b16, j1, MicroCore, N.I.G.E.: Forth im FPGA

Bernd Paysan

Forth–Tagung 2016, Augsburg

Inhalt

Motivation: b16

- Erfahrung mit einem 8051–Core und 3 Monaten, um den in einem Projekt ordentlich einzusetzen — und dann eine eingeschränkte, langsame, schwer zu programmierende (und teure) Lösung zu haben
- Besser: Minimalistische Forth–CPU (also näher an der Hochsprache), bei der in ein paar Tagen das Verilog und die Toolchain geschrieben werden konnte
- Inspiriert von Chuck Moore's c18
- Aktuelle Version ist näher am „normalen“ Forth

Motivation: b16

- Erfahrung mit einem 8051–Core und 3 Monaten, um den in einem Projekt ordentlich einzusetzen — und dann eine eingeschränkte, langsame, schwer zu programmierende (und teure) Lösung zu haben
- Besser: Minimalistische Forth–CPU (also näher an der Hochsprache), bei der in ein paar Tagen das Verilog und die Toolchain geschrieben werden konnte
- Inspiriert von Chuck Moore's c18
- Aktuelle Version ist näher am „normalen“ Forth

Motivation: b16

- Erfahrung mit einem 8051–Core und 3 Monaten, um den in einem Projekt ordentlich einzusetzen — und dann eine eingeschränkte, langsame, schwer zu programmierende (und teure) Lösung zu haben
- Besser: Minimalistische Forth–CPU (also näher an der Hochsprache), bei der in ein paar Tagen das Verilog und die Toolchain geschrieben werden konnte
- Inspiriert von Chuck Moore's c18
- Aktuelle Version ist näher am „normalen“ Forth

Motivation: b16

- Erfahrung mit einem 8051–Core und 3 Monaten, um den in einem Projekt ordentlich einzusetzen — und dann eine eingeschränkte, langsame, schwer zu programmierende (und teure) Lösung zu haben
- Besser: Minimalistische Forth–CPU (also näher an der Hochsprache), bei der in ein paar Tagen das Verilog und die Toolchain geschrieben werden konnte
- Inspiriert von Chuck Moore's c18
- Aktuelle Version ist näher am „normalen“ Forth

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

Die Forth–Maschine

Elemente:

Datenstack Zum Übergeben von Parametern für Subroutinen und Instruktionen (RPN, zero–address machine)

Return stack damit die Rücksprungadressen nicht im Weg sind

Wort Leerzeichenbegrenztes Symbol im Assembler, Aufruf oder Befehl im Maschinencode

Sonderzeichen–Bedeutung ':' definiert eine Subroutine (Wort), ';' springt zurück, '@' load (spricht: „fetch“), '!' store

Einfachheit Forth ist so einfach, dass man alle Aspekte verstehen kann

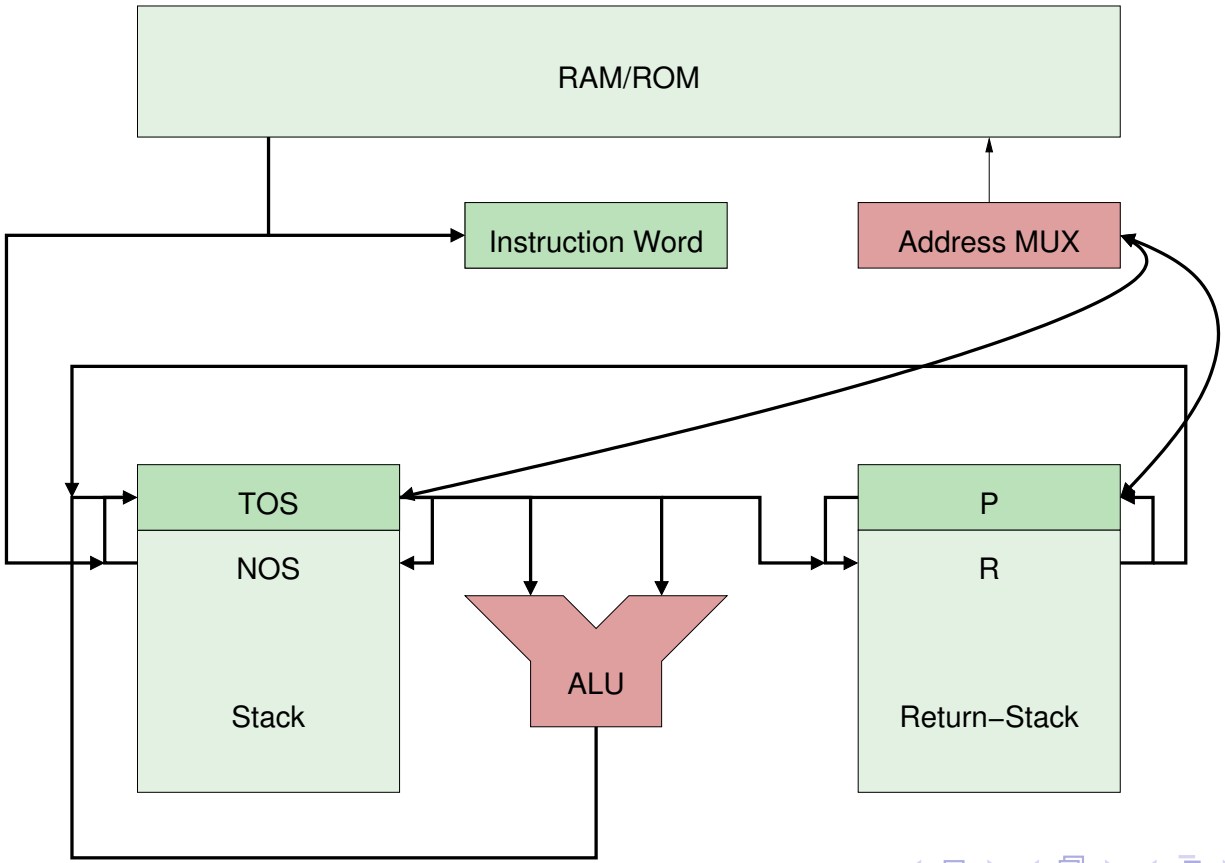
Abstraktion Forth ist low–level, man kann aber auch high level oder alles dazwischen implementieren: Unnötige Abstraktionen vermeiden, nötige selber machen

Ressourcensparend Forth ist kompakt, passt auch in ein paar kB Speicher

b16 Blockschaltbild

Der b16 ist etwa 400 Zeilen Verilog, da ist ein Debugger via UART dabei

b16 Blockschaltbild



b16 Befehlssatz

Bei 5 Bit pro Befehl passen 3 plus 1 Bit in ein 16–Bit–Wort.

b16-small Befehlssatz

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

Sprungbefehle

Sprungbefehle

nop	(—)
call	(— r:P) $\mathrm{P} \leftarrow jmp$; $\mathrm{c} \leftarrow 0$
jmp	(—) $\mathrm{P} \leftarrow jmp$
ret	(r:a —) $\mathrm{P} \leftarrow a \wedge \$\mathrm{FFFE}$; $\mathrm{c} \leftarrow a \wedge 1$
jz	(n —) $\mathbf{if}(n = 0)\,\mathrm{P} \leftarrow jmp$
jnz	(n —) $\mathbf{if}(n \neq 0)\,\mathrm{P} \leftarrow jmp$
jc	(x —) $\mathbf{if}(c)\,\mathrm{P} \leftarrow jmp$
jnc	(x —) $\mathbf{if}(c = 0)\,\mathrm{P} \leftarrow jmp$

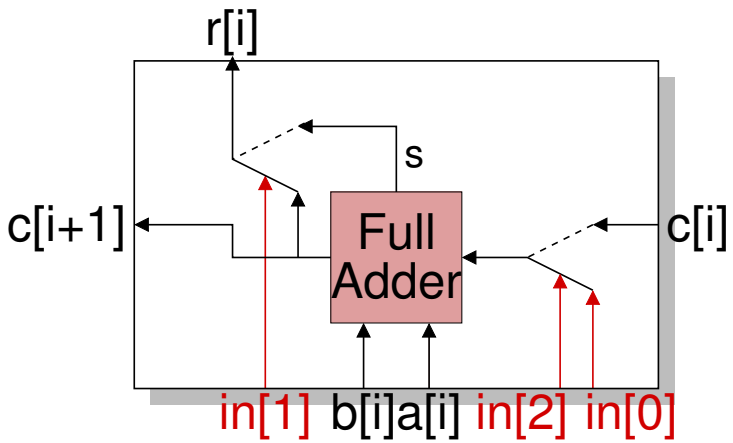
ALU–Operationen

ALU–Operationen

xor	(a b — r) $r \leftarrow a \oplus b$
com	(a — r) $r \leftarrow a \oplus \$FFFF$, $c \leftarrow 1$
and	(a b — r) $r \leftarrow a \wedge b$
or	(a b — r) $r \leftarrow a \vee b$
+	(a b — r) $c, r \leftarrow a + b$
+c	(a b — r) $c, r \leftarrow a + b + c$
2/	(a — r) $r, c \leftarrow r[15], r$
c2/	(a — r) $r, c \leftarrow c, r$

ALU Implementierung

ALU element



Speicherbefehle b16

Speicherbefehle b16

!+	(n A — A')	$mem[A] \leftarrow n;\ \mathrm{A}' \leftarrow \mathrm{A} + 2$
@+	(A — n A')	$n \leftarrow mem[\mathrm{A}];\ \mathrm{A}' \leftarrow \mathrm{A} + 2$
@	(A — n)	$n \leftarrow mem[\mathrm{A}];$
lit	(— n)	$n \leftarrow mem[\mathrm{P}];\ \mathrm{P} \leftarrow \mathrm{P} + 2$
c!+	(c A — A')	$mem.b[\mathrm{A}] \leftarrow c;\ \mathrm{A}' \leftarrow \mathrm{A} + 1$
c@+	(A — c A')	$c \leftarrow mem.b[\mathrm{A}];\ \mathrm{A}' \leftarrow \mathrm{A} + 1$
c@	(A — c)	$c \leftarrow mem.b[\mathrm{A}];$
litc	(— c)	$c \leftarrow mem.b[\mathrm{P}];\ \mathrm{P} \leftarrow \mathrm{P} + 1$

Stackoperationen b16

Stackoperationen b16

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

Beispiele (b16)

Timer mit einem Inline–Argument setzen

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

3 aus 4–Redundanz, nutzt sehr viel Stack

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

Beispiele (b16)

Timer mit einem Inline–Argument setzen

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

3 aus 4–Redundanz, nutzt sehr viel Stack

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

Was gibt's für Tools

- Assembler
- Umbilical debugger (download and run code) mit GUI
- Software-Simulator

Was gibt's für Tools

- Assembler
- Umbilical debugger (download and run code) mit GUI
- Software-Simulator

Was gibt's für Tools

- Assembler
- Umbilical debugger (download and run code) mit GUI
- Software-Simulator

James Bowman: J1

- Der J1 ist etwa 200 Zeilen Verilog, extrem einfach
- Jedes 16–Bit–Wort ein Befehl, weniger kompakter Code als b16. Inspiriert von RTX2000
- Für ein „Internet of Things“–Produkt gemacht (Netzwerkkamera eines Roboters), Netzwerkcode ist frei verfügbar
- Als Demo gibt es auch Space Invaders (über VGA)

James Bowman: J1

- Der J1 ist etwa 200 Zeilen Verilog, extrem einfach
- Jedes 16–Bit–Wort ein Befehl, weniger kompakter Code als b16. Inspiriert von RTX2000
- Für ein „Internet of Things“–Produkt gemacht (Netzwerkkamera eines Roboters), Netzwerkcode ist frei verfügbar
- Als Demo gibt es auch Space Invaders (über VGA)

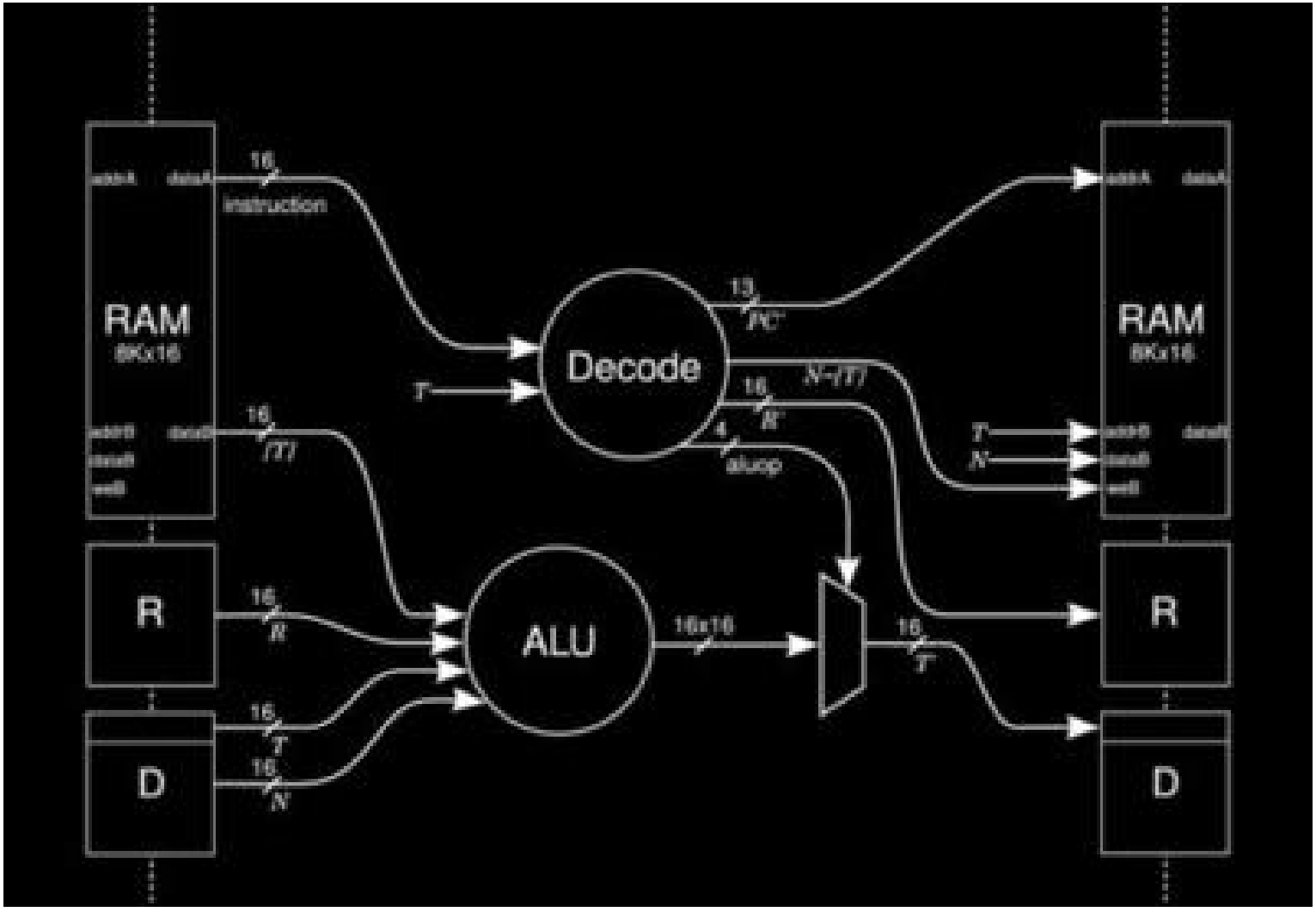
James Bowman: J1

- Der J1 ist etwa 200 Zeilen Verilog, extrem einfach
- Jedes 16–Bit–Wort ein Befehl, weniger kompakter Code als b16. Inspiriert von RTX2000
- Für ein „Internet of Things“–Produkt gemacht (Netzwerkkamera eines Roboters), Netzwerkcode ist frei verfügbar
- Als Demo gibt es auch Space Invaders (über VGA)

James Bowman: J1

- Der J1 ist etwa 200 Zeilen Verilog, extrem einfach
- Jedes 16–Bit–Wort ein Befehl, weniger kompakter Code als b16. Inspiriert von RTX2000
- Für ein „Internet of Things“–Produkt gemacht (Netzwerkkamera eines Roboters), Netzwerkcode ist frei verfügbar
- Als Demo gibt es auch Space Invaders (über VGA)

J1


RAM
8Kx16
instruction
Decode
T
PC'
N~T'
R'
aluop
T'
N'
RAM
8Kx16
R
ALU
16x16
R
D
T
N
D

Klaus Schleisiek: MicroCore

- MicroCore ist in VHDL, komplexer als J1 und b16
- Harward–Architektur: Befehlsspeicher und Datenspeicher sind getrennt, und der Datenspeicher ist in der Wortgröße frei konfigurierbar
- Verschiedene Projekte von Send

Klaus Schleisiek: MicroCore

- MicroCore ist in VHDL, komplexer als J1 und b16
- Harward–Architektur: Befehlsspeicher und Datenspeicher sind getrennt, und der Datenspeicher ist in der Wortgröße frei konfigurierbar
- Verschiedene Projekte von Send

Klaus Schleisiek: MicroCore

- MicroCore ist in VHDL, komplexer als J1 und b16
- Harward–Architektur: Befehlsspeicher und Datenspeicher sind getrennt, und der Datenspeicher ist in der Wortgröße frei konfigurierbar
- Verschiedene Projekte von Send

Andrew Read: N.I.G.E.

- N.I.G.E. ist auch VHDL, auch komplexer als J1 und b16. Ziel: Ein komplett selbst designter Home Computer
- Mehrere Takte pro Befehl
- Der Focus liegt auf Homecomputer, nicht auf embedded Controller

Andrew Read: N.I.G.E.

- N.I.G.E. ist auch VHDL, auch komplexer als J1 und b16. Ziel: Ein komplett selbst designter Home Computer
- Mehrere Takte pro Befehl
- Der Focus liegt auf Homecomputer, nicht auf embedded Controller

Andrew Read: N.I.G.E.

- N.I.G.E. ist auch VHDL, auch komplexer als J1 und b16. Ziel: Ein komplett selbst designter Home Computer
- Mehrere Takte pro Befehl
- Der Focus liegt auf Homecomputer, nicht auf embedded Controller

Links I

Bernd Paysan

b16

https://bernd-paysan.de/b16.html

James Bowman

J1

http://www.excamera.com/sphinx/fpga-j1.html

Klaus Schleisiek

MicroCore mirror

http://xilinx.pe.kr/_hdl/2/_ip/-microcore.org/index.html/

Andrew Read

N.I.G.E. github repository

https://github.com/Anding/N.I.G.E.-Machine