1. Performance-Berechnungen

Op	**Freq**	CPI_i	**Freq x** CPI_i
`ALU`	25%	5	1.25
`LOAD`	25%	10	2.5
`STORE`	25%	7.5	1.875
`Branch`	25%	7.5	1.875
			$\Sigma = 7.5$

a) Die Tabelle sieht dann wie folgt aus:

Op	**Freq**	CPI_i	**Freq x** CPI_i
`ALU`	25%	5	1.25
`LOAD`	25%	6	1.5
`STORE`	25%	7.5	1.875
`Branch`	25%	7.5	1.875
			$\Sigma = 6.5$

Die CPU ist 13.3% schneller.

b)

Op	**Freq**	CPI_i	**Freq x** CPI_i
`ALU`	25%	2.5	0.625
`LOAD`	25%	6	2.5
`STORE`	25%	7.5	1.875
`Branch`	25%	7.5	1.875
			$\Sigma = 6.875$

Die CPU ist 8.3% schneller.

2. Stackverwendung bei Subroutinen

- Beim Aufruf von Subroutinen wird Speicherplatz für die lokalen Variablen der Funktion reserviert ("stack frame", damit verbunden der "Frame Pointer").

- Ein weiterer Verwendungszweck besteht darin, dass die Parameter auf dem Stack abgelegt und zwischengespeichert werden, damit sie von der Subroutine weiterverwendet werden können.

3. ALU & Most Significant Bit

Die ALU muss für das *most significant bit* deshalb anders aufgebaut sein, damit `slt` unterstützt werden kann. Das *msb* gibt als einziges Bit das `Less` weiter (an das *lsb*). Zudem muss der Overflow abgefangen werden, weshalb diese Leitung nicht zum nächst höheren Bit führt (es gibt ja kein höherwertiges Bit).

4. ALU & `SLT`

$$A\ slt\ B = \begin{cases} 0...01 \text{ if } A < B & \text{i.e. if } A - B < 0 \\ 0...00 \text{ if } A \geq B & \text{i.e. if } A - B \geq 0 \end{cases} \tag{1}$$

Beim `slt`-Befehl (*set on less than*) wird beim "Operation"-control der Schalter auf 3 gesetzt. Dies bewirkt, dass beim höchstwertigen Bit das set übertragen wird auf dass less beim tiefstwertigen Bit. Alle Bits ausser dem niederwertigsten haben bei `less` 0 als Input.
Der Trick ist nun dass die Bedingung $A < B$ umformuliert werden kann zu $A - B < 0$. Bei dieser Operation kann einfach das höchstwertige Bit betrachtet werden, welches angibt, op $A - B$ negativ ist. Ist dies der Fall, muss $A < B$ gelten und das Bit wird übertragen.

5. Pop und push

- `pop`: Der Wert wird geladen und z.B. in Register `$r3` gespeichert. Danach wird zum (Stack-)Pointer 4 addiert, um ihn auf das nächste Element zeigen zu lassen.

```
pop:    lw      $r3, 0($sp)
        addi    $sp, $sp, 4
```

- `push`: Hier ist das Gegenteil der Fall. Der Stackpointer wird um -4 verschoben, sodass dort das neue Element eingefügt werden kann.

```
push:   addi    $sp, $sp, -4
        sw      $r3, 0($sp)
```

6. `loadi`

Da wir eine 32 Bit Konstante speichern wollen, aber der Befehl insgesamt nur 32 Bit sein kann, muss die Instruktion aufgeteilt werden in zwei separate Befehle. Mit `lui` können die 16 oberen Bits gesetzt werden, und mit `ori` die Unteren.
imm_upper sind die höherwertigen 16 Bits, *imm_lower* analog dazu die Niederwertigen.

```
lui    $r3, imm_upper
ori    $r3, $r3, imm_lower
```

7. ALU: OPCodes

operation	***opcode***	***funct***	**Erklärung**
`and`	000 000	100 100	Beim bitweisen and werden **Ainvert** und **Binvert** auf 0 gesetzt, und bei der Operation, also beim Multiplexer, wird das erste Resultat weiter verwertet. Vor dem Multiplexer führen die Daten durch ein and-Modul.
`or`	000 000	100 101	Die Bits hier sind gleich gesetzt wie beim and, ausser dass beim Multiplexer das zweite Resultat ausgewählt wird (das vorher durch ein OR-Modul geführt wurde).
`add`	000 000	100 000	Hier sind **Ainvert** und **Binvert** auf 0, und beim Multiplexer wird das dritte Resultat verwendet. Dieses Resultat stammt aus einem Halbaddierer.
`sub`	000 000	100 010	Gleich wie add, nur dass **Binvert** auf 1 gesetzt wurde.
`slt`	000 000	101 010	Gleich wie sub, nur dass bei Operation der Wert auf 3 gesetzt wird, so dass das Ergebnis von less als Resultat der ALU weiterverwertet wird.
`nor`	000 000	100 111	Gleich wie and, nur dass die beiden Schaltungen bei **Ainvert** und **Binvert** auf 1 gesetzt wurden, d.h. es wird mit dem Komplement der beiden weitergerechnet.