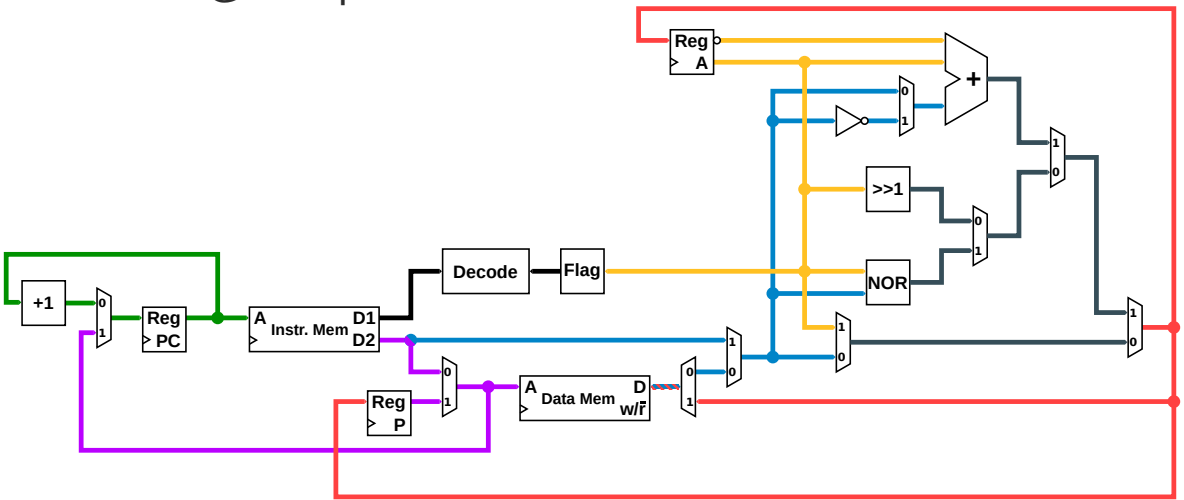
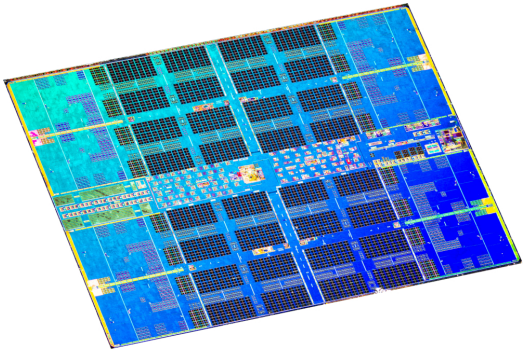
Computer selbst bauen

andi ‹andi@entropia.de›



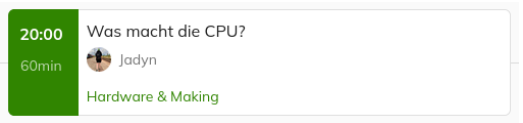
Es ist kompliziert



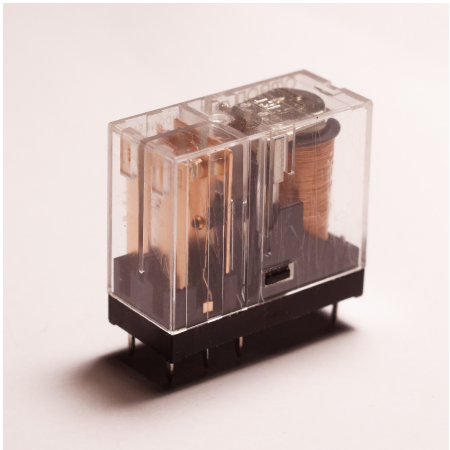
AMD Zen 2 „Matisse“

- 4 800 000 000 Transistoren
- Taktzeit:
  0.2 ns ≙ 6.4 Licht-cm ≈ 4 Strom-cm

Mehr dazu:

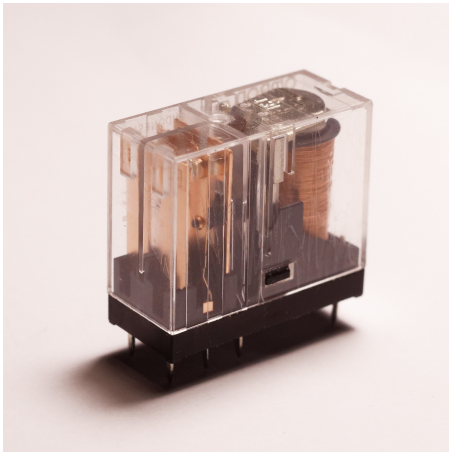


Es geht einfacher



- 120 Relais
- Takt: $\approx$ 1 s

Es geht einfacher

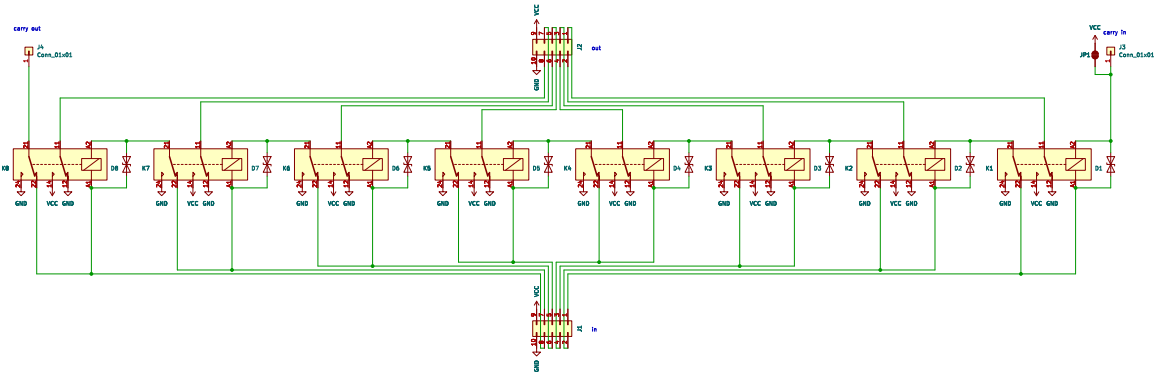


- 120 Relais
- Takt: $\approx$ 1 s

- auch binäre Schaltung (sort of)
- Kann auch Logik-Gates implementieren
- Details: ignorieren wir

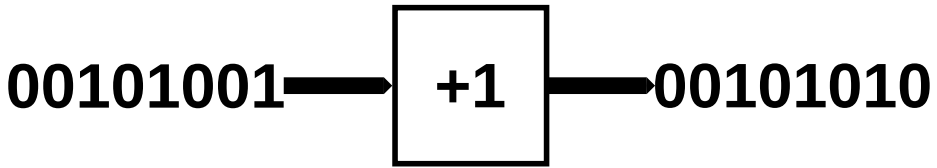
Thinking with Modules

So sieht es eigentlich aus:



Thinking with Modules

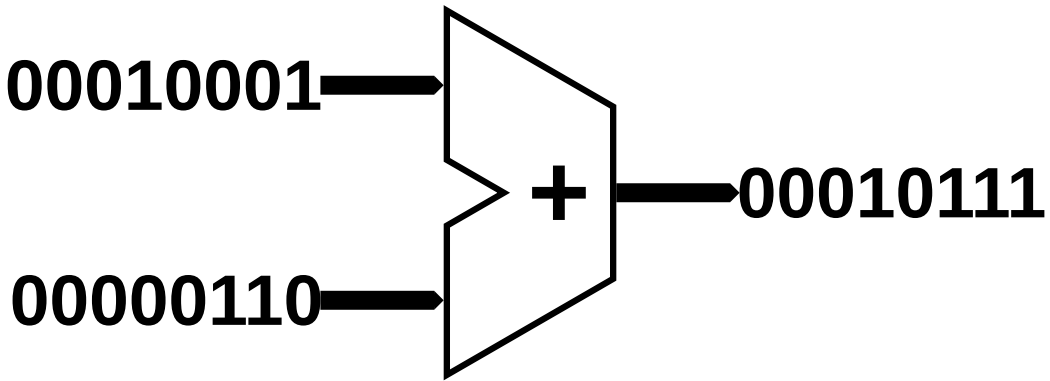
So sehen wir es:



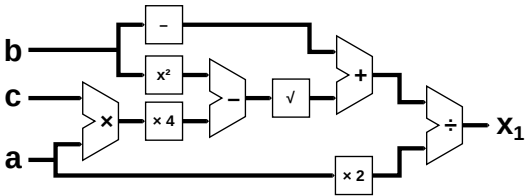
- 8 Signale auf einmal
  nur in unserem Kopf eine Zahl, Adresse, …
- Funktionalität in Module zusammengefasst
  Bits rein, Bits raus, dazwischen Magie

Rechnen ist nicht schwer

Zwei Eingänge für zwei Operanden, im Prinzip nicht schwieriger, nur mehr Hardware

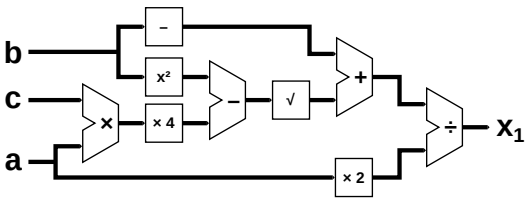


Ist das nicht schon alles?



$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Ist das nicht schon alles?



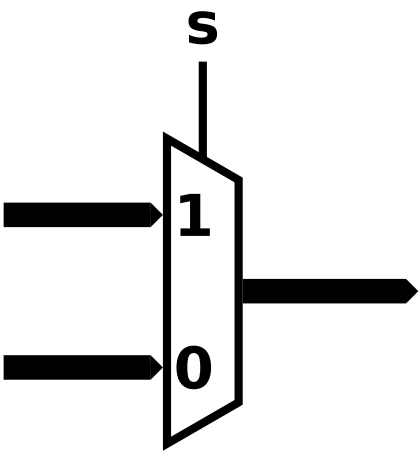
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Nein, es fehlt …

- **Auswahl**
  x_2 ohne Neuverkabeln
- **Speicher**
  a, b, c weg $\Rightarrow$ x weg :(
- **Programm**
  `goto`, `if`, `while`, `for`, `func`, …

1/3: Auswahl

Wichtigstes Modul: **Multiplexer** (Spitzname: Mux)

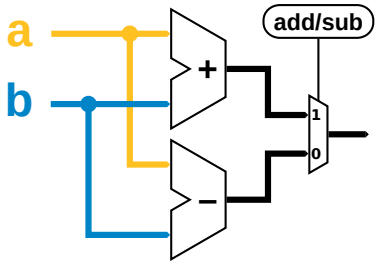


s ist nur ein Bit

- **s** $= 0$:
  Eingang 0 mit Ausgang verbunden
- **s** $= 1$:
  Eingang 1 mit Ausgang verbunden

Von zwei Bytes eins auswählen

Fast eine ALU

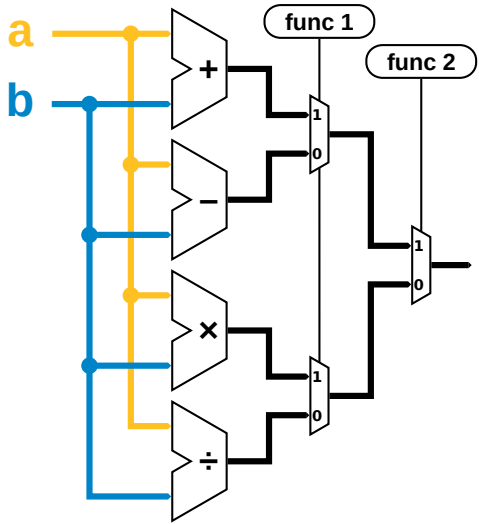


Prinzip: Erst rechnen, dann aussuchen

Kontrollsignale

- Steuern Datenfluss, bei uns vor allem an Multiplexern
- Jetzt: „Schalter“
- Später: Programmieren

Fast eine ALU



Prinzip: Erst rechnen, dann aussuchen

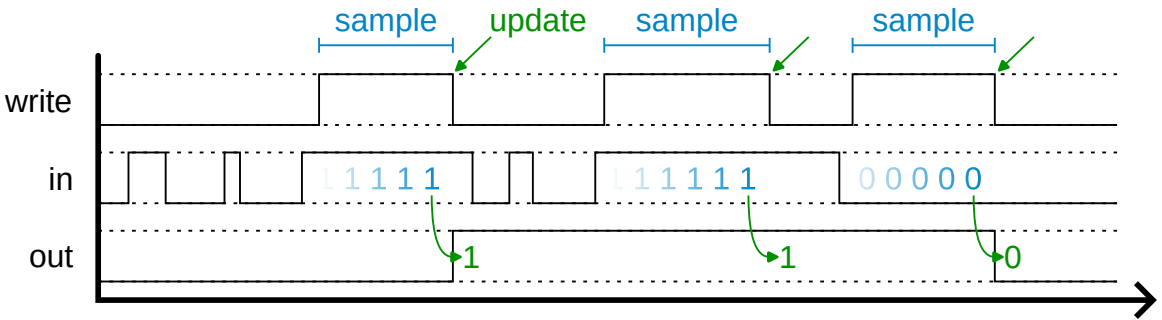
Kontrollsignale

- Steuern Datenfluss, bei uns vor allem an Multiplexern
- Jetzt: „Schalter“
- Später: Programmieren

- Mehr Möglichkeiten: Mux-Baum bauen
- nicht *ganz* realistisch

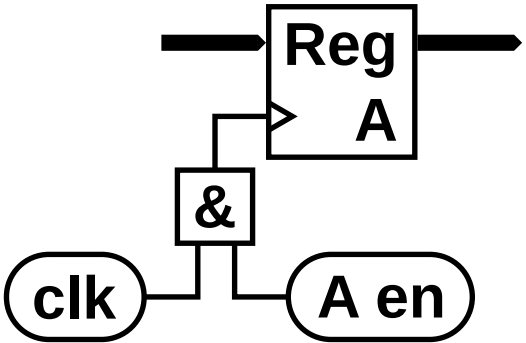
2/3: Speicher

Details ignorieren wir wieder (irgendwas mit Kondensatoren und Hysterese)



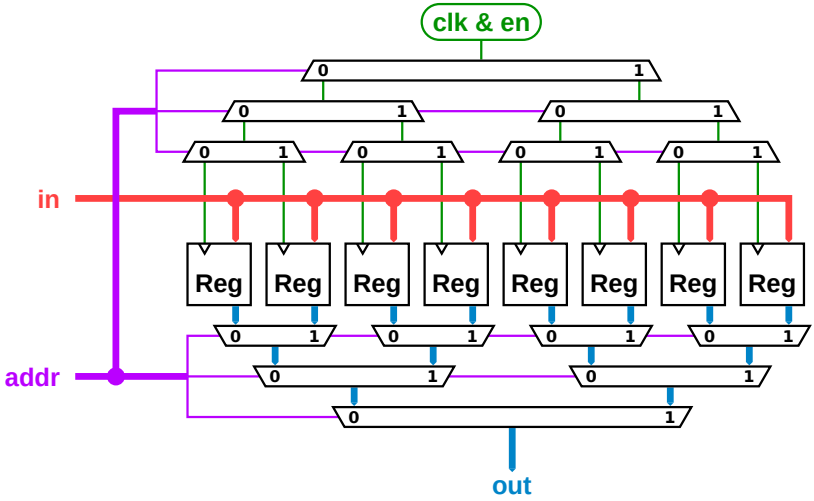
„write“-Signal wird Systemtakt/-clock sein

Speicher × 8

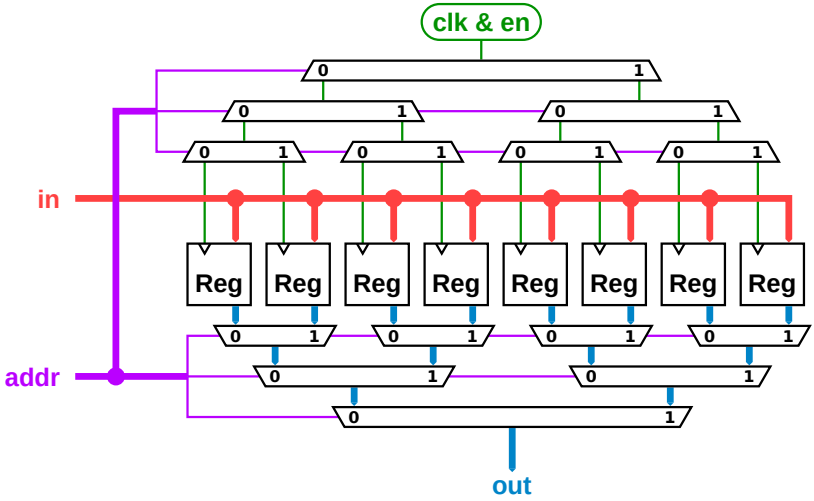


- erstes Modul mit Takt (Kennzeichnung mit ▷)
- nicht immer jedes Register überschreiben
  ⇒ *Clock-Gate*
- Kontrollsignal „A enable“

mehr Speicher



mehr Speicher



Das wird *groß*...

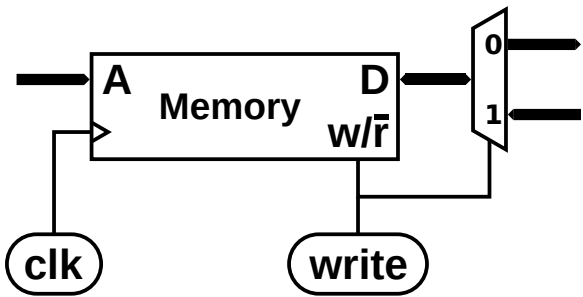
Leider gecheatet

- 1 Register $\hat{=} 5\,\text{cm} \times 10\,\text{cm}$
- 256 Register (16×16) $\hat{=} 80\,\text{cm} \times 160\,\text{cm}$
- …von den Muxen ganz zu schweigen

$\Rightarrow$ Geht nur mit ICs

RAM

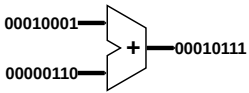
- In jedem Takt Lesen *oder* Schreiben
- Daten-Anschluss ist bidirektional ⇒ etwas verdächtiger Mux



Jetzt geht’s los

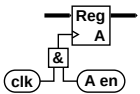
Alles beisammen:

1. Berechnungen

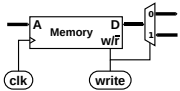


2. Multiplexer

3. Register



1. Speicher



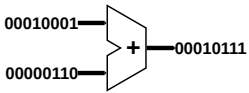
5. ???

6. ~~Profit~~ Prozessor

Jetzt geht’s los

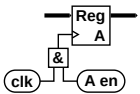
Alles beisammen:

1. Berechnungen

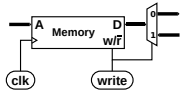


2. Multiplexer

3. Register



1. Speicher



5. ??? ⬅ **Sie befinden sich hier**

6. ~~Profit~~ Prozessor

Akkumulator-Architektur

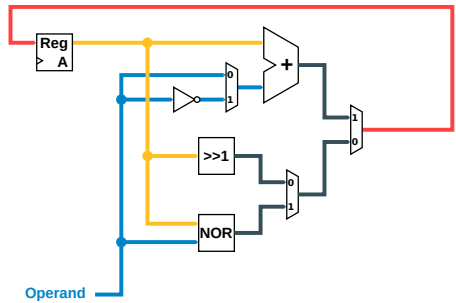
- Instruktion = Operation + Operand
- *ein einziges* Register (A)
- alle Berechnungen „A = A ⊛ Speicher“
- ein Speicherzugriff pro Instruktion ✓

`z = x + y;` wird zu

```
load  &x    A = x;
add   &y    A = A + y;
store &z    z = A;
```

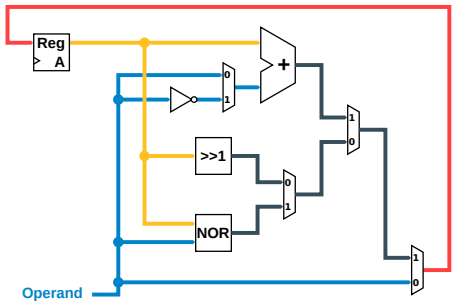
ALU-Aufbau: Rechnungen

ALU + Write-Back in den Akkumulator



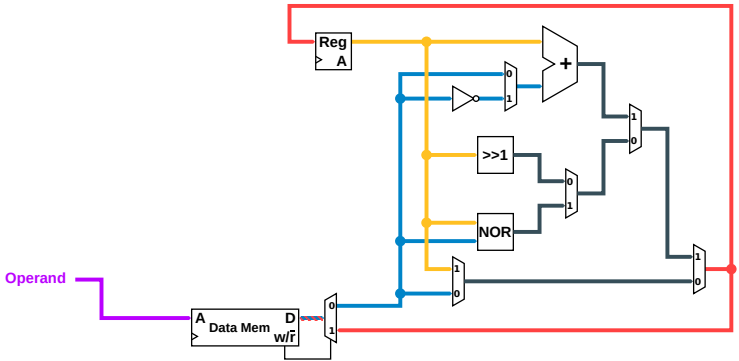
ALU-Aufbau: `load`

Ein Weg an der ALU vorbei um Daten in A zu laden



ALU-Aufbau: `store`

Operanden kommen aus dem Speicher, Ergebnisse müssen dahin zurück



Immediates

Konstanten sind auch wichtig:

- `answer = secret + 19;`
- `start = 0;`
- `i++`

In Assembler „Immediate“. Operand als Wert, nicht als Adresse

```
add 19  addi 19
load 0  loadi 0
```

Immediates

Konstanten sind auch wichtig:

- `answer = secret + 19;`
- `start = 0;`
- `i++`

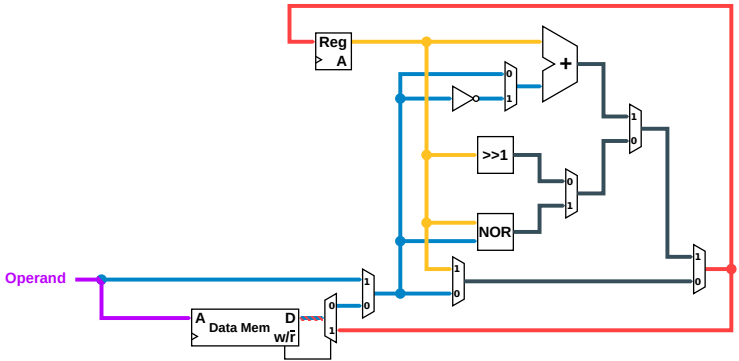
In Assembler „Immediate“. Operand als Wert, nicht als Adresse

```
add 19  addi 19
load 0  loadi 0
```

Wir brauchen noch ein Mux … (es wird nicht das Letzte sein)

Immediates

Operand direkt in die ALU geben



Pointer

Manchmal sind Adressen nicht konstant:

- `int *p = lol(); int x = *p;`
- `cpu->bits = 8;`
- `dinge[i] = d;`

Adresse in A berechnen und dann Laden/Speichern?
Leider nein: `*p = x;` braucht p und x gleichzeitig

Pointer

Manchmal sind Adressen nicht konstant:

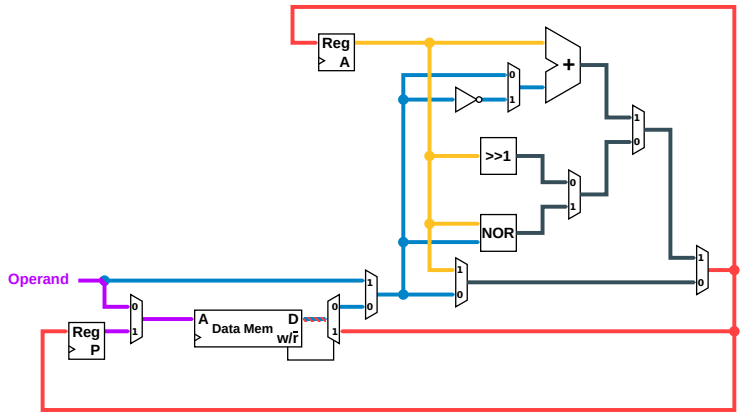
- `int *p = lol(); int x = *p;`
- `cpu->bits = 8;`
- `dinge[i] = d;`

Adresse in A berechnen und dann Laden/Speichern?
Leider nein: `*p = x;` braucht p und x gleichzeitig

Wir brauchen noch ein Mux und ein *Register* … (zum Glück das Vorletzte)

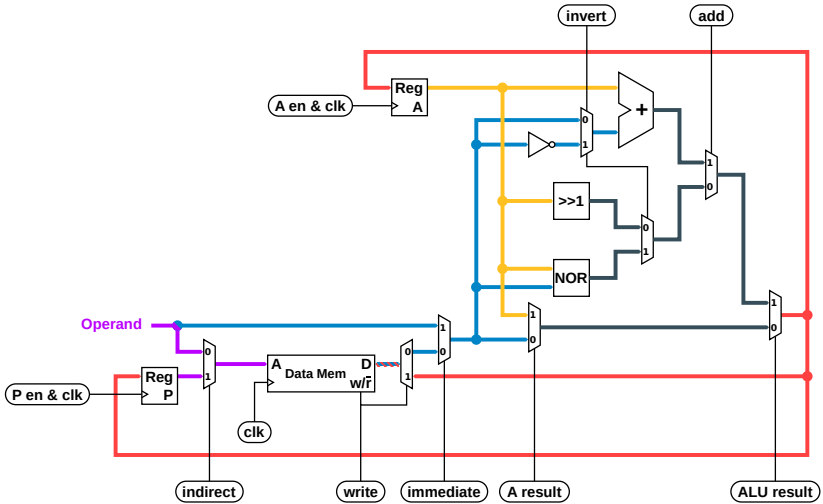
Pointer

Zweites Register mit Write-Back, Auswahl der Datenadresse



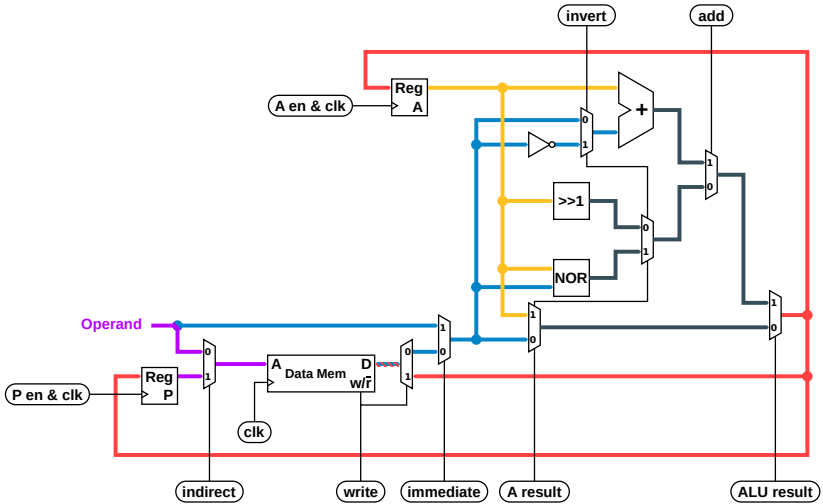
Fertig ist der *Data-Path*

9 Kontrollsignale steuern alles



Fertig ist der *Data-Path*

- 8 Kontrollsignale steuern alles



Plötzlich Maschinencode

Speicher schreiben	Reg. P schreiben	Reg. A schreiben	ALU benutzen	A lesen —— „invertieren“	Addierer benutzen	Immediate	Indirekt

- Eine Mux-Einstellung $\hat{=}$ ein Bit $\Rightarrow$ 8 in einem Byte zusammenfassen
- z.B. `add`: A schreiben, Ergebnis von der ALU, Addierer benutzen, sonst nichts
  $\Rightarrow$ `add` $=$ 0b00110100 $=$ 0x34
- Zweites Byte für den Operanden: `add 42` $=$ 0x342a

Plötzlich Maschinencode

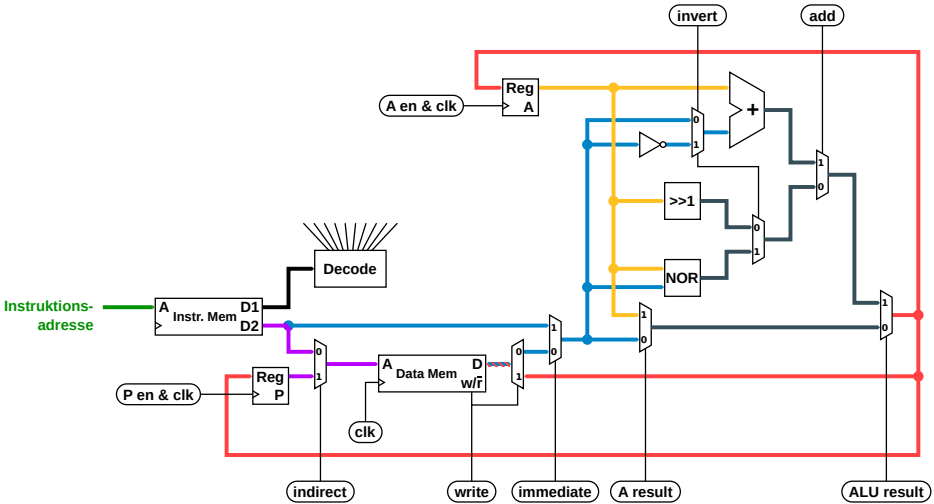
Speicher schreiben	Reg. P schreiben	Reg. A schreiben	ALU benutzen	A lesen —— „invertieren“	Addierer benutzen	Immediate	Indirekt

- Eine Mux-Einstellung $\hat{=}$ ein Bit $\Rightarrow$ 8 in einem Byte zusammenfassen
- z.B. `add`: A schreiben, Ergebnis von der ALU, Addierer benutzen, sonst nichts
  $\Rightarrow$ `add` $=$ 0b00110100 $=$ 0x34
- Zweites Byte für den Operanden: `add 42` $=$ 0x342a

Eine Instruktion = 16 bit $\Rightarrow$ **Das können wir speichern!**

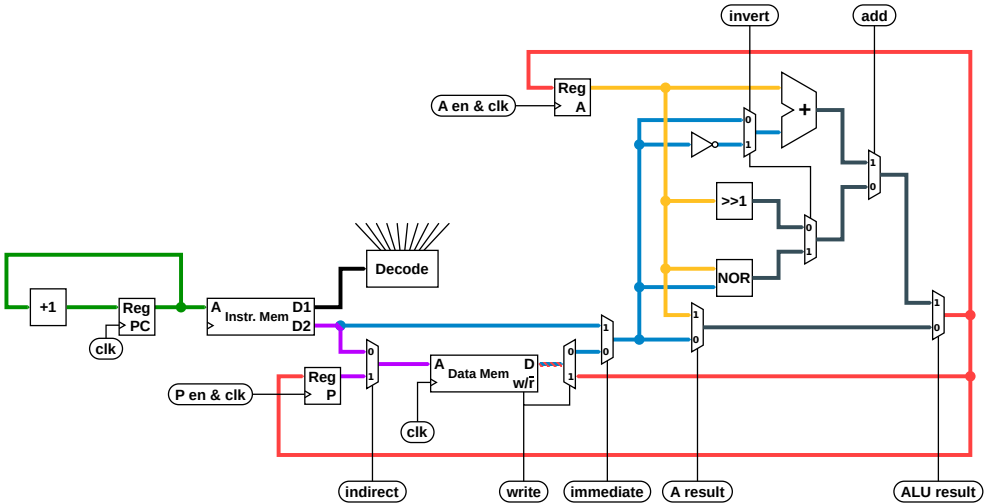
Programmspeicher

Noch ein Speicher für das Programm („Harvard“)



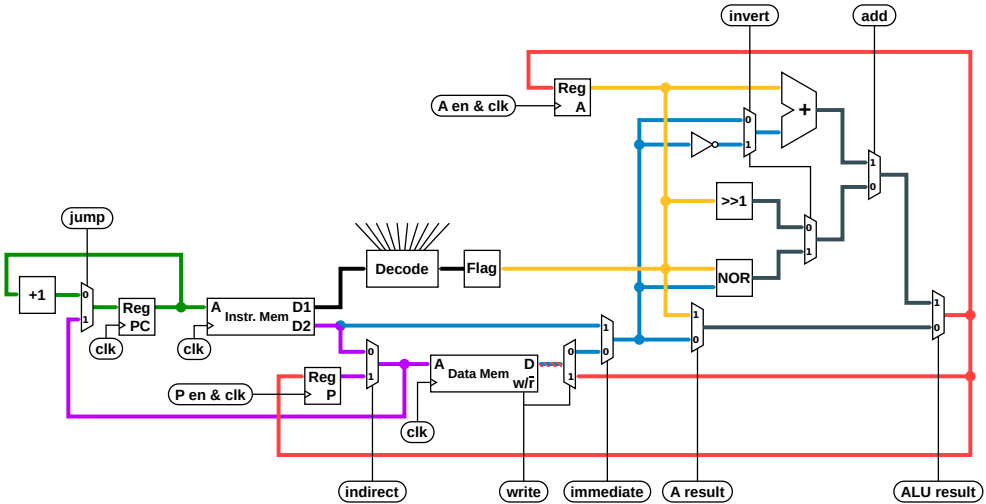
Program Counter

Wenn wir doch nur automatisch zählen könnten …



Sprünge

Noch ein Mux, die Details sind kompliziert



Klick-Klack

Was fehlt?

Was wir „Maschinencode“ genannt haben sieht sehr nach *Microcode* aus.

- Viele nutzlose Befehle: „Addiere `i` auf Inhalt der Adresse `i`“, anyone?
  `0x96 i`
- 3 Bit für Kombination aus „A schreiben“, „P schreiben“, „Speicher schreiben“
  Lieber 3 Bit für Auswahl von einer aus 8 Möglichkeiten!

Was fehlt?

Was wir „Maschinencode“ genannt haben sieht sehr nach *Microcode* aus.

- Viele nutzlose Befehle: „Addiere `i` auf Inhalt der Adresse `i`“, anyone?
  `0x96 i`
- 3 Bit für Kombination aus „A schreiben“, „P schreiben“, „Speicher schreiben“
  Lieber 3 Bit für Auswahl von einer aus 8 Möglichkeiten!

Decoding! Kleinere Assembler-Befehle intern zu Steuersignalen auspacken

Ihr könnt das auch!

und zwar einfacher und besser

- Simulationen
  - Paul Falstads Circuit Simulator
  - Logisim(-Evolution)
- FPGAs (jetzt auch in günstig und Open Source)
  - Yosys OSS CAD Suite
- Chips statt Relais

Bildnachweis

Die Shot eines AMD Zen2 https://en.wikipedia.org/wiki/File:Zen2_Matisse_Ryzen_7nm_Core_Die_shot.jpg
Fritzchens Fritz, 2019. CC-0.

Transparentes Relais https://commons.wikimedia.org/wiki/File:Omron_G2R-2-24V_relay_04.jpg
Retired electrician, 2022. CC-0.

Transparentes Relais https://commons.wikimedia.org/wiki/File:Omron_G2R-2-24V_relay_04.jpg
Retired electrician, 2022. CC-0.