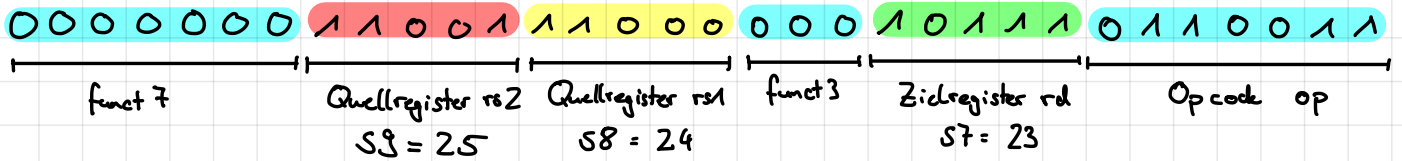


ERA-Übungsblatt 07

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

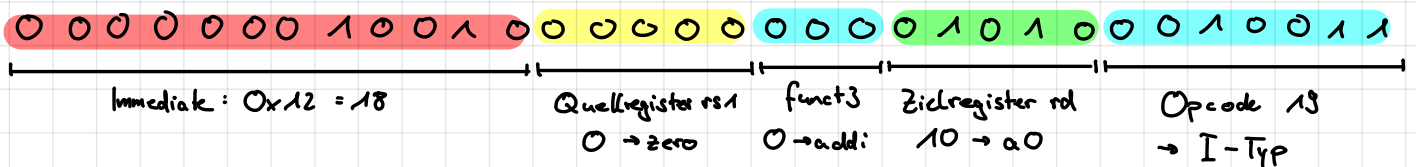


Das Layout der einzelnen Komponenten kann in der Tabelle nachgeschlagen werden, da wir wissen, dass "add" ein R-Typ-Befehl ist

Ins Hexadezimalsystem angewandelt: $0b00...011 = \underline{\underline{0x013C0BB3}}$
weiter 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.

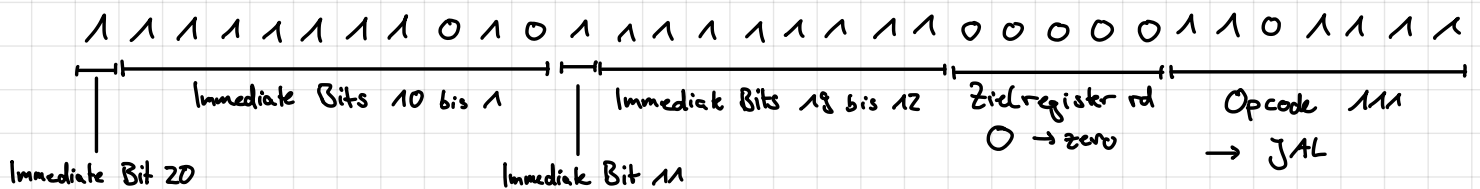
$0x01200513$ in Binär:



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

ein weiteres (interessantes) Beispiel:

$0xff5ff06f$ in Binär:



Wir können also den Immediate zusammen basteln:

$\text{imm} = 11111111111111010 = -12$ (Zweierkomplement!)

Loop:
ins1
ins2
ins3
jal zero, loop

die dekodierte Instruktion lautet also jal, zero, -12

Hinweis: Der Immediate "-12" wurde wahrscheinlich vom Assembler auf einem Label generiert: $-12 \hat{=}$ "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 ein Offset zum PC dazunaddieren (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 verwendet wird, '1' sein soll (="Bedingung erfüllt").

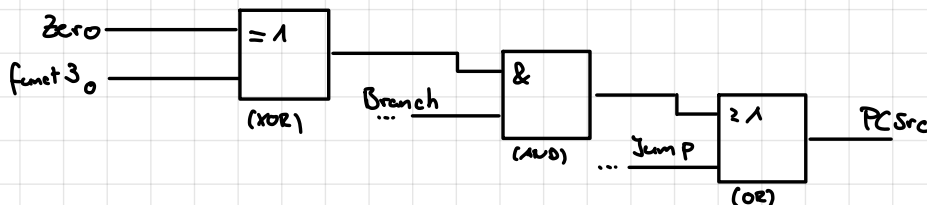
Das Zero-Signal der ALU gibt uns dabei an, ob $rs1 == rs2$. Es wird eine Subtraktion durchgeführt, d.h. $Zero = 1 \Rightarrow ALU\text{-Ergebnis war } 0 \Rightarrow rs1 - rs2 = 0 \Rightarrow rs1 = rs2$

funct3 ₀	Zero	Bedingung erfüllt?
0	0	0
0	1	1
1	0	1
1	1	0

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

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

mögliche Klausuraufgabe!



4. Klausuraufgabe!

Befehl	Befehlstype	ist der Befehl ein branch? ↑ Branch	ALU-Ergebnis, aus Speicher oder PC+4? ↑ ResultSrc	etwas in den Speicher schreiben? ↑ MemWrtik	was soll die ALU machen? ↑ ALUControl	Register oder Immédiat? ↑ ALUSrc	aus Tabelle ablesen ↑ Imm Src	etwas in ein Register schreiben? ↑ RegWrtik
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 \geq einer anderen ist? (Tipp: größer gleich $\hat{=}$ nicht kleiner, Zweierkomplement)
 Bessere aber komplexere Alternative: SLT ("set on less than") ist bereits implementiert
- sll: sll ist eine neue Operation d.h. ein neues ALUControl-Signal und die dazugehörige Left-Shift-Implementierung in der ALU muss hinzugefügt werden. (Tipp: Schenke die bereits implementierten arithmetischen/logischen Befehle an). Logisim verfügt über eine Shifting-Komponente: Arithmetic \rightarrow Shifter