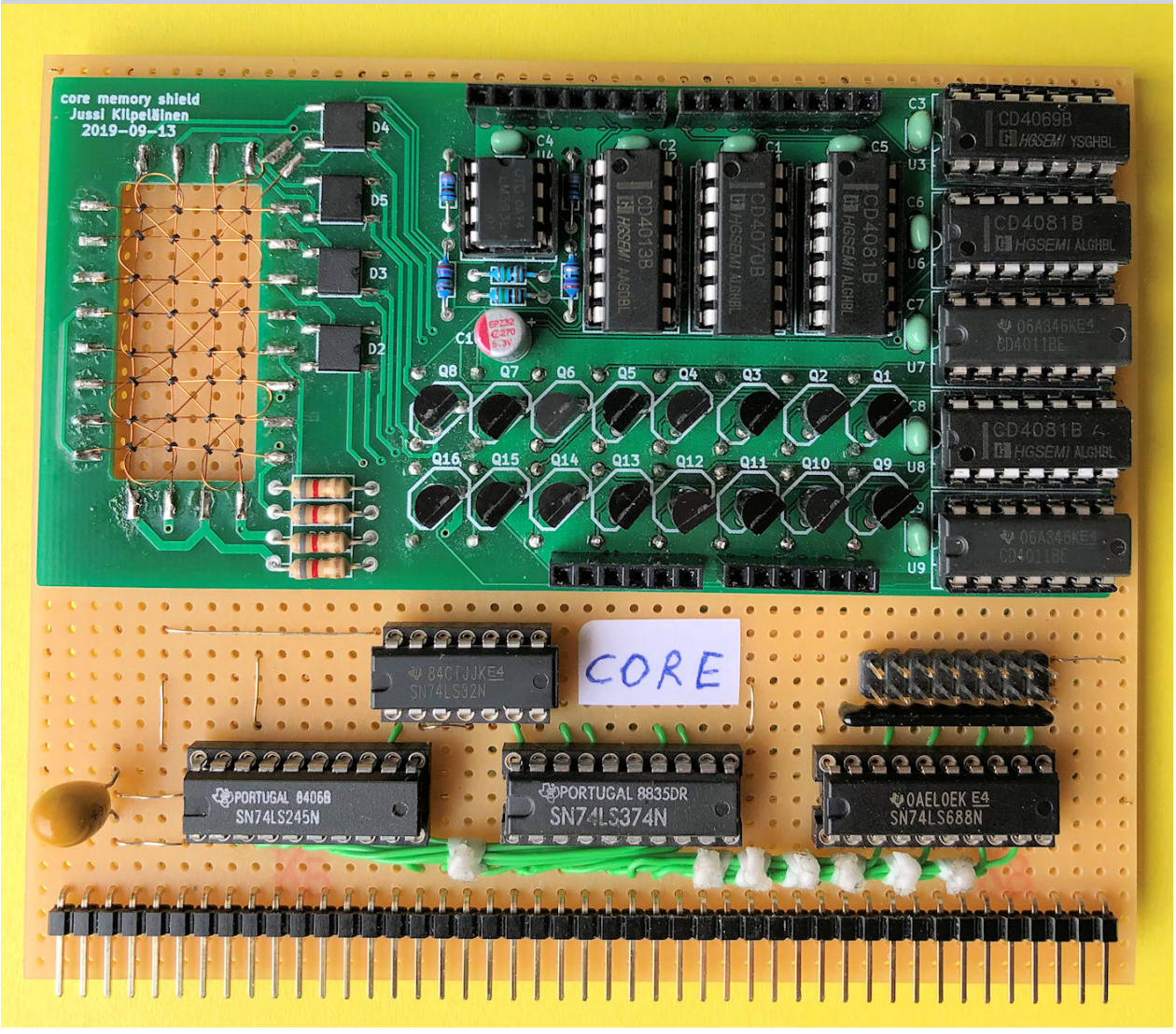
Spezifikation

CORE
Der Ringkernspeicher



Version 1.0

Idee:

Sascha Neuschl
Pirolweg 21
48167 Münster
Email: scn69@gmx.de

Dokumentenhistorie

Version	Autor(en)	Änderung	Datum
1.0	Neuschl, Sascha	Erste Version	11.06.2022

Inhaltsverzeichnis

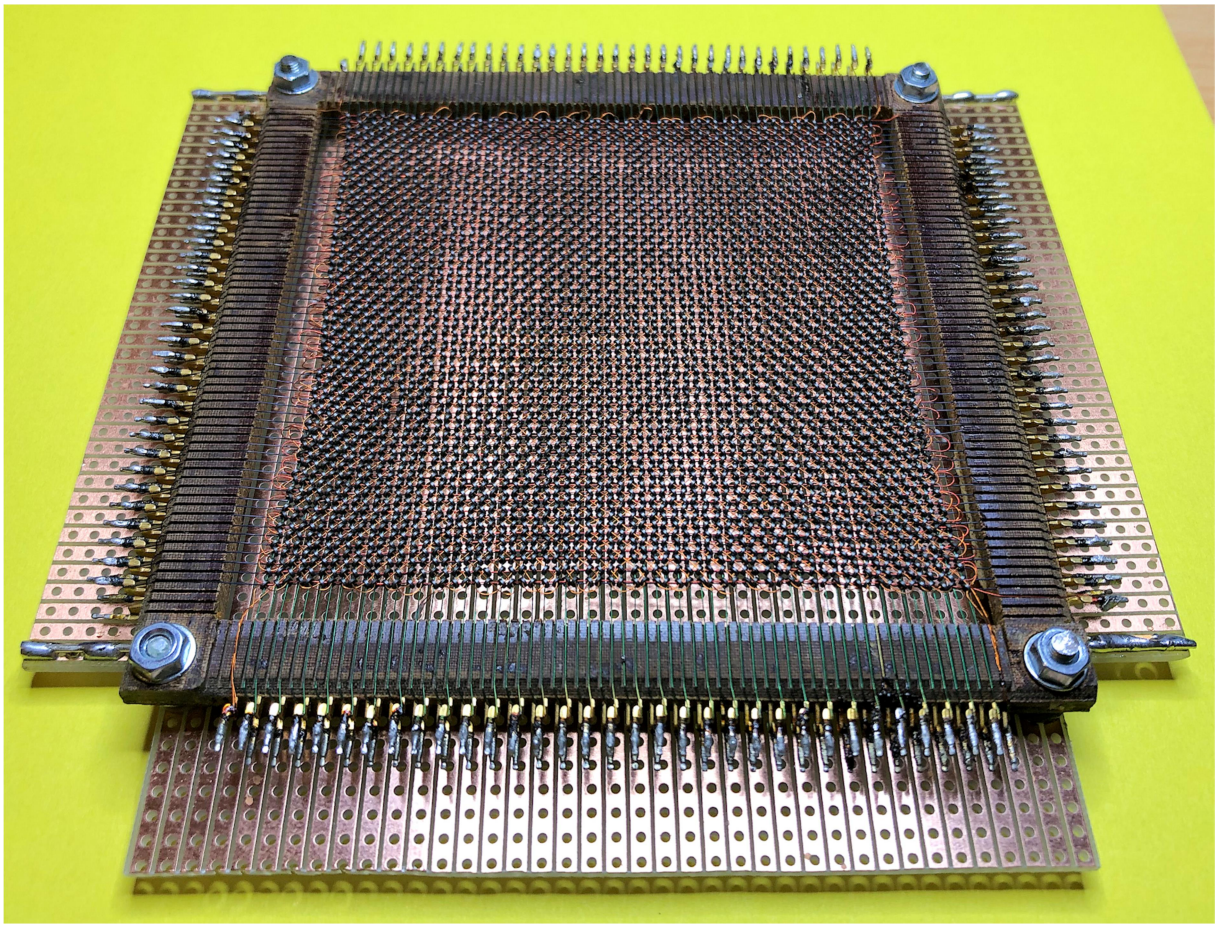
1 Vorwort

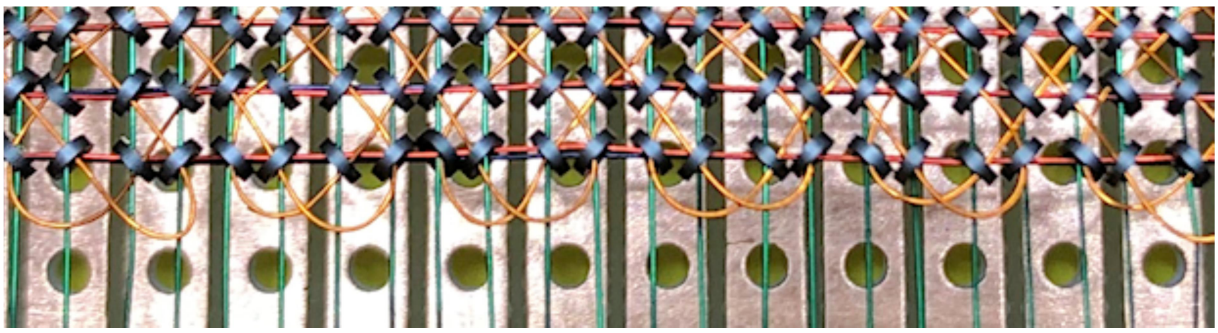
1.1 Idee

Während meines Studiums schenkte mir ein Freund damals einen Einschub mit Ringkernspeichern. Die fand ich damals schon interessant - wusste ich doch, dass der Computer in der Kapsel der ersten Mondmission der NASA auch mit solchen Speichern ausgerüstet war, um gegen die Höhenstrahlung resistent zu sein. Und da der NKC schon retro ist fand ich, dass ihm dieses Stück lebendiger Speichergeschichte noch fehlt.

1.2 Ansatz

Ich schaute mich bei ebay um und erwarb einen echten, alten Ringkernspeicher:





Ich las einiges im Web über den Betrieb dieser Speicher und schaute mich auch schon nach Bauteilen zur Ansteuerung um. Aber irgendwie schwante es mir, dass das – zumindest im Moment – dann doch eine zu große Aufgabe war.

Abhilfe schaffte an dieser Stelle das **core momory shield für einen arduino von Jussi Kilpeläinen** (https://jussikilpelainen.kapsi.fi/wordpress/?p=213). Man kann dort einen Bausatz mit Platine und Bauteilen erwerben. Den galt es, aufzubauen, über einen IO-Port an den NKC anzuschließen und mindestens Teile des Original C# - Codes in 6800X Assembler zu übersetzen.

1.3 Aktueller Stand

Mit dem aufgebauten Bausatz des core memory shield und dem Anschluss an einen IO-Port des NKC können per Programm alle 32 Kerne getestet und ein Byte oder Long geschrieben und gelesen werden.

2 Beschreibung des Konzepts

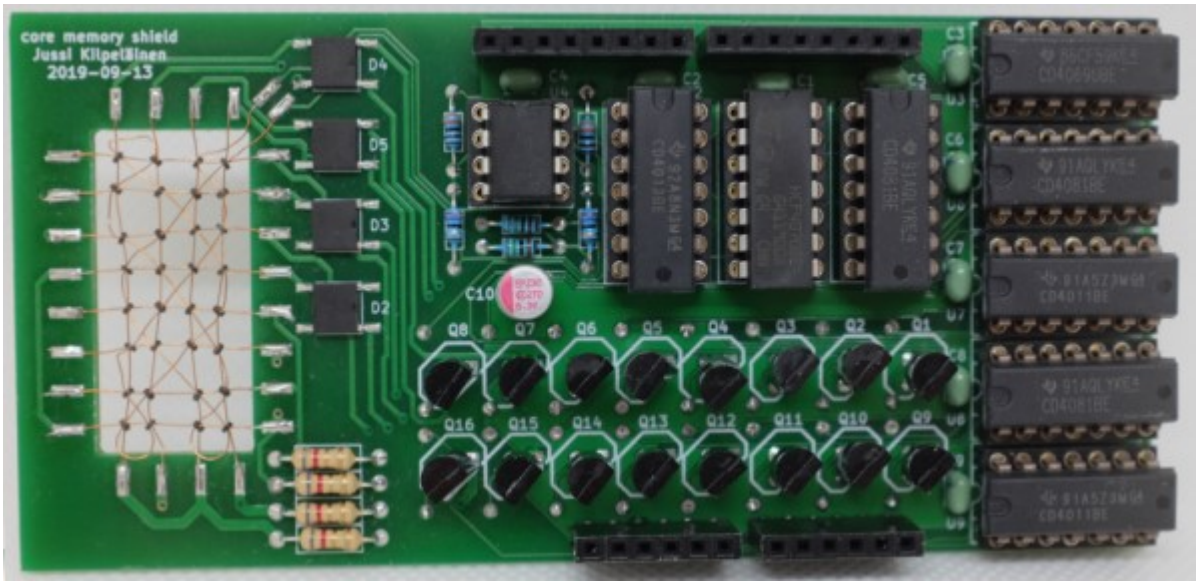
Hier wird ein Ringkernspeicher mit 32 Kernen / Bit betrieben. Die gesamte Logik zur Ansteuerung der Kerne befindet sich auf dem **core memory shield**.

Für den Anschluss an den NKC ist nur **ein IO-Port für Schreiben und Lesen** notwendig. Dieser kann explizit aufgebaut sein (wie bei mir) oder auch durch eine IOE-Karte realisiert sein.

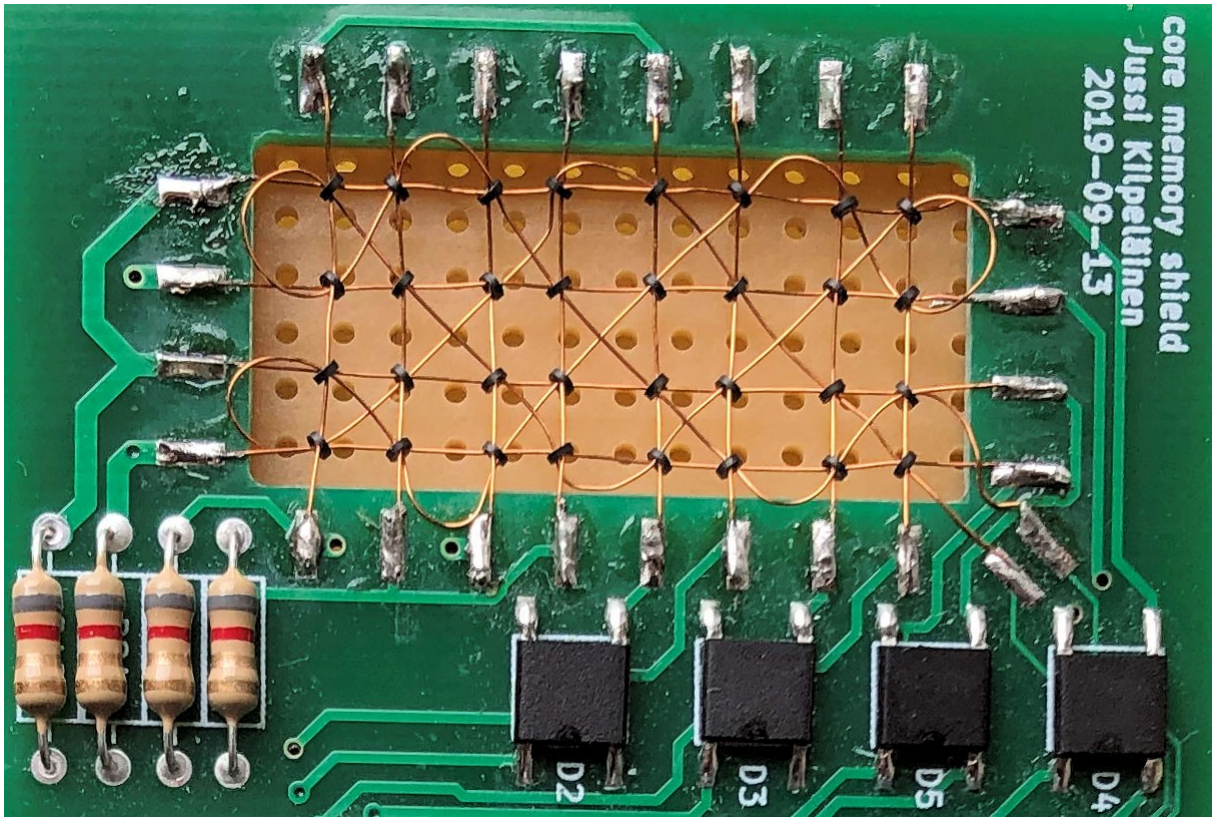
2.1 Das core memory shield

Wenn man sich den Bausatz für das core memory shield beschafft, gibt es eine Dokumentation dazu (coremem-shield.pdf) mit dem Inhalt:

- Theorie des Ringkernspeichers
- Design des Bausatzes
- Aufbauanleitung
- Benutzerhinweise
- Fehlersuche
- Anhang:
  - Schaltplan
  - Platine
  - Stückliste



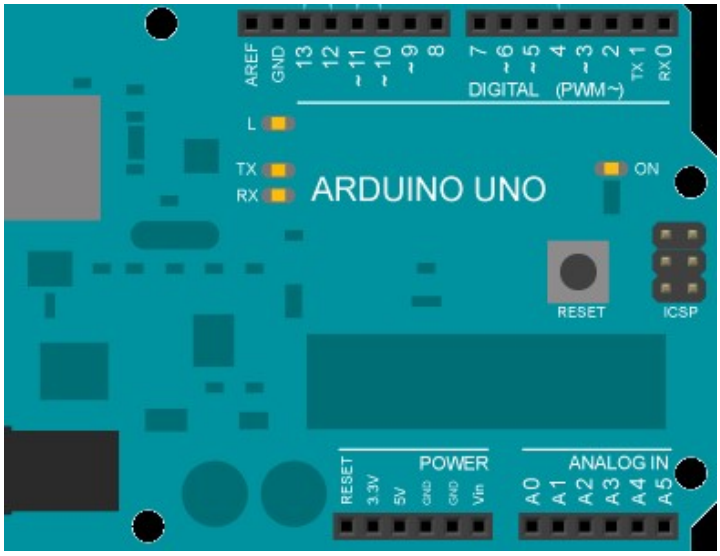
Die wesentliche Aufgabe beim Aufbau der Platine ist die Erstellung der Kernmatrix. Mit einer Lupenlampe, Pinzette und ruhigen Hand klappt es aber recht gut.



2.2 Anbindung an den NKC

Wie man das **core memory shield** an den **NKC** anschließt, wir hier gezeigt.

- Layout arduino:



- NKC Port-Bytes und Zuordnung zum arduino:

Bit	7	6	5	4	3	2	1	0	Modus
	X	K4	K3	K2	K1	K0	WR	EANABLE	**Schreiben**
	NC.	Kernadresse					Aktivieren		
arduino Pin	D7	D6	D5	D4	D3	D2	D8	D0	
	X	X	X	X	X	X	RD		**Lesen**
	NC.							Lesen	
arduino Pin	X	X	X	X	X	X	X	D9	

- Spannungsversorgung:
  - +5 Volt an arduino Pin Power +5V
  - 0 Volt an arduino PINs Power GND und GND – links von digital 13

2.3 Funktionsweise

Die **Funktionsweise eines Kernspeichers** ist einmal in der Dokumentation des Bausatzes des core memory shields (coremem-shield, pdf) und in der Dokumentation des Grundlagenprojekts zum core memory shield von Ben North und Oliver Nash (coremem.pdf) nachzulesen.

Die **Funktionsweise des core memory shields** findet sich in der Dokumentation des Bausatzes

Die **Funktionsweise bei der Ansteuerung durch den NKC** ist wie folgt:

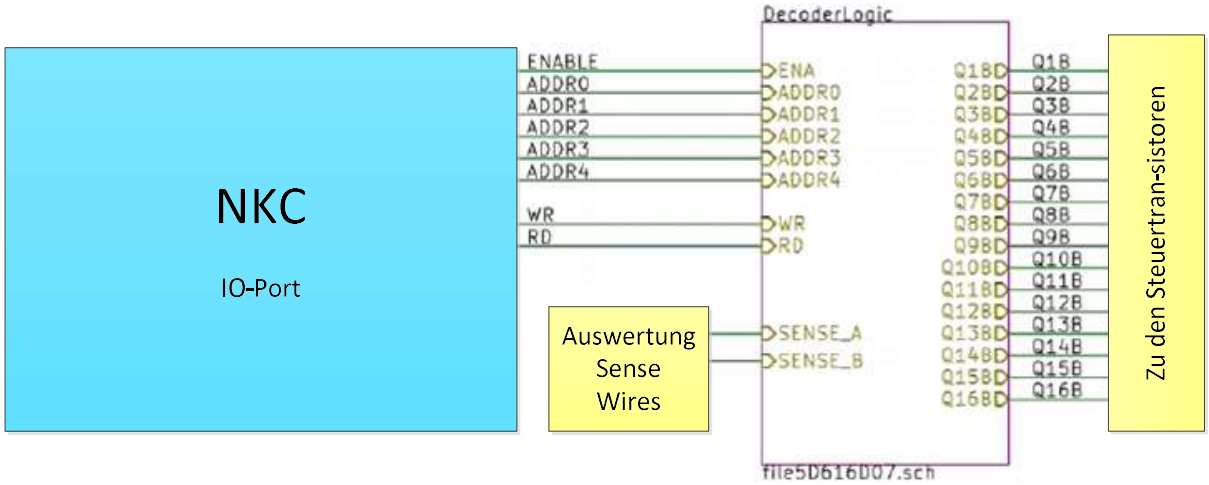
- **Schreiben eines Kerns:**
  - Zuerst wird die gewünschte Kernadresse (Bit 6-2) in das Portbyte geschrieben
  - Danach wird WR (Bit 1) zum Portbyte hinzugefügt :
    - Mit WR = 1 wird ein Kern/Bit gesetzt
  - Danach wird ENABLE = 1 (Bit 0) gesetzt:
    - ENABLE = 1 wird für mindesten 3 µs angelegt
  - Zum Schluss wird ENABLE = 0 für 25 µs angelegt

- **Lesen eines Kerns:**
  - Zuerst wird die gewünschte Kernadresse (Bit 6-2) in das Portbyte geschrieben
  - Danach wird WR (Bit 1) zum Portbyte hinzugefügt :
    - Mit WR = 0 wird ein Kern/Bit gelöscht
  - Danach wird ENABLE = 1 (Bit 0) gesetzt:
    - ENABLE = 1 wird für mindesten 3 µs angelegt
  - Zum Schluss wird ENABLE = 0 für 25 µs angelegt
  - Dann wird RD (Bit 0) ausgelesen:
    - **Wenn RD = 1**, dann gab es bei dem Kern **eine Statusänderung**. Er besaß den Wert „1“ und besitzt jetzt den Wert „0“. D.h. das **Leseergebnis für den Kern ist „1“**. Durch das Auslesen des Kerns ist dessen Wert aber „0“, und deshalb muss der Kern danach wieder mit „1“ geschrieben werden. **Das ist typisch für Kernspeicher.**
    - **Wenn RD = 0**, dann gab es bei dem Kern **k e i n e Statusänderung**. Er besaß den Wert „0“ und besitzt jetzt den Wert „0“. D.h. das **Leseergebnis für den Kern ist „0“**.

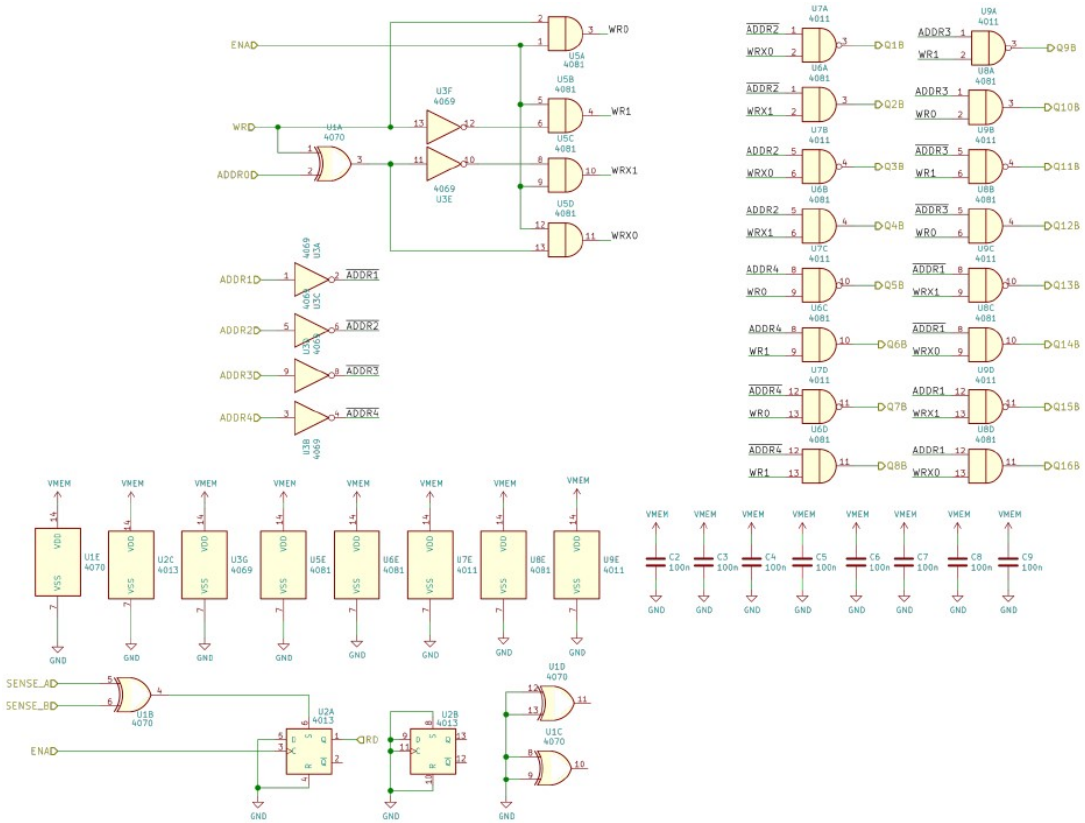
3 Die Schaltung

Die Schaltung der NKC-Core-Platine besteht aus der Schaltung für den **IO-Port des NKC**, der Dekoder-Logik des core memory shield zur Adressierung der Kerne, der eigentlichen **Ansteuerung der Kerne** und der **Auswertung der Sense Wires**, ob eine Änderung eines Kernstatus stattgefunden hat.

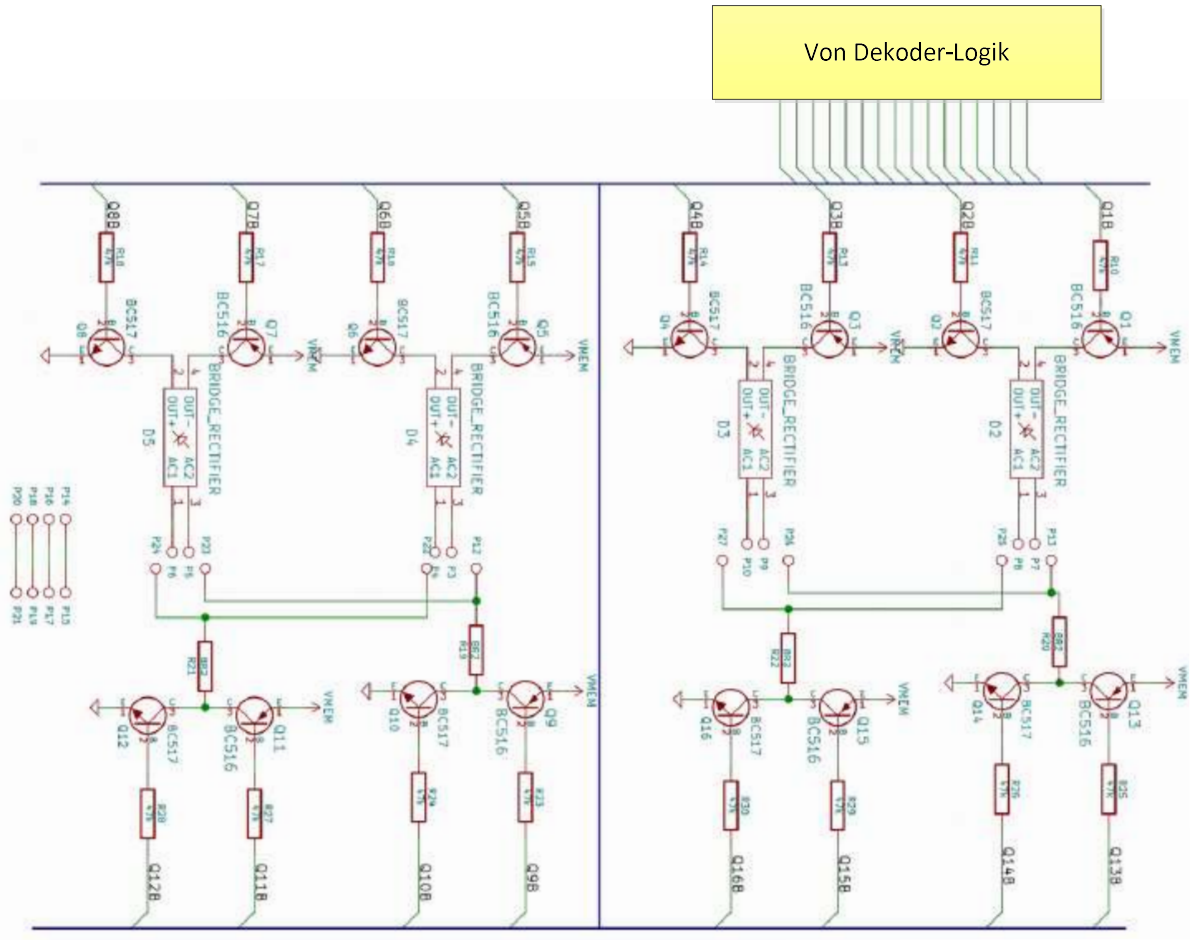
3.1 Überblick



3.2 Dekoder-Logik

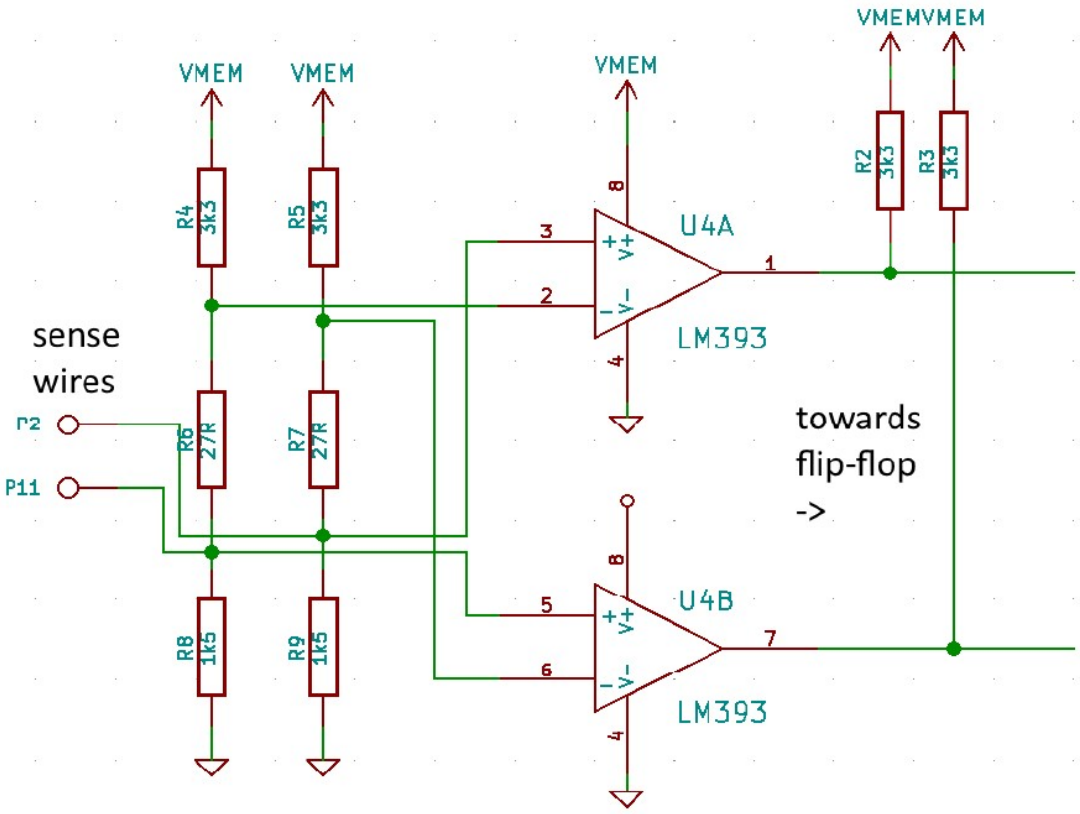


3.3 Ansteuerung der Kerne



3.4 Auswertung Sense Lines

Wenn auf **den Sense Wires eine Spannung von 25 mV** auftritt, zeigt das die **Änderung des Kernstatus von „0“ auf „1“ oder umgekehrt** an. Es wird dann ein Flipflop gesetzt, dass dann über RD vom NKC ausgelesen wird. Dieses Flipflop wird jeweils bei Auslösen von ENABLE = 1 zunächst zurückgesetzt.



4 Stückliste

Die **Stückliste des core memory shield** befindet sich in Dokumentation zu dem Bausatz (coremem-shield.pdf). Für die **Realisierung des NKC IO-Ports** auf einer Platine wird folgendes benötigt:

<u>mechanisch</u>:	
Lochrasterplatine 12 cm X 10 cm	1 Stück
Stiftleiste gewinkelt – 45 polig	1 Stück
Stiftleiste gerade, zweireihig – 8 polig	1 Stück
IC-Fassung – 14-polig	1 Stück
IC-Fassung – 20-polig	3 Stück
<u>Kondensatoren</u>:	
10 µF/16 V	1 Stück
<u>ICs</u>:	
74LS32	1 Stück
74LS245	1 Stück
74LS374	1 Stück
74LS688	1 Stück

5 Anwendung der NKC CORE Platine

5.1 Programme

Mit den folgenden Programmen kann man folgendes tun:

- Einen Kern/Bit schreiben und lesen
- Ein Byte schreiben und lesen
- Ein Long schreiben und lesen
- Alle 32 Kerne testen und Ausgabe, welche Kerne funktionieren und welche nicht

```
*******************************************************************************
*                                    NKC CORE                                 *
*******************************************************************************
* Ringkernspeicher mit 32 Bit fuer den NKC auf der Basis des Projekts von      *
* Ben North und Oliver Nash Copyrigth 2011 / Jussi Kilpeläinen Copyright 2019  *
*******************************************************************************
* Uebersetzt in 68008 Assembler und Anpassung an einen NKC IO-Port durch sEn   *
* Sascha Neuschl 2022-06-11                                                    *
*******************************************************************************
* - Es wird 1 IO-Port verwendet (hier $FFFFFFFF)                               *
* - Schreiben = 7       6       5       4       3       2       1       0      *
*               NC      ---------- Kernadresse ----------       WRITE   ENABLE *
*                                                                              *
* - Lesen =     7       6       5       4       3       2       1       0      *
*               NC      NC      NC      NC      NC      NC      NC      READ   *
*******************************************************************************
* Version 1.0 / 11.06.2022                                                     *
*  - Testen aller Kerne, Schreiben und lesen Byte / Long                       *
*                                                                              *
*******************************************************************************


*** Konstanten

;* IO-Port
port            equ     $FFFFFFFF

DISABLE         equ     %11111110
ENABLE          equ     %00000001
DWRITE          equ     %00000010
```

```
;* Wartezeit fuer Schreiben eines Kerns beim Ein- und Ausschalten
;* 1 NOP = 1,5 us bei 68008, 2 wait states und 8 Mhz Takt

WRONW  equ  1           ;NOPs (3 us als default)
WROFFW equ  8           ;NOPs = (25 us als default)

;* Zeichen

kette:                  dc.b 'aBcD', 0  ;fuer Test Long
corokay:                dc.b 'Kerne okay:',0
cornokay:               dc.b 'Kerne  n o t  okay:',0

*** Variblen

coreno:         ds.b 1
value:          ds.b 1
buffer:         ds.b 15
byte:           ds.b 1
long:           ds.b 4

ds 0


;* Einen Kern = Bit schreiben
;* Wert in value (0,1) und Kernadresse in coreno (0..31)
;* Port Byte: Bit 7 = nc, 6-2 = core no, 1 = Write, 0 = ENABLE

wrcore:
 move.b coreno, d0              ;Kernnummer fuer Port Byte
 rol.b #2, d0
 cmp.b #0, value                ;Kern setzen oder loeschen
 beq.s do                       ;Loeschen durch Lesen
 or.b #DWRITE, d0               ;Setzen durch Schreiben
 do:
  or.b #ENABLE, d0              ;Aktivieren
  move.b d0, port               ;Port Byte mit ENABLE
  move #wronw-1, d2             ;Zeitdauer ENABLE
  writeon:
   nop
  dbra d2, writeon
  and.b #DISABLE,d0             ;Deaktivieren
  move.b d0, port               ;Port Byte ohne ENABLE
```

```
  move #wroffw-1, d2            ;Zeitdeuer -ENABLE
  writeoff:
   nop
  dbra d2, writeoff
rts


;* Einen Kern = Bit lesen
;* Wert in value (0,1) und Kernadresse in coreno (0..31)
;* Port Byte - schreiben: Bit 7 = nc, 6-2 = core no, 1 = WR, 0 = ENABLE
;* Pot Byte - lesen:      Bit 7-1 = nc, Bit 0 = RD (Aenderung Kernzustand)

rdcore:
 clr.b value            ;Schreiben einer "0" in den Kern ... und schauen, ob es
                        ;eine Aenderung am Kernzustand gab
 bsr wrcore
 move.b port, d1        ;Lesen, ob eine Aenderung am Kernzustand aufgetreten ist
                        ;In diesem Fall von "1" zu "0"!
 and.b #%00000001, d1
 cmp.b #0, d1
 beq.s notset           ;Kern haelt "0"
                        ;Aenderung Kernzustand - Kern haelt "1" - Daten in d1
 move.b #1, value       ;Nach dem Auslesen von "1" wieder "1" schreiben
 bsr wrcore
 notset:                ;K e i n e  Aenderung Kernzustand - Kern haelt "0"
                        ;Daten in d1
rts


;* Ein Byte schreiben
;* Wert in byte und Startkernadresse in coreno

wrbyte:
 move.b coreno, d1              ;1. Kern fuer Bit 0
 move.b byte, d3
 clr d4                         ;Schleifenzaehler
 wbit:
  add.b #1, d4
  and.b #%00000001, d3          ;Jedes Bit einzeln in value laden
  move.b d3, value
  move.b d1, coreno             ;Kernadresse
  bsr wrcore                    ;und in Kern schreiben
```

```
  add.b #1, d1                  ;naechster Kern
  move.b byte, d3               ;naechstes Bit
  ror d4, d3
  cmp.b #8, d4                  ;8 Bit = 1 Byte
  bne.s wbit
rts


;* Ein Long schreiben
;* Wert in long und Startkernadresse = 0

wrlong:
 lea kette, A0                  ;Zeichenkette mit 4 Zeichen in d0.l uebertragen
 adda.l #3, A0                  ;Ende der Zeichenkette
 move.b (A0), d3
 move #3-1, d4
 wrreg:
  rol.l #8, d3
  suba.l #1, A0
  move.b (A0), d3
 dbra d4, wrreg
 move.l d3, long

 clr.b d4                       ;Schleifenzaehler und 1. Kern
 wlbit:
  and.l #%00000000000000000000000000000001, d3
  move.b d3, value              ;jedes Bit einzeln in value ...
  move.b d4, coreno             ;Kernadresse
  bsr wrcore                    ;und in Kern schreiben
  add #1, d4                    ;Schleifenzaehler und naechster Kern
  move.l long, d3               ;naechstes Bit
  ror.l d4, d3
  cmp.b #32, d4                 ;32 Bit = 1 Long
bne.s wlbit
rts


;* Ein Byte lesen
;* Wert in byte und Startkernadresse in coreno
;* Ausgabe in byte

rdbyte:
```

```
 clr.b d3                       ;Initialisierung byte
 clr d4                         ;Schleifenzaehler
 move.b coreno, d5              ;1. Kern
 rbit:
  move.b d5, coreno             ;Kernadresse
  bsr rdcore                    ;und lesen
  cmp.b #0, d1                  ;Kern/Bit gesetzt?
  beq.s zero                    ;Bit = 0
  bset d4, d3                   ;Passendes Bit in d3 setzen
  zero:
  add.b #1, d5                  ;naechster Kern
  add #1, d4                    ;Schleifenzaehler
  cmp.b #8, d4                  ;8 Kerne/Bit = 1 Byte
 bne.s rbit
 move.b d3, byte                ;Ausgabe in byte
 lea buffer, A0                 ;Schreiben des Zeichens auf den Bildschirm
 move.b d3, (A0)+
 move.b #0, (A0)
 lea buffer, A0
 move.b #$23, d0
 move #50, d1
 move #100, d2
 moveq #!write, d7
 trap #1
rts


;* Ein Long lesen
;* Wert in byte und Startkernadresse in coreno
;* Ausgabe in long

rdlong:
 clr.l d3                       ;Initialisierung Datenlangwort
 clr.b d4                       ;Schleifenzaehler und 1. Kern
 rlbit:
  move.b d4, coreno             ;Kernadresse
  bsr rdcore
  cmp.b #0, d1                  ;Kern/Bit gesetzt?
  beq.s lzero                   ;Bit = 0
  bset d4, d3                   ;Passendes Bit in d3 setzen
  lzero:
  add #1, d4                    ;Schleifenzaehler und naechster Kern
```

```
  cmp.b #32, d4                 ;32 Kerne/Bit = 1 Long
 bne.s rlbit
 move.l d3, long               ;Ausgabe in long

 lea buffer, A0                ;Schreiben der 4 Zeichen auf den Bildschirm
 move #4-1, d4
 print:
  move.b d3 , (A0)+
  ror.l #8, d3
 dbra d4, print
 move.b #0, (A0)               ;Pufferabschlus
 lea buffer, A0
 move.b #$23, d0
 move #50, d1
 move #100, d2
 moveq#!write, d7
 trap #1
rts


;* Schreiben und Lesen eines Bytes
corebyte:
 move.b #'*', byte             ;Zeichen = "*"
 move.b #8, coreno
 bsr wrbyte
 move.b #8, coreno
 bsr rdbyte
rts


;* Schreiben und Lesen eines Longs
corelong:
 bsr wrlong
 bsr rdlong
rts


;* Test, ob sich alle Kerne schreiben und lesen lassen
;* Jeder Kern wird mit "1" geschrieben und ausgelesen
;* Wird eine "1" gelesen ist der Kern okay, sonst n o t  okay
;* Kerne , die okay sind, werden oben ausgegeben, Kerne, die  n o t  okay
;* sind, werden unten ausgegeben
```

```
coretest:
 clr d4                          ;Schleifenzaehler und 1.Kern
 clr d5                          ;X-Komponente Ausgabe "Kerne okay"
 clr d6                          ;X-Komponente Ausgabe "Kerne  n o t  okay"
 lea corokay, A0                 ;Ueberschrift Kerne okay
 move.b #$21, d0
 clr d1
 move #180, d2
 moveq #!write, d7
 trap #1
 lea cornokay, A0                ;Ueberschrift Kerne okay
 move.b #$21, d0
 clr d1
 move #80, d2
 moveq #!write, d7
 trap #1

 loop:
  move.b d4, coreno              ;Kernadresse
  move.b #1, value               ;Schreiben einer "1"
  bsr wrcore                     ;Kern schreiben
  bsr rdcore                     ;Kern lesen
  cmp.b #1, d1                   ;Kern/Bit gesetzt?
  bne.s notokay                  ;Kern/Bit = 0
  lea buffer, A0                 ;Ausgabe Kerne okay
  move.b d4, d0                  ;Kernadresse
  moveq #!print4d, d7            ;Wandlung in ASCII
  trap #1
  lea Buffer, A0                 ;Ausgabe auf Bildschirm
  move.b #$11, d0
  move d5, d1
  add #15, d5
  move #150, d2
  moveq #!write, d7
  trap #1
  bra.s next                     ;naechster Kern
  notokay:
   lea buffer, A0
   move.b d4, d0                 ;Kernadresse
   moveq #!print4d, d7           ;Wandlung in ASCII
   trap #1
   lea buffer, A0                ;Ausgabe Kerne  n o t  okay
```

```
   move.b #$21, d0
   move d6, d1
   add #15, d6
   move #50, d2
   moveq #!write, d7
   trap #1
  next:
   add #1, d4
   cmp.b #32, d4                ;Alle 32 Kerne
   bne loop
rts
```

6 Anhang

6.1 Quellennachweis:

Komponente	Datei
core memory shield von Jussi Kilperläinen	coremem-shield.pdf
Grundlagenprojekt zum core memory shield von Ben North und Oliver Nash	Coremem.pdf