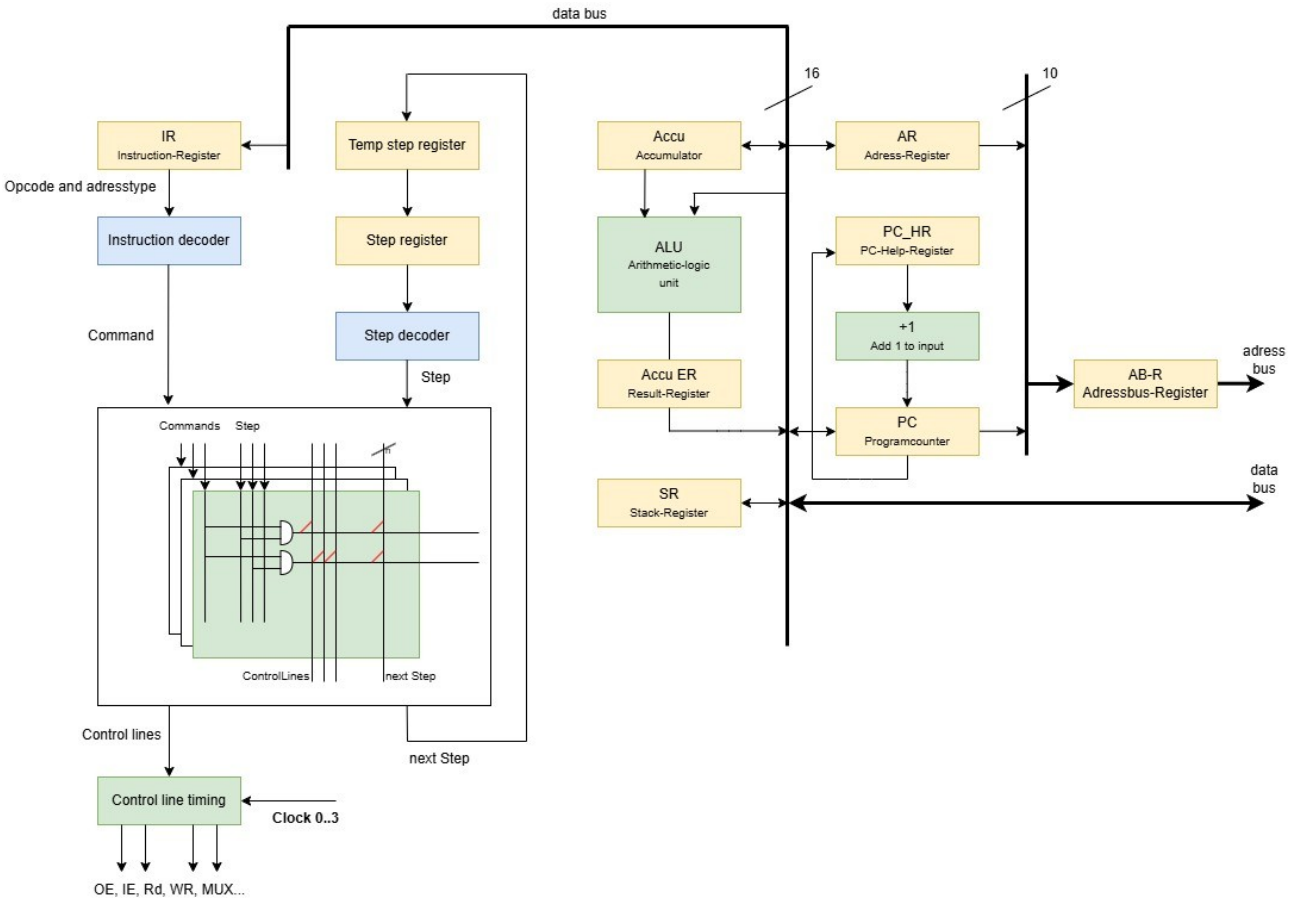
Struktur des Relay-Systems



Der linke Block ist das Steuerwerk, eine State Machine (ganz besonders crazy war hier der Mechanismus um das Instruction Register neu laden zu können ohne dass die ganze Statemachine ihren Anker verliert, nur nebenbei). Rechts das Rechenwerk mit dem einzigen für den Programmierer wichtigen Register, dem Accu (alles 16 bit). Die anderen Register werden im Zuge der Befehle verwendet, aber nie explizit angesprochen um Werte darin speichern zu können.

Du vermisst vielleicht ein Status-Register mit den Flags, gibt es eingeschränkt. Dazu kommt noch ein Kapitel.

Am Adress und Datenbus hängt dann ein Arduino der Speicher spielt, er liest Read und Write Signale, den Adressbus und den Datenbus und legt entweder Daten auf den Datenbus oder speichert das im Datenbus im Speicher.

Der Datenbus hat 16 bit, der Adressbus 10 bit.

Wie sieht ein Programm für den Relaisrechner aus

```
daten[0] = 0xF000;          // NOP

daten[1] = 0x1400;          // LDA
daten[2] = 118;

daten[3] = 0x2800;          // STAM
daten[4] = 116;

daten[5] = 0x1800;          // LDAM
daten[6] = 110;

daten[7] = 0x2C00;          // STAI
daten[8] = 116;

daten[9] = 0x1800;          // LDAM
daten[10] = 116;

daten[11] = 0x3440;         //ADD
daten[12] = 0x0001;
```

Einfach ein int16 Array im Arduino. Des Rest der SW des Arduino checkt die ganze Zeit Read und Write und spielt Speicher, aber das daten[] Array ist der eigentliche Speicher des Systems. Man sieht: NOP hat nur den Opcode, LDA hat den Opcode und im nächsten Word dann den Wert den man in den Accu laden möchte, also LDA 118 lädt dezimal 118 in den Accu. Du siehst auch: ich hab schön chaotisch programmiert, manchmal hex, manchmal dezimal.

Die Befehle in der Übersicht:

A: Accu-register, M(X) Speicherstelle X, X: Konstante

LDA X	**X → accu**
LDA M(X)	**M(X) → accu**
LDA M(M(X))	**M(M(X)) → accu**
STA M(X)	**accu → M(X)**
STA M(M(X))	**accu → M(M(X))**
NOP	**do nothing, load next command**
ADD X	**A = A + X**
ADD M(X)	**A = A + M(X)**
ADD M(M(X))	**A = A + M(M(X))**
AND X	**A = A & X**
AND M(X)	**A = A & M(X)**
AND M(M(X))	**A = A & M(M(X))**
OR X	**A = A | X**
OR M(X)	**A = A | M(X)**
OR M(M(X))	**A = A | M(M(X))**
NOT A	**A = not(A)**
SHL A	**A = SHL(A)**
SHR A	**A = SHR(A)**
CALL M(X)	**SR = PC, PC = M(X)**
RET	**PC = SR**
BZ M(X)	**A == 0? → PC = M(X)**
BN M(X)	**A<0? → PC = M(X)**
BC (MX)	**Carry == 1? PC = M(X)**
BA (MX)	**PC = M(X)**
BE (MX)	**LSB(Accu) == 0? PC = M(X)**

Im wesentlichen sind es Befehle zu laden/speichern des Accu, mathematische Operationen (+, and, or, not, shift left, shift right), Sprungbefehle, Unterprogrammbefehle mit einer Ebene (später mehr) und der NOP. Subtraktion mittels 2er Komplement.

Die grau hinterlegten wären vorbereitet (der Decoder hat die Befehle) aber sind noch nicht implementiert. Sie sind nicht zwingend notwendig und ich kann sie umgehen, aber ich werde sie noch einbauen, zumindest die M(X) Varianten.

Statt ADD M(X) kommt folgende Befehlskette zum Einsatz:

daten[n]: LDAM
daten[n+1]: X
daten[n+2]: STAM
daten[n+3]: Y+1

dann eine beliebige Anzahl von Befehlen…

daten[y]: ADD

daten[Y+1]: irgendein Wert, 0, der ja überschrieben wird

Ich umgehe also die fehlenden Befehle einen mathamatische Operation mit was anderen als einer Konstanten direkt nach dem Opcode zu machen indem ich mit einem Load/Store Block die Konstante mit einem Wert überschreibe...kostet Performance und das entsprechende Programm könnte in einem ROM nicht laufen da es sich selber adaptiert.

Die unterschiedlichen Adressierungsarten werden durch Bits im Opcode spezifiziert, genauso wie die gewünschte mathematische Operation wie auch die Sprungbedingung.

Das Carry-Flag wird gelöscht, wenn das Ergebnisregister der ALU gelöscht wird. Dies geschieht bei einer ALU-Operation, bevor das Ergebnisregister mit dem Rechenergebnis geladen wird. Und gesetzt bei einer mathematischen Operation mit Übertrag (Addition, nicht beim shift)

Bei einem NOT-Befehl wird der Wert aus dem Speicher (X, M(X) oder M(M(X))) zwar geladen, aber ignoriert.

Die Flags N, Z und E werden per Logik ständig aus dem Accuinhalt abgeleitet und werden als Statusleitung (abhängig von der Sprungedingung) zum Steuerwerk geschickt. Sind also vom aktuellen Accuinhalt abhängig.

CALL und RET sind eine Unterprogrammebene...die Rücksprungsadresse kommt in das Register rein.

Und <, > gibt es nicht direkt, in Zusammenarvbeit mit der KI wurde folgende Codesequenz entwickelt, diese testet a<b, a==b oder a>b:

```
daten[0]  = 0x1800;  // LDAM b
daten[1]  = 51;     //   b bei daten[101]
daten[2]  = 0x3540;  // NOT A       ; A = ~b
daten[3]  = 0;
daten[4]  = 0x2800;  // STAM 9      ; daten[9] = ~b
daten[5]  = 11;
daten[6]  = 0x1800;  // LDAM a      ; A = a
daten[7]  = 50;     //   a bei daten[100]
daten[8] = 0x3440;  // ADD 1       ; A = a – b
daten[9] = 1;
daten[10]  = 0x3440;  // ADD [9]     ; A = a + (~b)
daten[11]  = 0;       //   wird von STAM(9) mit ~b ersetzt
daten[12] = 0x4018;  // BZ → 300    ; if A==0   ⇒ a==b
daten[13] = 300;
daten[14] = 0x4008;  // BC → 100    ; if Carry ⇒ a>b
daten[15] = 100;
daten[16] = 0x4000;  // BA → 200    ; else      a<b
daten[17] = 200;

daten[50] = 31000;                  // a
daten[51] = 31000;                  // b

daten[100] = 0x1400;
daten[101] = 100;

daten[200] = 0x1400;
daten[201] = 200;

daten[300] = 0x1400;
daten[301] = 300;
```

Die Opcodes der Befehle:

Opcode				
Typ	LD	0001	IR(15)	IR(12)
4	ST	0010		
	MOP	0011		
	B	0100		
	CALL	0101		
	RET	0110		
	NOP	1111		
Adr	---	00	IR(11)	IR(10)
2	X	01		
	M(X)	10		
	M(M(X))	11		
MOP	ADD	0001	IR(9)	IR(6)
4	AND	0010		
	OR	0011		
	NOT	0101		
	SHR	0110		
	SHL	0111		
B	A (always)	000	IR(5)	IR(3)
3	C (carry)	001		
	N (negative)	010		
	Z (zero)	011		
	E (even)	100		

Typ	Adr	MOP	B	Res
4	2	4	3	000

Der exe-Simulator

Das exe-File ist ein Simulator, den gibt man als Parameter ein Textfile mit einem Relais-Computer Programm mit genau der Struktur daten[0] = …, Kommentare sind erlaubt. Das Ergebnis sieht dann so aus:

Idx	Cmd	PC	Accu	Addr	Data	Read	Write	Flag	Stack	Cycles
0001	4000	0000	0000	0000	0000	0000	0000	0000	0000	0004
0002	1800	000B	0010	00CA	0010	0010	0000	0000	0000	000A
0003	4018	000D	0010	0000	0000	0000	0000	0000	0000	000E
0004	3440	000F	000F	0010	FFFF	FFFF	0000	0001	0000	0012
0005	2800	0011	000F	000F	000F	0000	000F	0001	0000	0018
0006	1800	0013	0190	00CB	0190	0190	0000	0001	0000	001E
0007	4010	0015	0190	0000	0000	0000	0000	0001	0000	0022
0008	1400	0017	0000	0018	0000	0000	0000	0001	0000	0026
0009	2800	0019	0000	0000	0000	0000	0000	0001	0000	002C
000A	4000	001B	0000	0000	0000	0000	0000	0001	0000	0030
000B	1800	0021	0190	00CB	0190	0190	0000	0001	0000	0036
000C	35C0	0023	0320	0000	0000	0000	0000	0000	0000	003A
000D	2800	0025	0320	0320	0320	0000	0320	0000	0000	0040
000E	1800	0027	0000	00C9	0000	0000	0000	0000	0000	0046
000F	35C0	0029	0000	0000	0000	0000	0000	0000	0000	004A

Der Excel-Simulator:

Ist einfach ein Excel.Makro, hab ich dann später kaum mehr verwendet.

Display:

Die letzten 64 Words des Adressraums bilden ein memory mapped Display ab. Die erste Adresse ist rechts oben. Es gilt:

daten[960] = oberste Zeile, rechte 16 Pixel daten[961] = oberste Zeile, linke 16 Pixel daten[962] = zweite Zeile, rechte 16 Pixel daten[963] = zweite Zeile, linke 16 Pixel ... daten[1022] = unterste Zeile, rechte 16 Pixel daten[1023] = unterste Zeile, linke 16 Pixel

und

Und innerhalb jedes 16-Bit-Wortes wächst die Bitnummer von rechts nach links:

```
Bit 15 ............................... Bit 0
links                                   rechts
```

Damit ergeben sich beispielsweise:

```
Pixel (31,0), ganz rechts oben -> daten[960] = 0x0001
Pixel (16,0)                   -> daten[960] = 0x8000

Pixel (15,0)                   -> daten[961] = 0x0001
Pixel (0,0), ganz links oben   -> daten[961] = 0x8000
```