Digitaltechnik & Rechnersysteme

Rechnerarchitektur II

Martin Kumm

Hochschule Fulda
University of Applied Sciences

AI | Angewandte Informatik

WiSe 2022/2023

Probeklausuren und alte Klausuren online

Probeklausuren und alte Klausuren sind nun online

Diese können als zusätzliches Lehrmaterial verwendet werden.

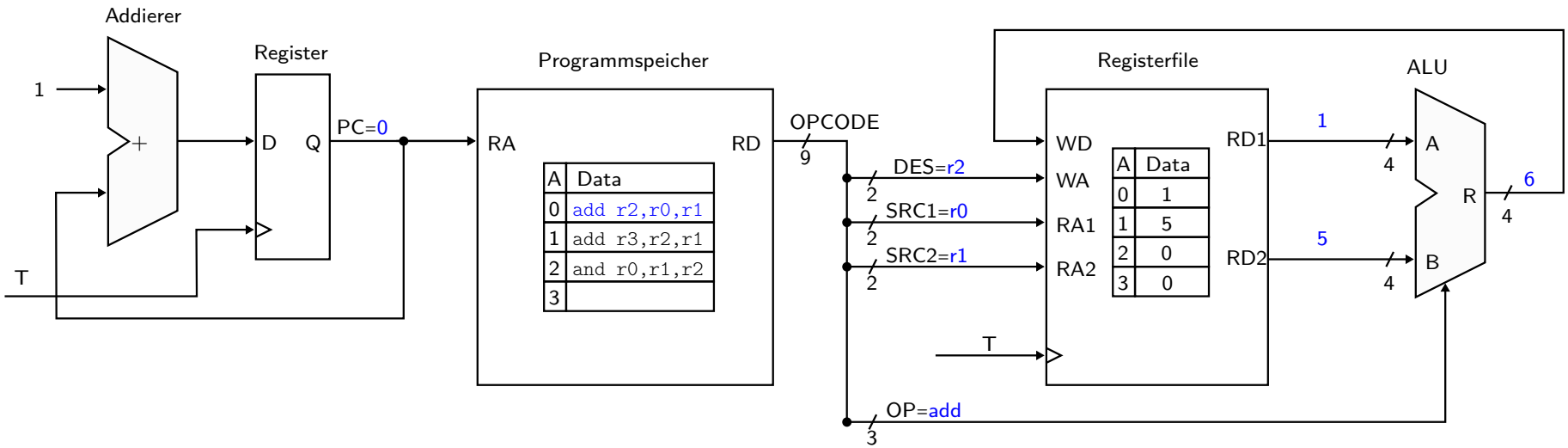
Bitte nicht selbst betrügen: Erst nach der eigenen Lösung in die Musterlösung schauen!

Was bisher geschah...

- Speicher
  - Aufteilung von Daten- und Adressleitungen: n Address- und m Datenleitungen um $2^n \times m$ Bit Speicher anzusprechen
  - Registerfile: D-FF als Speicherzelle
  - Statischer RAM (SRAM): Optimierte Speicherzelle, erlaubt statische Speicherung
  - Dynamischer RAM (DRAM): Kondensator als Speicherzelle, muss dynamisch aufgefrischt werden
- Minimal-Prozessor
  - Verbindung zwischen ALU und Registerfile erlaubt prinzipielle Befehl mit Zwischenspeicherung
  - Programmspeicher mit Programmzähler liefert in jedem Takt neuen Befehl

Minimal-Prozessor

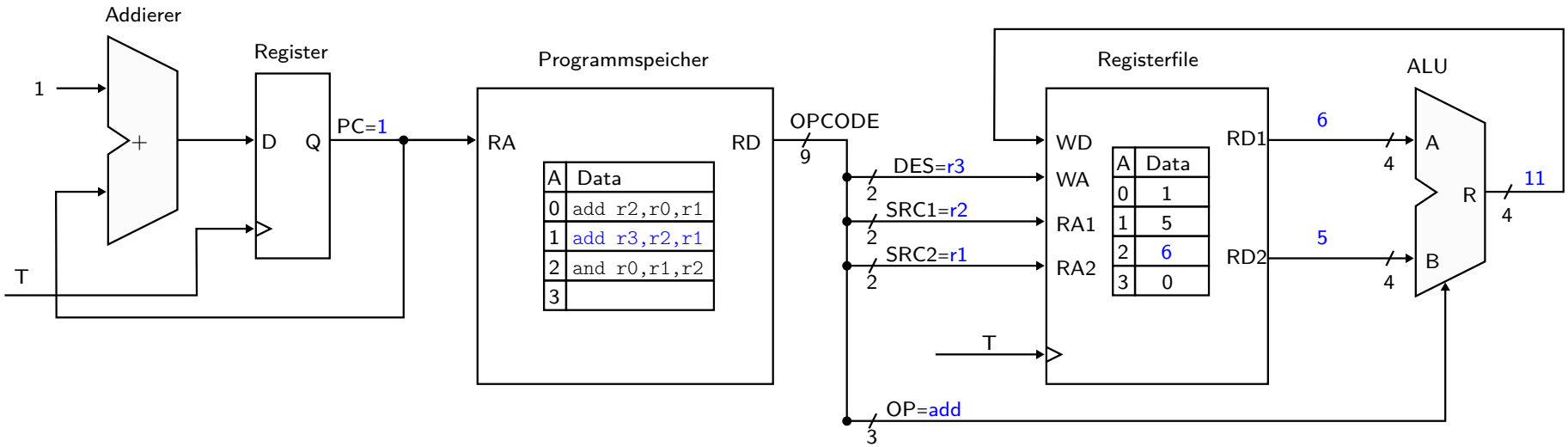
Simulation: Takt 0 (vor der 1. Taktflanke)



Minimal-Prozessor

Angewandte Informatik

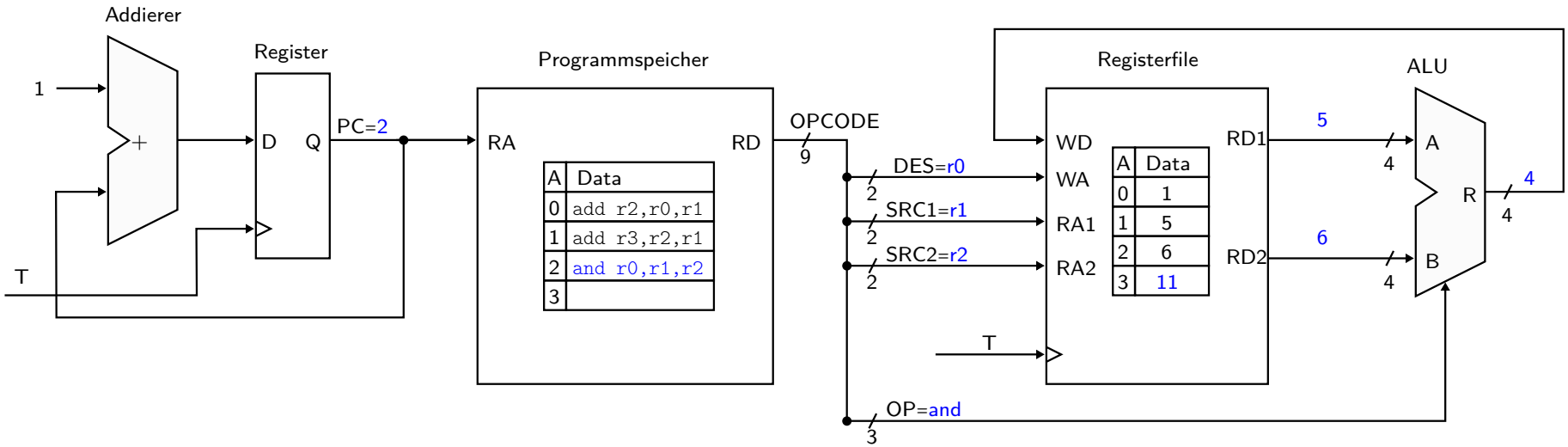
Simulation: Takt 1



Minimal-Prozessor

Angewandte Informatik

Simulation: Takt 2



Minimal-Prozessor

Was unser Minimal-Prozessor bisher kann:

- Ein lineares Programm »abspulen«
- Register können arithmetisch und logisch kombiniert werden
- Speicher limitiert auf Größe des Registerfiles

⇒ Wir müssen unseren Rechner um neue Befehle erweitern!

Inhalte

AI Angewandte Informatik

Registerinitialisierung

Angewandte Informatik

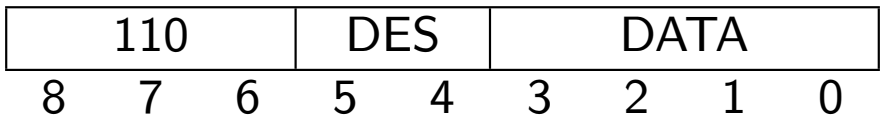
Der Befehl hierzu lautet *load immediate* (ldi): »lade umgehend«

Dieser erlaubt das Setzen von Registern aus dem Programmcode

Der OP Code 110 wird momentan noch nicht verwendet:

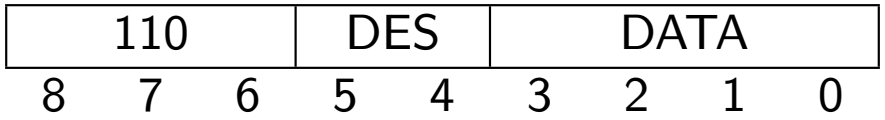
o_2	o_1	o_0	Befehl
0	0	0	add
0	0	1	sub
0	1	0	mul
0	1	1	and
1	0	0	or
1	0	1	xor
1	1	0	ldi
1	1	1	

Die Befehlscodierung könnte wie folgt aussehen:

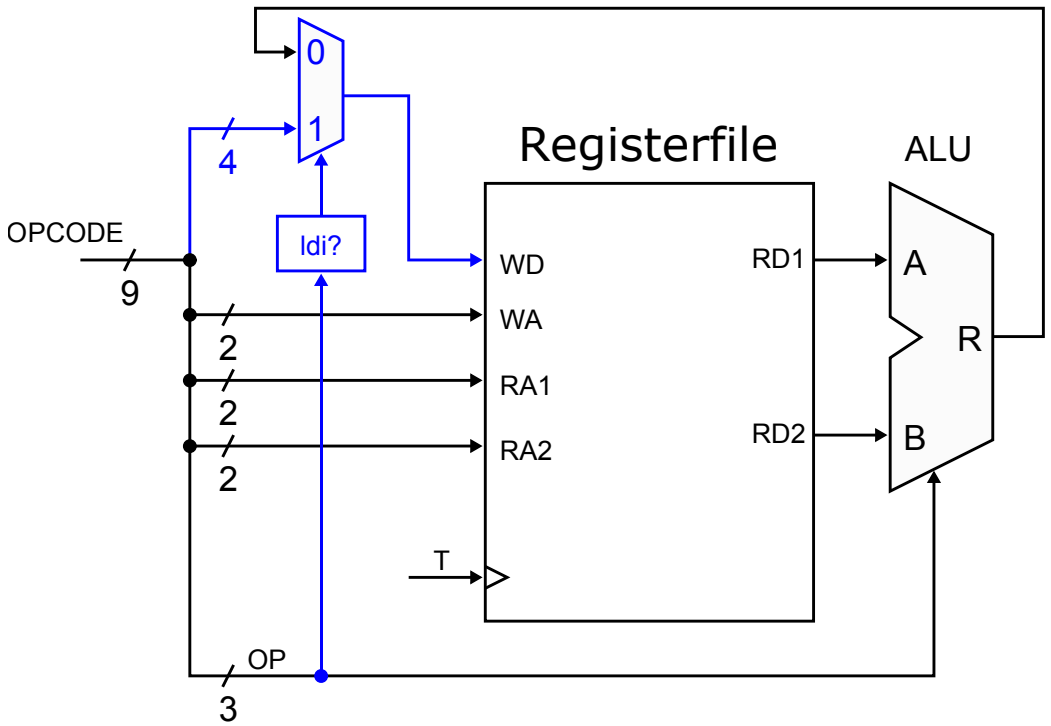


Beispiel: `ldi r1,12` ⇒ Lade Register `r1` mit Datenwort `12`.

Load Immediate Realisierung



Realisiert werden könnte das wie folgt:



D.h., wenn `ldi` erkannt wird (OP= 110_2) werden die untersten 4 Bits (DATA) über den Multiplexer in das Zielregister geschrieben.

Sprünge

Angewandte Informatik

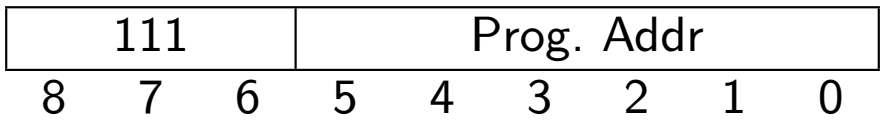
Der Befehl hierzu lautet *jump* (jmp): Springe

Dieser erlaubt Sprünge im Programmcode und dadurch Schleifen

Der Code 111 wird momentan noch nicht verwendet:

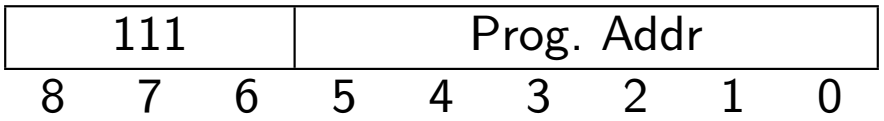
o_2	o_1	o_0	Befehl
0	0	0	add
0	0	1	sub
0	1	0	mul
0	1	1	and
1	0	0	or
1	0	1	xor
1	1	0	ldi
1	1	1	jmp

Die Befehlscodierung könnte wie folgt aussehen:

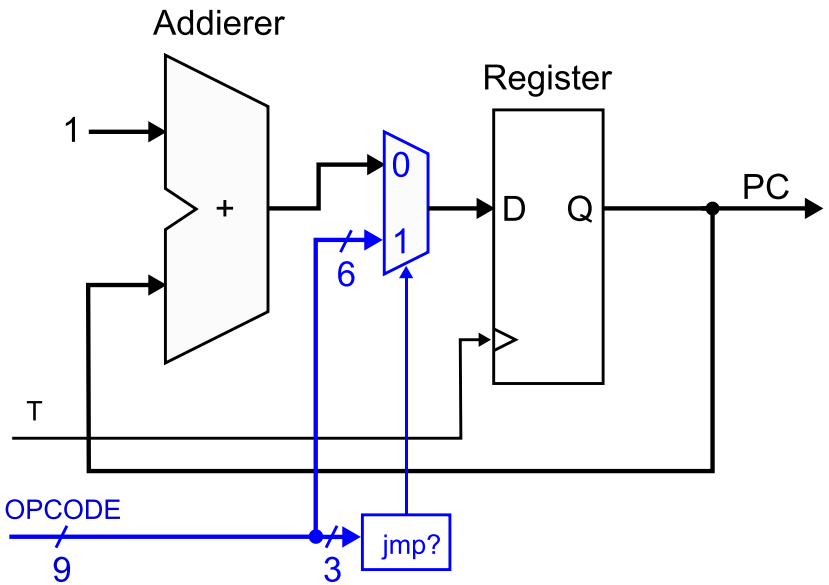


⇒ Springe zu Adresse »Prog. Addr«

Jump Realisierung



Realisiert werden könnte das wie folgt:



D.h., wenn `jmp` erkannt wird (OP= 111_2) wird der PC durch die neue Adresse ersetzt.

Bedingte Sprünge

Sprünge wie der zuvor eingeführte Befehl *jump* (`jmp`) werden immer ausgeführt

Für Schleifen im Programmablauf müssen wir den Sprung an eine Bedingung knüpfen

Beispielbefehl: `bne SRC1,SRC2,ADDR` (*branch on not equal*)

Sprung zu Programmadresse `ADDR` wird nur bei Ungleichheit der Register `SRC1` und `SRC2` durchgeführt

Beispiel: `bne r1,r2,5` ⇒ Wenn `r1`≠`r2`, dann Springe zu Programmadresse 5

Hinweis: Im Folgenden wird auf eine detaillierte Befehlscodierung und -erweiterung verzichtet

Beispiel: for-Schleife

C/Java:

```
for(int i=0; i != 10; i++)
{
  //loop body
  //...
}
```

Für Variable `i` wird Register `r0` verwendet.

Assembler:

```
00: ldi r0,0       # i=0
01: ldi r1,1       # r1=1 (Konstante 1)
02: ldi r2,10      # r2=10 (Konstante 10)
#loop body
#...
10: add r0,r0,r1 # i++
11: bne r0,r2,03 # springe zu loop body wenn i != 10
```

Vorlesungsaufgabe: if-Bedingung

Wie lautet der Assembler-Code folgenden C/Java-Programms?

```
if(x != y) {
  z=42;
} else {
  z=0;
}
```

Für Variablen `x`, `y` und `z` sollen die Register `r0`, `r1` und `r2` verwendet werden.

Realisierung bedingter Sprünge (Wdh.)

Realisierung eines *branch on not equal* (`bne`) Befehls:

- Erweiterung des `jmp` Programmwortes um die beiden Quellregister `SRC1` und `SRC2`
- Beide müssen auf Gleichheit geprüft werden (kombinatorischer Vergleich, oft in ALU realisiert)
- Sprungbedingung (MUX-Ansteuerung) wird nun mit Gleichheit verknüpft

Weitere mögliche bedingte Sprungbefehle:

- *branch on equal* (`beq`): Sprung bei Gleichheit
- *branch on zero* (`bz`): Sprung bei Null
- *branch on greater zero* (`bgz`): Sprung bei > 0
- *branch on less than zero* (`bltz`): Sprung bei < 0
- ...

Erweiterter Minimal-Prozessor

Was unser erweiterter Minimal-Prozessor bisher kann:

- Ein lineares Programm »abspulen«
- Register können arithmetisch und logisch kombiniert werden
- Register können initialisiert werden
- Sprünge im Programmablauf

Was unser erweiterter Minimal-Prozessor bisher nicht kann:

- Speicher limitiert auf Größe des Registerfiles

⇒ Wir müssen unseren Rechner um weitere Befehle erweitern!