Bit0=0 liefert.

Durch die Umstellung des Befehlssatzes konnte ich das freie Bit des J1 (grau in Bild 1) gut nutzen. Das war mir aber doch zu wenig und deshalb habe ich auch noch eine Sonderbedingung herangezogen. Bei ALU und Speicherbefehle sind Änderungen der beiden Stackzeiger von -2 bis +1 möglich, wobei -2 beim Datenstack eigentlich nicht sinnvoll genutzt werden kann und deshalb hier missbraucht wurde, um weitere 16 „Spezialbefehle“ zu ermöglichen. Diese ändern die Stacktiefe nicht oder um +1, wenn das Bit 8 (>N) im Befehl gesetzt ist.

	J1 (rr/ss)	K1 (rr)	K1 (ss)
00	---	---	--
01	1	1	1
10	-2	-2	1 (bei >N) oder 0
11	-1	-1	-1

Bild 5: Stackänderungen

Das SMAF (SmallForth)

Damit kommen wir schon zu dem zugehörigen 16-Bit Forth, das hier nur im RAM läuft und deshalb keine Besonderheiten für Nutzung von Flash enthält.

```
init 'init quit find words .s dump ?Abort" ?Abort" ?abort abort 'abort postpone ['] ' name> >name body> >body \IFNDEF \IFDEF \IF \ELSE \THEN ENDCASE ENDOF OF CASE NEXT ?FOR FOR BEGIN WHILE UNTIL REPEAT AGAIN ELSE THEN IF AHEAD recurse Does> Create Variable Constant ; : :noname Header forget indirect restrict immediate ." s" [char] char Literal ] [ call, $, , c, align allot $>number? >number . .r u. u.r 0u.r d. d.r decimal hex #> sign #s # hold <# \\ \ ( -parse parse refill source upc accept type cr spaces space ms key key? emit emit? button? acc_data acc_baud pmod_dir pmod_data counter_hi counter_lo leds uart_data uart_state uart_baud 2@ 2! count ! @ c! c@ dabs d- d+ / m/mod um/mod * m* um* nop negate invert xor or and cells chars flip du2/ d2/ d2* rshift ashift lshift u2/ 2/ 2* within max min abs u< < <> = 0<> 0= 0< aligned cell+ char+ 2- 1- 2+ 1+ - + goto execute ?exit exit rdrop ra> r> r@ >r rclear rdepth ?dup -rot rot 2drop 2swap 2over 2dup nip drop swap tuck over dup dclear depth span >in hld dpl base last state here voc-link #ramstart #ramend #tib #c/tib true false bl #msb #bits
```

Es ist weitestgehend ANS-Konform und überwacht die Kontrolstrukturen in :-Definitionen. Auf `DO`...`LOOP` wurde verzichtet und dafür das sehr schnelle (3 + n Takte) `FOR`...`R@`...`NEXT` verwendet, wobei bei `FOR` ein `1- >R` compiliert wird und deshalb die gewünschte Schleifenanzahl anzugeben ist. Bei 0 wäre es dann 65536 mal oder man verwendet `?FOR`, was dies verhindert und die Schleife überspringt. Zusätzlich gibt es die schon häufig beschriebene `CASE` ... `OF` ... `ENDOF` ... `ENDCASE` Struktur.

Mit `FORGET <Name>` kann man die neueren Befehle wieder löschen. Da es keinen getrennten Speicher für Variablen oder Header und nur ein Vokabular gibt, wird einfach `HERE` wieder zurückgesetzt und der Linkpointer auf das letzte Wort gesetzt.

Mit `'init` und `'abort` gibt es zwei Variablen mit der CFA für Initialisierung und nach Fehler, hier mit NOP vorgelegt. Die Eingabe geht aktuell über die integrierte UART mit 115200 Baud. Da kein Filesystem realisiert ist, nutze ich z.B. Terraterm mit Copy/Paste für Eingabe bei 20ms pro Zeile. Bei einem Fehler werden alle weiteren Zeichen ignoriert, bis 100ms lang nichts mehr kommt.

Entwicklungsschritte

Eigentlich musste ich mich mit vier Themen befassen:

- K1 mit Speicher-Anbindung (nutzt 9Kb-RAM-Blöcke)
- Targetcompiler und Assembler für das SmallForth
- Das SmallForth selbst
- K1-Simulator zum Testen des SmallForth

Beginnend mit dem Verilog Code des J1 (kann man noch teilweise erkennen) ging es an die Umstellung des Befehlssatzes. Viel Kopfzerbrechen hat mir das Speicherinterface bereitet, das jetzt wie die Stacks die 9k-Blöcke des MAX10 nutzt. Bei diesen muss immer bei der positiven Flanke des Taktes die nächste Adresse anstehen. Das ist beim Lesen des Befehlscode einfach, da die nächst Adresse rechtzeitig berechnet wird. Auch beim Lesen von Daten steht schon die Adresse vor dem eigentlichen Befehl bereit und kann genutzt werden. Aber das man lesen oder schreiben will, wird erst durch den Befehl ermittelt und es kann deshalb erst am Ende des Taktes neue Daten geschrieben oder der Peripherie anzeigen werden, dass etwas gelesen wurde. Dies führt dazu, das Lesen unmittelbar nach Schreiben nicht geht, was aber normalerweise nicht vorkommt.

Im Betrieb will man auch mit dem Forth kommunizieren und die vorhandenen Schnittstellen nutzen. Darunter fallen die 8 verfügbaren LEDs, ein Taster (neben den LEDs, der zweite auf der anderen Seite ist Reset) und die UART über den FDTI-Chip, der auch für die Programmierung verwendet wird. Dafür verwendet der K1 den Adressbereich ab $FF00, was $-100 entspricht und damit auch durch ein 16-Bit-Literal ohne INV14 erreichbar ist.

Für erste Tests des Prozessors war der Simulator (Intel ModelSim) für das FPGA sehr nützlich, wenn auch aufwendig und es braucht Zeit. Deshalb habe ich im ersten Schritt auf den ganzen Spaß der Optimierung verzichtet und nur die für SMAF genutzten Befehle getestet.

Damit ich nicht alles mit der Hand codieren muss, folgten dann ein Assembler und ein Targetcompiler – natürlich wie bei mir üblich auch in Forth realisiert. Bei über 35 Jahren Erfahrung darin eigentlich kein Problem, jedoch fällt man trotzdem immer wieder über die eigentlich bekannten Probleme von 16-Bit Target (K1) auf 32-Bit Systeme (mein mcForth), High-Endian und Alignment.

Das SMAF gab es schon größtenteils aus einem älteren FPGA-Forth Projekt und wurde nur angepasst und erweitert. Jedoch läuft sowas garantiert nicht gleich beim ersten Mal und der Simulation aus der FPGA-Software hat sehr lange Testzyklen und braucht Millionen von Takten z.B. bei Ausgaben. Deshalb musste auch noch ein Simulator her, der mir die Testzeit auf Sekunden reduziert und Single-Step, Breakpoints oder einfache Debug-Ausgaben erlaubt. Da ich bisher auf Optimierungen verzichtet habe und damit die zu simulierenden Befehle überschaubar sind, war auch dieses Programm relativ schnell realisiert und zeigte mir in kurzer Zeit noch vorhandene Probleme. Für Ein-/Ausgabe verwendete ich die simulierte UART-Schnittstelle.

Erste Tests

Glücklicherweise lief dann die erste Version des SmallForth innerhalb kurzer Zeit auf dem simulierten K1. Auch im FPGA lief es fast sofort und zu meiner Verblüffung sogar mit 100MHz, obwohl ich mir das Timing überhaupt noch nicht angeschaut habe. Als ich dann noch einige Features in das FPGA integriert habe, zeigten sich erste Abstürze, was mich dazu bewog, das Board aktuell mit 48MHz laufen zu lassen.

Damit habe ich jetzt ein System, mit dem ich Arbeiten kann und viele Möglichkeiten zur Erweiterung bietet. Mittels des angepassten ANS-Tester habe ich den Befehlssatz geprüft und auch den Beschleunigungsaufnehmer kann ich über eine SPI-Schnittstelle schon ansprechen.

Nächste Schritte

Der Befehlssatz des Prozessors bietet enormes Potential zur Optimierung und diese sollte genutzt werden. Dies bedeutet aber auch erhebliche Arbeit am Target-Assembler und -Compiler. Auch der Simulator sollte dann der vollständigen FPGA-Implementierung entsprechen. Erste Tests zeigen, dass ich dabei >180Byte von 6KB spare.

Hardwareseitig werden schon ein 32-Bit Systemcounter (mit Prozessortakt), UART, LEDs, Taster und die I/O-Ports von PMOD und die SPI-Schnittstelle zum Beschleunigungsaufnehmer unterstützt. Die Schnittstelle zum integrierten Flash und dem 8MByte SPI-Flash steht ganz oben auf der Wunschliste, weil dann Autostart-Anwendungen möglich sind und ein Filesystem wieder interessant wird. Die Schnittstellen zu den Arduino Nano Pins und dem im MAX10 integrierte A/D-Wandler sind noch nicht realisiert. Da der J1 für Telespiele genutzt wurde, steht auch auf dem K1 einem Display und einer Joystick-Anbindung nichts im Wege. Die restlichen 9kByte RAM und mehr als 6k LE (der K1 braucht aktuell weniger als 20% der 8k LE) sollten dafür eigentlich ausreichen. Vermutlich werden nur die auf dem MAX1000 verfügbaren 8MByte SDRAM noch länger warten müssen, da dafür ein entsprechendes Speicherinterface benötigt wird.

Wie gleich am Anfang erwähnt, habe ich das entsprechende FPGA-Projekt, aber auch meine Entwicklungsumgebung für das SmallForth mit Sourcen des Target-Assembler/-Compiler, SmallForth, Simulator und Beispielprogramme auf **http://www.mcForth.net** zur privaten Nutzung zur Verfügung gestellt. Als Entwicklungsumgebung verwendete ich das ebenfalls auf dieser Webseite ausführlich beschriebene mcForth unter Windows. Entsprechende Batch-Dateien und zusätzliche Beschreibungen sollten die Einarbeitung einfacher machen.

Ich würde mich über Anregungen, Hinweise auf Fehler und Verbesserungen aber natürlich auch über damit realisierte Projekte freuen. Erreichbar bin ich auch für Fragen unter kks@designin.de.

Viel Spaß mit Forth-Prozessor K1 und SmallForth.
Euer Klaus Kohl-Schöpe.

- Ende -