

**Allgemeiner Hinweis:** Für viele der nachfolgenden Aufgaben ist es von Vorteil, den in der Vorlesung gezeigten Simulator zu verwenden. Relevante Links finden Sie im TUWEL. Sämtliche Aufgaben können aber auch ohne Verwendung des Simulators gelöst werden.

Weiters finden Sie im TUWEL zu der Programmieraufgabe Assemblycode. Dieser Code beinhaltet einige Testfälle. Kopieren Sie dazu die bereitgestellten Assembly Files einfach in den Editor vom Simulator.

**Wichtig:** Die Testfälle sollen Ihnen bei der Lösung der Aufgaben helfen. Sie decken womöglich nicht alle Fälle ab und positive Testergebnisse bedeuten nicht, dass Sie automatisch 100% auf Ihre Tafelleistung bekommen.

### Aufgabe 1: RISC-V – Rekursion

Implementieren Sie eine Funktion, welche die  $n$ -te Fibonacci-Zahl<sup>1</sup> *rekursiv* berechnet. Die Gleichungen in 1 definieren die Fibonacci-Folge. Sie dürfen annehmen, dass  $n \geq 0$  hält.  $n$  wird Ihrer Funktion via Register `a0` übergeben. Weiters, soll Ihre Funktion das Label `fibonacci` tragen.

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1}, \text{ für } n > 1 \end{aligned} \tag{1}$$

Halten Sie sich beim Aufruf von Subroutinen an die in der Vorlesung besprochenen Aufrufkonventionen und verwenden Sie den Stack, wie in der Vorlesung besprochen. Dieser sollte von den hohen zu niedrigeren Adressen wachsen. Der von uns empfohlene Simulator wird den Stackpointer bereits auf `0xbffffff00` initialisieren. Falls Sie andere Systeme für das Testen Ihrer Lösung verwenden, stellen Sie sicher, dass der Stackpointer vor dem ersten Aufruf Ihrer Funktion initialisiert wird.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Fibonacci\\_sequence](https://en.wikipedia.org/wiki/Fibonacci_sequence)

## Aufgabe 2: Konstanten in RISC-V Instruktionen

RISC-V definiert verschiedene Instruktionstypen. Diese kodieren Konstanten (Immediates) unterschiedlich. In dieser Aufgabe werden Sie einen Baustein entwerfen, welcher diese Konstanten dekodiert und als 32-Bit-Zahl im 2er-Komplement aufbereitet. Ihr Baustein hat zwei Eingangswerte. Der erste ist der gesamte Bitvektor der Instruktion. Der zweite Eingangswert gibt den Instruktionstypen als 3-Bit (unsigned) Zahl an. Letzteres könnte auch in Ihrem Baustein berechnet werden. Da diese Information aber auch in anderen Teilen unseres fiktiven Prozessors benötigt wird, wird diese Information zentral berechnet und Ihrem Baustein zur Verfügung gestellt. Tabelle 1 gibt die Kodierung des Instruktionstypen an. Ihr Baustein soll den Wert 0 zurückliefern für Instruktionstypen, die keinen Immediate definieren (z. B. R-Type). Nutzen Sie die „RISC-V Instruction Set Summary“<sup>2</sup>, die von dem Lehrbuch zur Verfügung gestellt wird, um die genaue Kodierung der Konstanten zu erfahren.

*Hinweis:* Die von uns hier verlangte Notation kann am Anfang etwas unintuitiv für Sie sein. Jedoch erlaubt sie Ihnen in der letzten Teilaufgabe (diese ist optional) einen echten Hardwareentwurf mithilfe der Hardwarekonstruktionsprache Chisel<sup>3</sup> zu erstellen und ihren Entwurf zu testen. Ihr Beispiel wird auch als richtig gewertet, falls Sie keinen korrekten Chisel Code produzieren, solange die Intention Ihrer Ausdrücke klar erkennbar ist.

Kodierter Wert	Instruktionstyp
0	R-Typ
1	I-Typ
2	S-Typ
3	B-Typ
4	U-Typ
5	J-Typ
6	R4-Typ

Tabelle 1: Kodierung des Instruktionstyps

- a) Im ersten Schritt, überlegen Sie sich welche Bitvektoren Sie aus der kodierten Instruktion extrahieren müssen, um die Konstanten zusammenbauen zu können. Beachten Sie hier, dass Sie gegebenenfalls Umordnungen durchführen müssen. Zum Beispiel, das Feld `imm4:1,11` im B-Typ beinhaltet zwei Bitvektoren die an *unterschiedlichen* Stellen in der finalen Konstante eingefügt werden müssen. Listen Sie alle notwendigen Bitvektoren als Text in der Form `ir(x, y)`, bzw. `ir(x)` für 1-Bit lange Teilbereiche. Zum Beispiel, das Feld `imm11:0` des I-Typs können Sie mit `ir(31, 20)` anführen. Die Variable `ir` steht hier für die Instruktion.
- b) Im nächsten Schritt, fügen Sie die in a) gelisteten Bitvektoren für jeden Typ zusammen. Nutzen Sie hier die Notation `Cat(a, b, ...)` für die Konkatenation von Bitvektoren. Um Konstanten in Binär anzugeben, nutzen Sie die Notation `"b0".U`. Wollen Sie der Konstante eine bestimmte Länge geben verwenden Sie die Notation `"b0".U(12.W)` für eine 12-Bit breite Konstante mit dem Wert 0. Hier ein Beispiel: `Cat("b0".U(2.W), "b1".U) = "b001".U(3.W)`. Weiters, um eine Konstante zu wiederholen (z. B. sign-extension), verwenden Sie die Notation `Fill(20, "b1".U)`. *Hinweis:* Alle Instruktionstypen (außer U-Type wo diese Überlegung entfällt) fordern eine sign-extension auf 32-Bit.

Hier ein Beispiel Berechnung für den I-Typ: `Cat(Fill(20, ir(31)), ir(31, 20))`. Der erste Term in der Konkatenation wiederholt das MSB der Konstante 20 Mal (sign-extension). Die unteren 12 Bits werden aus dem `imm11:0` Feld übernommen.

- I-Typ:
- S-Typ:
- B-Typ:
- U-Typ:
- J-Typ:

<sup>2</sup>[https://pages.hmc.edu/harris/ddca/ddcarv/DDCarv\\_AppB\\_Harris.pdf](https://pages.hmc.edu/harris/ddca/ddcarv/DDCarv_AppB_Harris.pdf)

<sup>3</sup><https://www.chisel-lang.org/>

- c) Im letzten Schritt müssen Sie lediglich, basierend auf dem Instruktionstyp `ir_type`, entscheiden welches dieser Ergebnisse Sie als Ausgangswert verwenden wollen. Füllen Sie die unten angegebenen Notation eines Multiplexers aus. Sie können in der PDF Abgabe Platzhalter für die Ausdrücke aus Aufgabe b) verwenden, falls Sie sich das Abschreiben ersparen möchten.

```
1 MuxLookup(ir_type, 0.U(32.W), Seq(  
2   0.U(3.W) -> ,  
3   1.U(3.W) -> ,  
4   2.U(3.W) -> ,  
5   3.U(3.W) -> ,  
6   4.U(3.W) -> ,  
7   5.U(3.W) -> ,  
8   6.U(3.W) -> ,  
9 ))
```

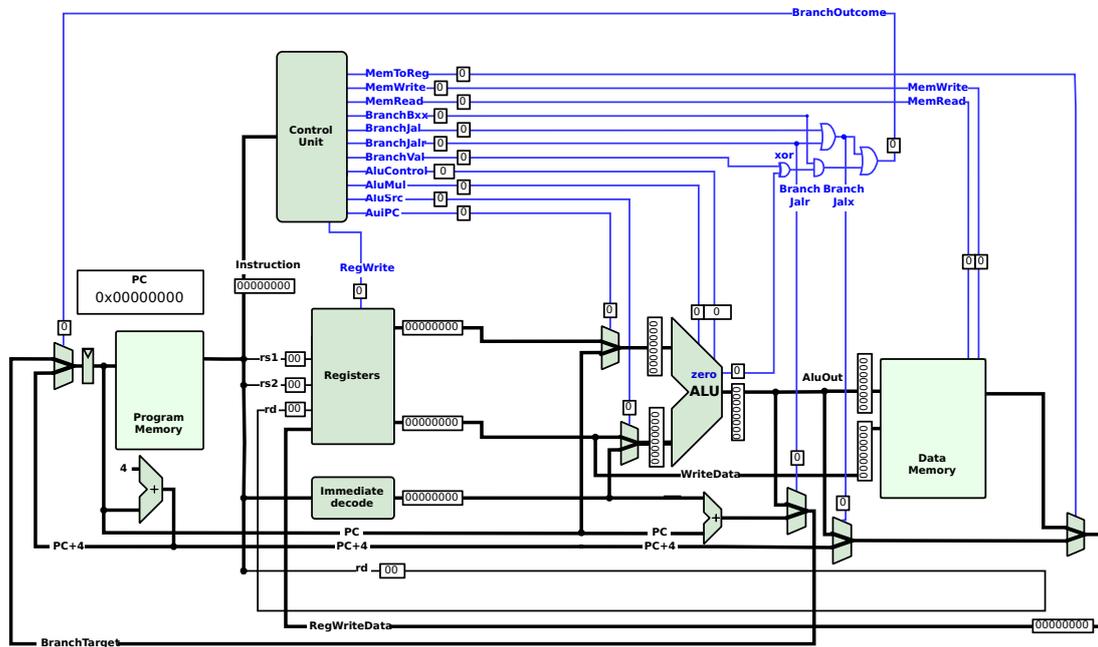
- d) **Optional:** Sie können nun Ihren Baustein mit Hilfe von Chisel definieren um einen echten Hardwareentwurf zu erstellen. Verwenden Sie dazu das Chisel online Bootcamp<sup>4</sup> um ein lokales Setup zu vermeiden und laden Sie das von uns bereit gestellte Notebook (siehe TUWEL) hoch. Nutzen Sie dazu die *Upload Files* Schaltfläche links oben im User Interface. Öffnen Sie das Notebook mit einem Doppelklick auf den entsprechenden Eintrag in der Dateiliste. Ersetzen Sie den Platzhalter in der Sektion “Hardware Definieren” mit dem Code aus Aufgabe c). Führen Sie alle Blöcke nach der Reihe aus, um die Tests auszuführen. Wir gratulieren Ihnen zu Ihrem (möglicherweise) ersten Hardwareentwurf. Sie können einen Screenshot der Ausgabe der letzten Zelle in das PDF kopieren. Wenn das Notebook **Success!** ausgibt sind alle Tests ohne Fehler durchgelaufen.

---

<sup>4</sup><https://mybinder.org/v2/gh/freechipsproject/chisel-bootcamp/master>

### Aufgabe 3: QtRVSim Architektur – Aufbau und Funktionsweise

Nachfolgende Abbildung zeigt Ihnen die simulierte single-stage Mikroarchitektur des QtRVSim<sup>5</sup> simulators. Diese unterscheidet sich in einigen Aspekten von der in den Foliensätzen gezeigten Mikroarchitektur (Mikroarchitektur, Folie 43). Die meisten Konzepte können aber leicht auf die unten gezeigte Mikroarchitektur übertragen werden.



a) Beschreiben Sie kurz die Funktionsweise der folgenden Komponenten in der Mikroarchitektur des Simulators.

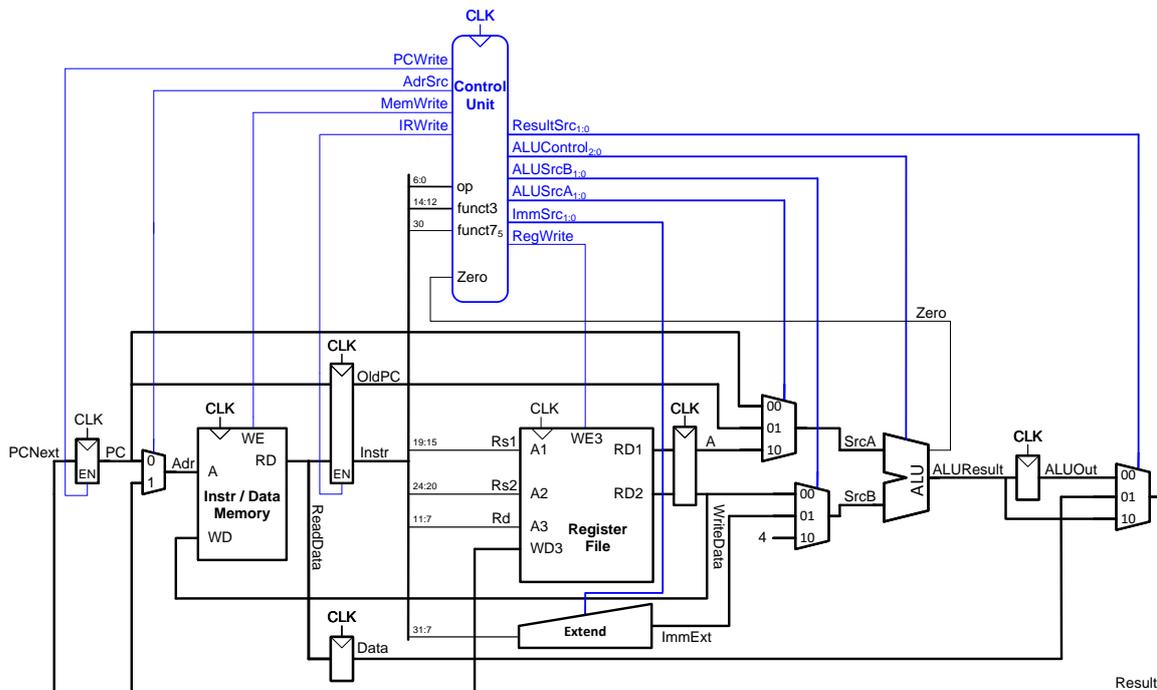
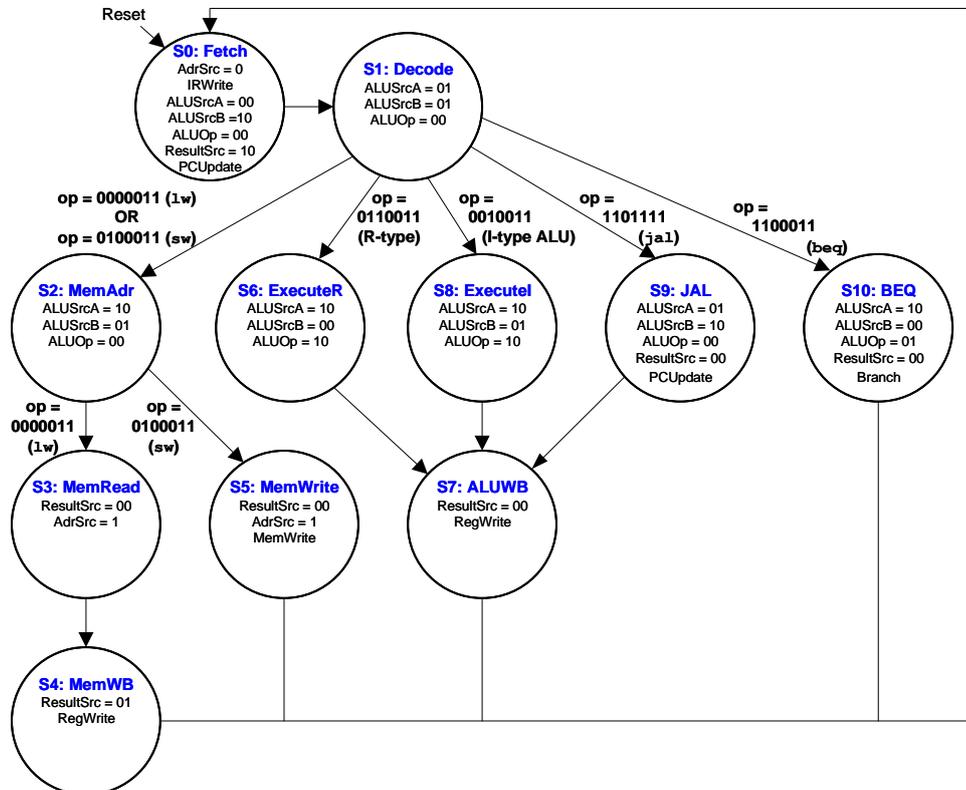
- Registers:
- Arithmetic Logic Unit (ALU):
- Program Memory:
- Datenspeicher:
- Program Counter Register (PC):

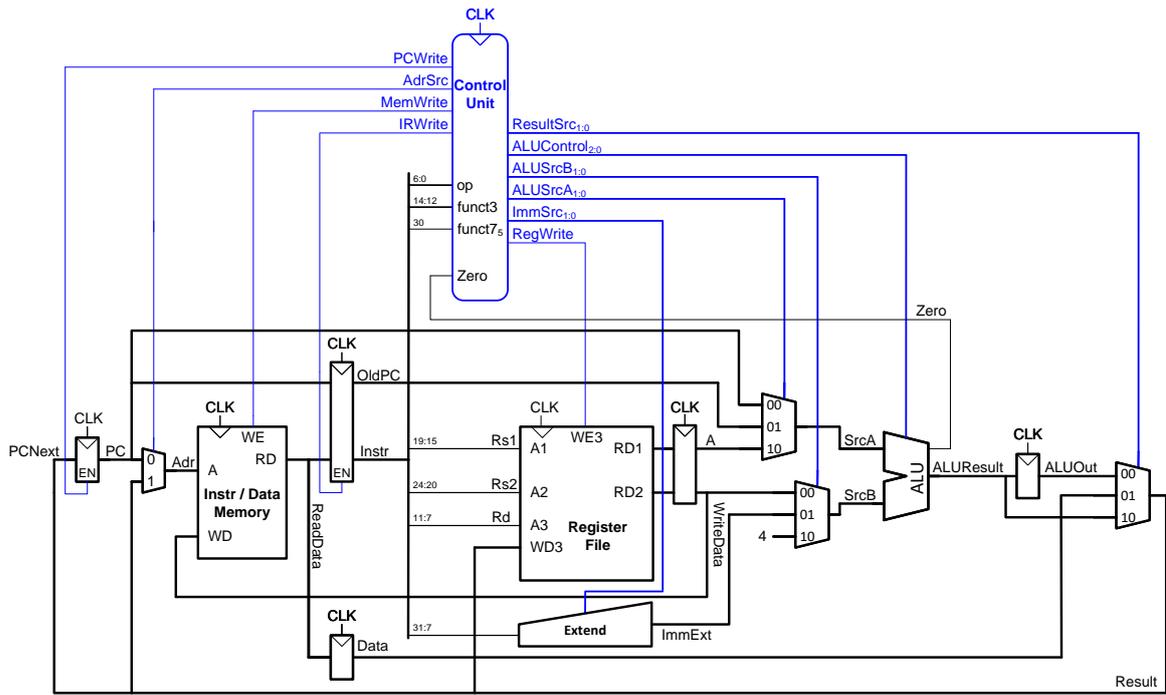
<sup>5</sup><https://github.com/cvut/qtrvsim>

- b) Finden Sie mindestens 3 Unterschiede zwischen der Mikroarchitektur aus der Vorlesung und der des Simulators. Erklären Sie wieso diese Abweichungen notwendig sind. Es ist nicht notwendig, dass Sie eine gründliche Literaturrecherche durchführen oder im Code des Simulators nachschlagen. Denken Sie über die Funktionsweise des Simulators nach und stellen Sie nachvollziehbare Vermutungen auf. Rein namentliche Unterschiede (z.B., Instruction Memory vs. Program Memory) dürfen nicht gezählt werden. *Hinweis:* Einige Abweichungen sind aufgrund der Menge an implementierten Instruktionen.

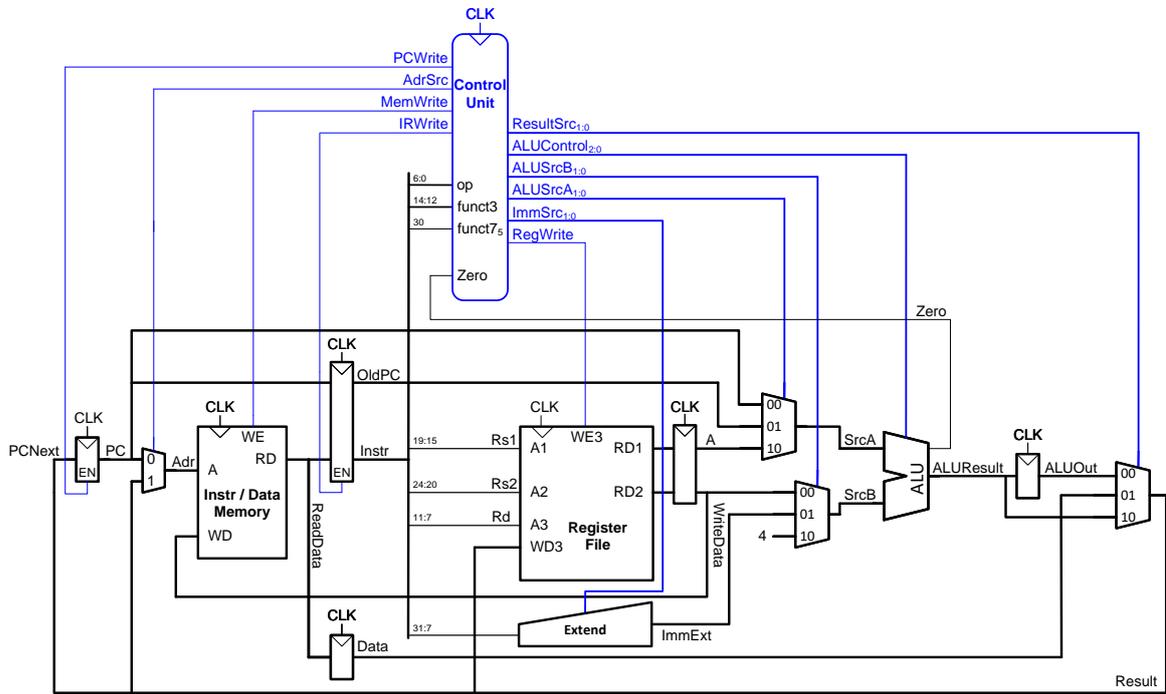
### Aufgabe 4: Multi-Cycle Processor

Die folgenden Grafiken zeigen einen Multi-Cycle RISC-V Prozessor. Ihre Aufgabe ist es in der Übung, die Kontrollsignale und Datenpfade über alle Zyklen der Befehle `and` und `jal` zu erklären. Erstellen Sie dazu ein konkretes Beispiel mit sinnvollen Registern und Speicherinhalten Ihrer Wahl. Zeigen Sie zudem auch, wie diese im Maschinenbefehl codiert sind, und beschriften Sie die relevanten Signale mit den Werten, die Sie während der Ausführung der Befehle annehmen.





Reserve:



### Aufgabe 5: Pipelining – Leiterplattenherstellung

Bei der Herstellung einer Leiterplatte (PCB ... *printed circuit board*) werden fünf Produktionsschritte wie folgt im Pipeline-Verfahren durchlaufen. Die Dauer eines Produktionsschrittes ist in generischen Zeiteinheiten (ZE) angegeben.

- 1.) Belichten und Entwickeln: einseitige Leiterplatte (standard): 1 ZE  
doppelseitige 3 ZE
- 2.) Ätzen: 1 ZE
- 3.) Bohren: SMD(*surface mounted devices*)-Montage (standard): 1 ZE  
Durchsteckmontage (THT ... *through hole technology*): 3 ZE
- 4.) Kupferabscheidung: 35 $\mu$ m (standard): 1 ZE  
70 $\mu$ m: 2 ZE
- 5.) Fräsen und Trennen: Standardformat: 2 ZE  
Sonderformat: 3 ZE

Jede Leiterplatte muss jeden Produktionsschritt durchlaufen und die betreffenden Maschinen können nur jeweils eine Leiterplatte aufnehmen. Im Unterschied zu Takt-gesteuerten Pipelines in Prozessoren rückt hier ein Bauteil nach Abschluss einer Verarbeitungsstufe in die nächste weiter, sobald diese frei ist. Es muss kein gemeinsamer Takt gefunden werden. Es existieren für alle Produktionsschritte, außer Ätzen, zwei Varianten: eine Standardvariante und eine etwas aufwändigere Spezialvariante.

Folgende Leiterplattentypen sollen im weiteren Verlauf hergestellt werden:

Leiterplattentyp	Belichten	Ätzen	Bohren	Kupferabscheidung	Fräsen
PCB1	doppelseitig	standard	SMD	35 $\mu$	Standardformat
PCB2	einseitig	standard	SMD	70 $\mu$	Standardformat
PCB3	einseitig	standard	SMD	35 $\mu$	Sonderformat
PCB4	einseitig	standard	THT	70 $\mu$	Standardformat
PCB5	doppelseitig	standard	SMD	70 $\mu$	Sonderformat

Gehen Sie bei den nachfolgenden Unteraufgaben immer davon aus, dass die Pipeline anfangs leer ist und somit die erste Leiterplatte ohne Verzögerung bearbeitet werden kann.

- a) Wie lange dauert die Herstellung einer einzelnen Leiterplatte vom Typ PCB1?
- b) Wie lange dauert es, zehn Leiterplatten vom Typ PCB1 in Folge herzustellen?
- c) Wie lange dauert die Herstellung der drei Leiterplatten PCB1, PCB3 und PCB4 in der Reihenfolge PCB1–PCB3–PCB4?
- d) Kann die Gesamtdauer aus Teilaufgabe c) durch Umordnen der Reihenfolge verkürzt werden?
- e) Welche der folgenden Verbesserungen des Prozesses bringt mehr Zeitgewinn für die Herstellung von PCB5–PCB2–PCB5–PCB2:  
A: Belichtung von doppelseitigen Platinen um zwei Zeiteinheiten beschleunigen.  
B: Fräsen/Trennen bei Sonderformat um eine Zeiteinheit beschleunigen.

## Aufgabe 6: Pipelining – Performanceverbesserung

Ein Prozessor besitzt eine fünfstufige Pipeline: *Fetch*, *Decode*, *Execute*, *Memory* und *Write Back*.

Der Instruktionssatz des Prozessors umfasst die Instruktionstypen  $i_1$ ,  $i_2$ ,  $i_3$  und  $i_4$ . Die Dauer der Ausführung einer Verarbeitungsstufe, abhängig vom Typ der Instruktion, ist in folgender Tabelle angegeben:

Instruktionstyp	Fetch	Decode	Execute	Memory	Write Back	Summe
$i_1$	55ns	50ns	150ns	125ns	0ns	380ns
$i_2$	75ns	50ns	250ns	100ns	55ns	530ns
$i_3$	95ns	75ns	150ns	175ns	120ns	615ns
$i_4$	50ns	125ns	150ns	200ns	0ns	525 ns

- a) Geben Sie die kleinstmögliche Taktzykluszeit für diesen Prozessor an, wenn die Instruktionen ohne Pipelining ausgeführt werden. Pro Taktzyklus soll genau eine Instruktion ausgeführt werden.
- b) Der in Unteraufgabe a) verwendete Prozessor soll auf Pipelineverarbeitung umgestellt werden. Aus Kostengründen sollen die Verarbeitungsstufen unverändert bleiben. Wie groß wählen Sie unter dieser Voraussetzung die Taktzykluszeit der Pipeline?
- c) Berechnen Sie den theoretischen Durchsatz in MIPS für die Prozessoren aus Unteraufgabe a) und b).
- d) Angenommen, bei Pipelining (vgl. Aufgabe b) liegt der reale Durchsatz des Prozessors 25% unter dem theoretischen Durchsatz. Wie viele Instruktionen verlassen in 500ms durchschnittlich die Pipeline?
- e) Welche der folgenden Änderungen der Pipelinestruktur bringt eine Verbesserung hinsichtlich des theoretischen Durchsatzes?
- Zusammenfassen und Optimieren von *Decode* und *Execute*, sodass alle Instruktionen in der neuen Stufe *Decode & Execute* 150ns benötigen.
  - Auftrennen der Memory-Stufe in zwei Stufen *Memory1* und *Memory2*, wobei  $i_1$  und  $i_2$  jeweils 90ns in den beiden neuen Stufen brauchen,  $i_3$  100ns und  $i_4$  125ns.
  - Eine allgemeine Optimierung, die jede Stufe, die mehr als 100ns um 25ns verkürzt und Stufen mit mehr als 150ns sogar um 55ns.

## Aufgabe 7: Pipelining – RAW-Hazard

Sie arbeiten mit einem Prozessor, der eine vierstufige Pipeline besitzt: Fetch (F), Decode (D), Execute (E) und Store (S).

Bedingt durch die Pipelinestruktur kann es zu *RAW Data Hazards* kommen, welche durch verzögerte Ausführung (*stall*) der lesenden Instruktion vermieden werden. Dabei wird die lesende Instruktion erst dann in Stufe D verarbeitet, wenn die schreibende Instruktion Stufe S abgeschlossen hat. Nehmen Sie zwecks Vereinfachung an, dass Lesezugriffe auf den Speicher ebenfalls in Stufe D und Schreibzugriffe in Stufe S ausgeführt werden. Auf dem Prozessor wird folgendes Programm ausgeführt:

```

1 sw    a2, 0(sp)
2 add   a1, a2, a1
3 div   a5, a3, a4
4 addi  a1, a1, 1
5 sub   a6, a5, a1
6 mul   a1, a5, a6
7 lw    a3, 0(sp)

```

- a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Voraussetzung, dass die Pipeline am Beginn und am Ende leer ist.

Zeit ↓	F	D	E	S
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

- b) Kreuzen Sie nachfolgend an, ob es sich um korrekte Umordnungen der Instruktionsfolge handelt oder nicht. Eine Umordnung ist korrekt, wenn die Funktionalität erhalten bleibt. Begründen Sie Ihre Antwort und geben Sie bei korrekten Umordnungen an, *wie viele Takte* die Ausführung benötigt.

add	add	add	sw
sw	sw	addi	lw
addi	addi	lw	add
lw	div	div	addi
mul	sub	sub	div
sub	mul	sw	sub
div	lw	mul	mul

- |                                     |                                     |                                     |                                     |
|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| <input type="radio"/> korrekt       | <input type="radio"/> korrekt       | <input type="radio"/> korrekt       | <input type="radio"/> korrekt       |
| <input type="radio"/> nicht korrekt |