ERA-Übungsblatt 07

1. • Übersetzung Assembly → Maschinensprache in RISC-V-32 sehr einfach (weil fixe 32-Bit-Größe)
→ einfach in Tabellen nachschauen und daraus Binärzahl bauen

add s7, s8, s9

0000000	11001	11000	000	10111	0110011
funct 7	Quellregister rs2 s9 = 25	Quellregister rs1 s8 = 24	funct3	Zielregister rd s7 = 23	Opcode op

Das Layout der einzelnen Komponenten kann in der Tabelle nachgeschlagen werden, da wir wissen, dass "add" ein R-Typ-Befehl ist

Ins Hexadezimalsystem umgewandelt: 0b00...011 = 0x019C0BB3
weiteres siehe ML

• Jetzt umgekehrt! Da unabhängig vom Instruktionstypen die hintersten 7 Bit immer den Opcode enkodieren, können wir daraus den Befehlstyp ablesen und anschließend alle anderen Parameter auslesen.

0x 01200513 in binär:

000000010010	00000	000	01010	0010011
Immediate: 0x12 = 18	Quellregister rs1 0 → zero	funct3 0 → addi	Zielregister rd 10 → a0	Opcode 19 → I-Typ

die dekodierte Instruktion lautet also: addi a0, zero, 18

ein weiteres (interessantes) Beispiel:

0x ff5ff06f in Binär:

1	1111111010	1	11111111	00000	1101111
Immediate Bit 20	Immediate Bits 10 bis 1	Immediate Bit 11	Immediate Bits 19 bis 12	Zielregister rd 0 → zero	Opcode 111 → JAL

Wir können also den Immediate zusammen basteln:

implizit durch immediate-extension, siehe WOG

imm = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 = -12 (Zweierkomplement!)

```
Loop:
  ins1
  ins2
  ins3
  jal zero, loop
```

die dekodierte Instruktion lautet also jal, zero, -12

Hinweis: Der Immediate "-12" wurde wahrscheinlich vom Assembler aus einem Label generiert: -12 ≙ "Springe 3 Instruktionen zurück"

2. Siehe ML

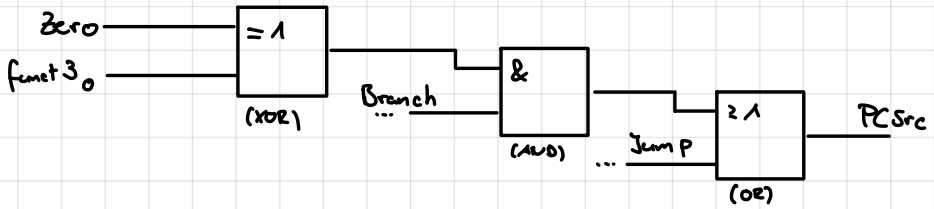
3. Unser Single-Cycle-Prozessor kann bereits einen beq. Wir wollen diesen nun so erweitern, dass bne auch funktioniert.
PCSrc = 1 bedeutet, dass wir einen Offset zum PC dazuaddieren (d.h. wir springen). Unser main decoder setzt dieses Signal also auf '1', falls ein jump (unbedingt) oder ein branch bei erfüllter Bedingung durchgeführt wird.
Schauen wir in unsere Tabelle, so sehen wir, dass bne und beq sich nur durch das 0te Bit in funct3 unterscheiden. Wir stellen also eine Wahrheitstabelle auf, wann das Signal, das mit branch verundet wird, '1' sein soll (≙ „Bedingung erfüllt").
Das Zero-Signal der ALU gibt uns dabei an, ob rs1 == rs2. Es wird eine Subtraktion durchgeführt, d.h. Zero = 1 ⇒ ALU-Ergebnis war 0 ⇒ rs1 − rs2 = 0 ⇒ rs1 = rs2

funct3₀	Zero	Bedingung erfüllt?
0	0	0
0	1	1
1	0	1
1	1	0

beq, aber Register sind nicht gleich → kein Sprung
bne und Register ungleich → Sprung!
beq und Register gleich → Sprung!
bne, aber Register sind gleich → kein Sprung

Aus der Wahrheitstabelle ist ersichtlich, dass unsere Funktion einem xor entspricht.
Die Schaltung wird also folgendermaßen erweitert:

möglicher Klausuraufgabe!



4. Klausuraufgabe!

Befehl	Befehlstype	ist der Befehl ein branch? ↑ Branch	ALU-Ergebnis, aus Speicher oder PC+4? ↑ ResultSrc	etwas in den Speicher schreiben? ↑ MemWrite	was soll die ALU machen? ↑ ALUControl	Register oder Immediate? ↑ ALUSrc	aus Tabelle ablesen ↑ ImmSrc	etwas in ein Register schreiben? ↑ RegWrite
OR	R	0	00	0	OR	0	–	1
BNE	B	1	–	0	SUB	0	10	0
XORI	I	0	00	0	XOR	1	00	1
SW	S	0	–	1	ADD	1	01	0

Zur Hausaufgabe: Der Großteil ist schon implementiert, die Lösung bedarf weniger Gatter und einiger neuer Verbindungen.

- bge: beq ist schon implementiert, die Erweiterung ist analog zu Aufgabe 3. Angenommen, wir können zur ALU ein weiteres Ausgangssignal (neben zero) hinzufügen: Wie kann überprüft werden, ob eine Zahl ≥ einer anderen ist? (Tipp: größer gleich ≙ nicht kleiner, Zweierkomplement)
  Bessere aber komplexere Alternative: SLT ("set on less than") ist bereits implementiert
- slli: sl ist eine neue Operation d.h. ein neues ALUControl-Signal und die dementsprechende Left-Shift-Implementierung in der ALU muss hinzugefügt werden. (Tipp: Schau dir die bereits implementierten arithmetischen/logischen Befehle an). Logisim verfügt über eine Shifting-Komponente: Arithmetic → Shifter