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[1] *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 `0xbfffff00` 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.

```
fibonacci:
addi t1, zero, 1
beq a0, zero, end  # if a0 = 0 return 0
beq a0, t1, end    # if a0 = 1 return 1
                   # else
   addi sp, sp, -16      # reserve space on the stack
   sw a0, 0(sp)          # save a0 to stack
   sw ra, 4(sp)          # save return address to stack
   addi a0, a0, -1       # a0 = a0-1
   jal fibonacci         # fibonacci(n-1)
   mv t0, a0             # t0 = a0
   lw a0, 0(sp)          # get a0 from stack
   sw t0, 0(sp)          # save t0 to stack
   addi a0, a0, -2       # a0 = a0-1
   jal fibonacci         # fibonacci(n-2)
   mv t1, a0             # t1 = a0
   lw t0, 0(sp)          # get t0 from stack
   lw ra, 4(sp)          # get the return address from stack
   addi sp, sp, 16       # free space on the stack
   add a0, t0, t1        # return t0 + t1
end:
ret
```

[1] `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 Eingangwerte. 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 Intruktionstypen 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“[2], 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 Hardwarekonstruktionssprache Chisel[3] 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 gegebenfalls Umordungen durchführen müssen. Zum Beispiel, das Feld $\texttt{imm}_{4:1,11}$ 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 $\texttt{ir}(x, y)$, bzw. $\texttt{ir}(x)$ für 1-Bit lange Teilbereiche. Zum Beispiel, das Feld $\texttt{imm}_{11:0}$ des `I-Typs` können Sie mit $\texttt{ir}(31, 20)$ anführen. Die Variable `ir` steht hier für die Instruktion.

```
R = 0
I = ir(31, 20)
S = ir(31, 25) + ir(11, 7)
B = ir(31) + ir(7) + ir(30, 25) + ir(11, 8)
U = ir(31, 12)
J = ir(31) + ir(19, 12) + ir(20) + ir(30, 21)
R4 = 0
```

b) Im nächsten Schritt, fügen Sie die in a) gelisteten Bitvektoren für jeden Typ zusammen. Nutzen Sie hier die Notation $\texttt{Cat}(a, b, \dots)$ für die Konkatenation von Bitvektoren. Um Konstanten in Binär anzugeben, nutzen Sie die Notation `"0b0".U`. Wollen Sie der Konstante eine bestimmte Länge geben verwenden Sie die Notation `"0b0".U(12.W)` für eine 12-Bit breite Konstante mit dem Wert 0. Hier ein Beispiel: $\texttt{Cat}(\texttt{"0b0".U(2.W)}, \texttt{"0b1".U}) = \texttt{"0b001".U(3.W)}$. Weiters, um eine Konstate zu wiederholen (z. B. sign-extension), verwenden Sie die Notation $\texttt{Fill}(20, \texttt{"0b1".U})$. *Hinweis*: Alle Instruktionstypen (außer U-Type wo diese Überlegung entfällt) fordern eine sign-extension auf 32-Bit.

Hier eine Beispiel Berechnung für den `I-Typ`: $\texttt{Cat}(\texttt{Fill}(20, \texttt{ir}(31)), \texttt{ir}(31, 20))$. Der erste Term in der Konkatenation wiederholt das MSB der Konstante 20 Mal (sign-extension). Die unteren 12 Bits werden aus dem $\texttt{imm}_{11:0}$ Feld übernommen.

- `I-Typ`: `I = Cat(Fill(20, ir(31)), ir(31, 20))`
- `S-Typ`: `S = Cat(Fill(20, ir(31)), ir(31, 25), ir(11, 7))`
- `B-Typ`: `B = Cat(Fill(19, ir(31)), ir(31), ir(7), ir(30, 25), ir(11, 8), "0b0".U)`
- `U-Typ`: `U = Cat(ir(31, 12), Fill(12, "b0".U))`
- `J-Typ`: `J = Cat(Fill(11, ir(31)), ir(31), ir(19, 12), ir(20), ir(30, 21), "0b0".U)`

[2] `https://pages.hmc.edu/harris/ddca/ddcarv/DDCArv_AppB_Harris.pdf`
[3] `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 angegeben 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.

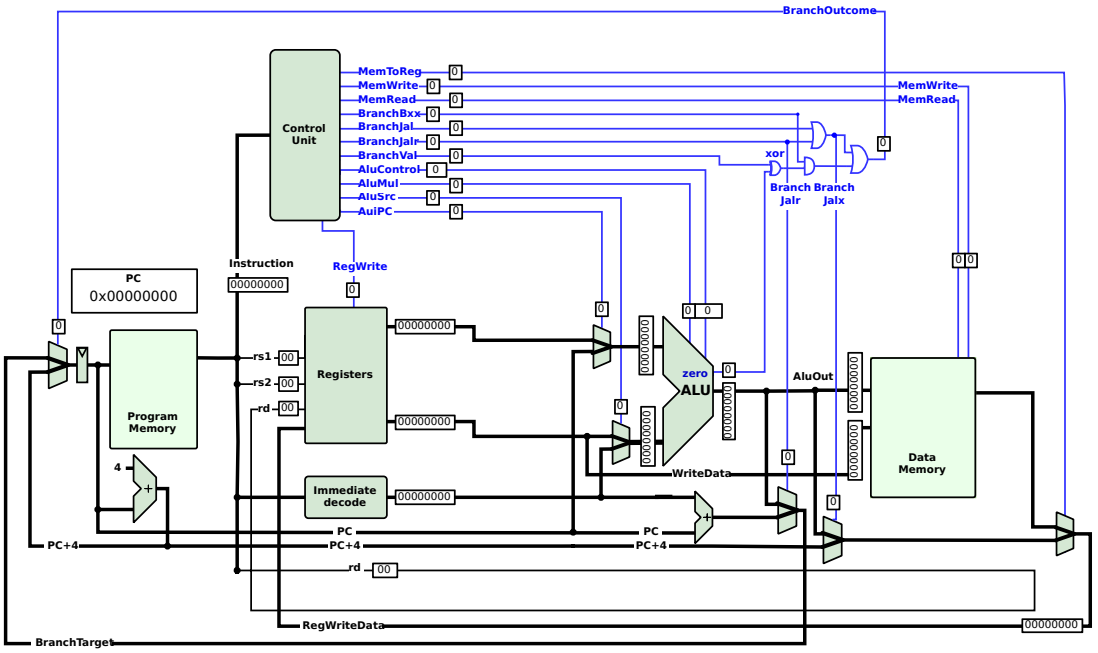
```
MuxLookup(ir_type, 0.U(32.W), Seq(
  0.U(3.W) -> Fill(32, "b0".U)                                          ,
  1.U(3.W) -> Cat(Fill(20, ir(31)), ir(31, 20))                         ,
  2.U(3.W) -> Cat(Fill(20, ir(31)), ir(31, 25), ir(11, 7))              ,
  3.U(3.W) -> Cat(Fill(19, ir(31)), ir(31), ir(7),  ir(30, 25),  ir(11, 8), "0b0".U)  ,
  4.U(3.W) -> Cat(ir(31, 12), Fill(12, "b0".U))                         ,
  5.U(3.W) -> Cat(Fill(11, ir(31)), ir(31), ir(19, 12), ir(20), ir(30, 21), "0b0".U)  ,
  6.U(3.W) -> Fill(32, "b0".U)
))
```

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[4] 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.

[4] `https://mybinder.org/v2/gh/freechipsproject/chisel-bootcamp/master`

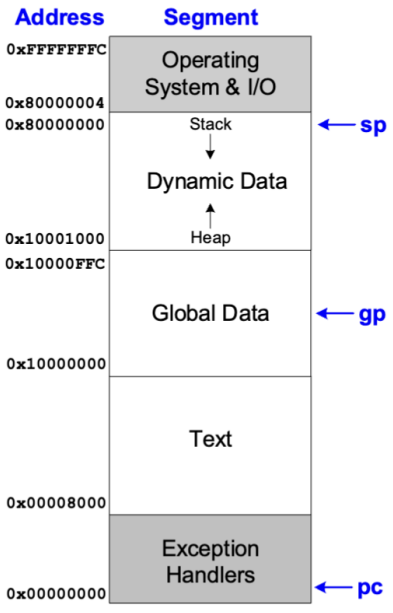
Aufgabe 3: QtRVSim Architektur – Aufbau und Funktionsweise

Nachfolgende Abbildung zeigt Ihnen die simulierte single-stage Mikroarchitektur des QtRVSim[5] 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.



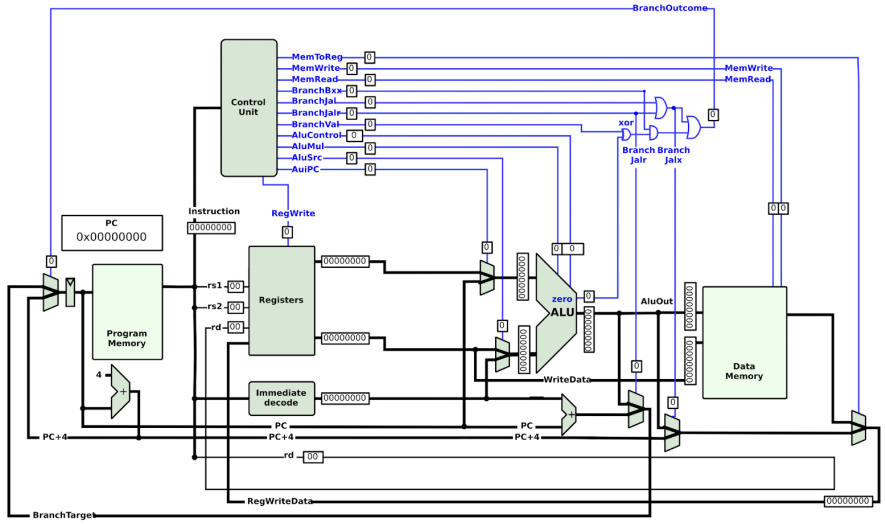
a) Beschreiben Sie kurz die Funktionsweise der folgenden Komponenten in der Mikroarchitektur des Simulators.

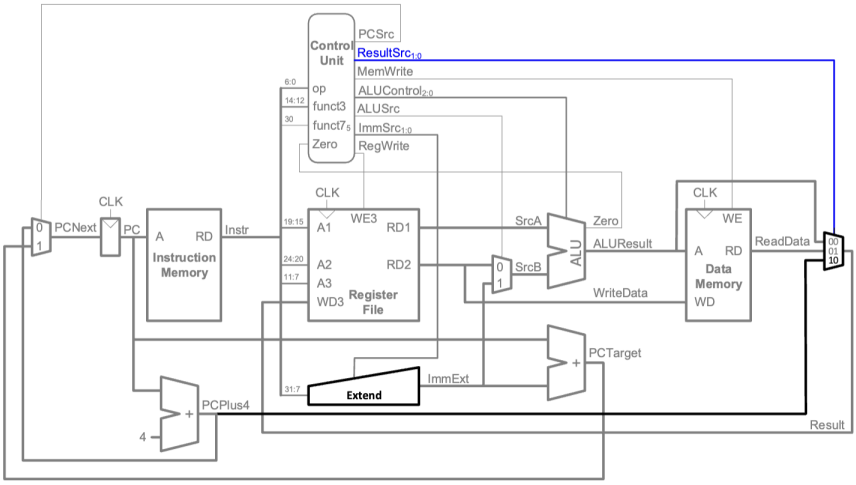
- Registers:
  Ein Speicher von 32 32-bit Wörtern. X0 ist die Konstante 0, alle anderen können vom Programm verwendet werden um Daten zwischenzuspeichern.
- Arithmetic Logic Unit (ALU):
  Hier werden die Berechnungen durchgeführt. Mit AluMul und AluControl kann man steuern, ob die ALU + - and oder or verwendet
- Program Memory:
  Hier stehen alle Instruktionen des aktuellen Programms. Der PC zeigt auf die Adresse, an der die aktuelle Instruktion liegt
- Datenspeicher:
  Hier sind alle Daten, die unser Programm brauchen könnte gespeichert. Hier liegt unter anderem der Stack.
- Program Counter Register (PC):
  Ist zusätzlich zu den regulären Registern ein Register, das auf die nächste Instruktion im Program Memory zeigt.



[5] https://github.com/cvut/qtrvsim

b) Finden Sie mindestens 3 Unterschiede zwischen der Mikroarchitektur aus der Vorleseung 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.





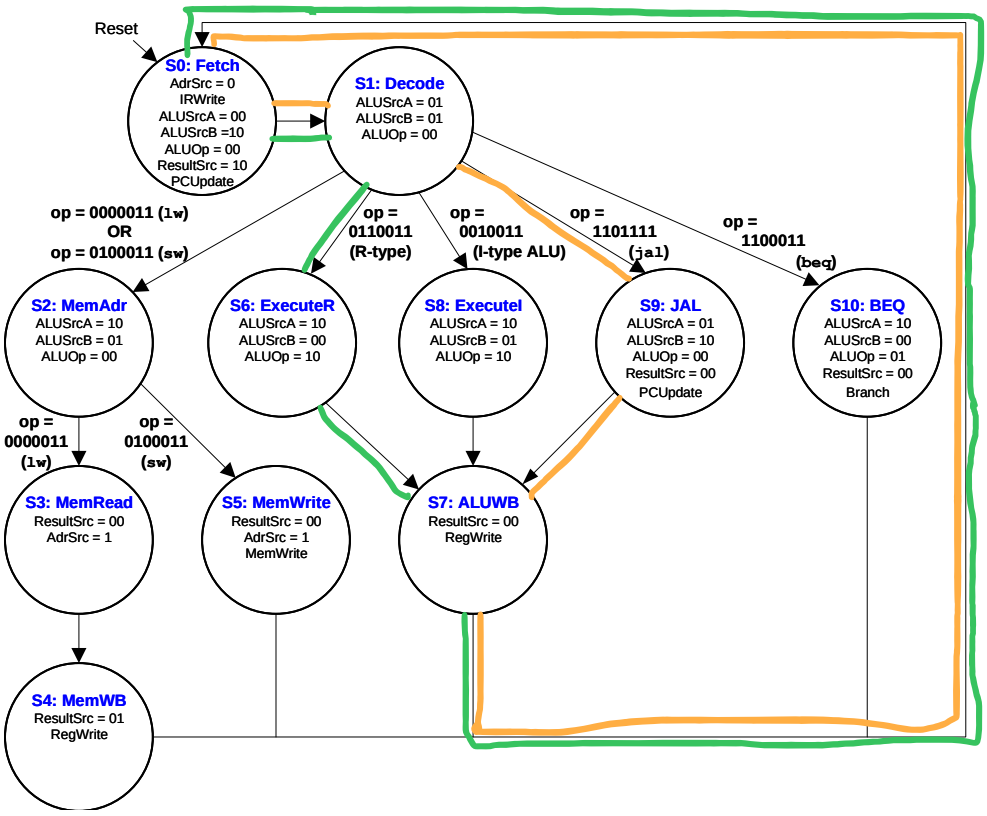
Zusätzlicher MUX vor ALU SrcA:
Wird für den Befehl auipc benötigt

Der MUX nach Data Memory ist anders implementiert:
ResultSrc => BranchJalx, MemToReg
00 => 01
01 => X0
10 => 11

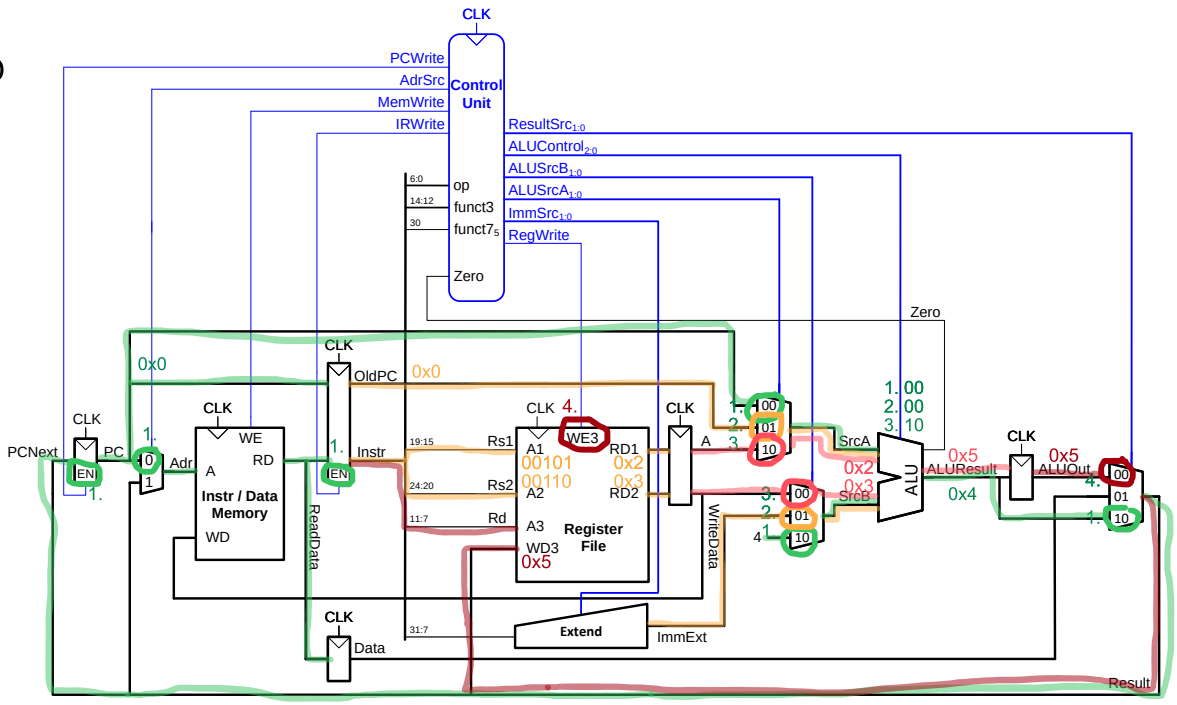
Zusätzlicher MUX BranchJalr:
Ermöglicht jalr
zusätzlicher Abzweig von der ALU in den PC da bei jalr
PC = rs1 + SignExt(imm)
gerechnet wird.

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.



AND



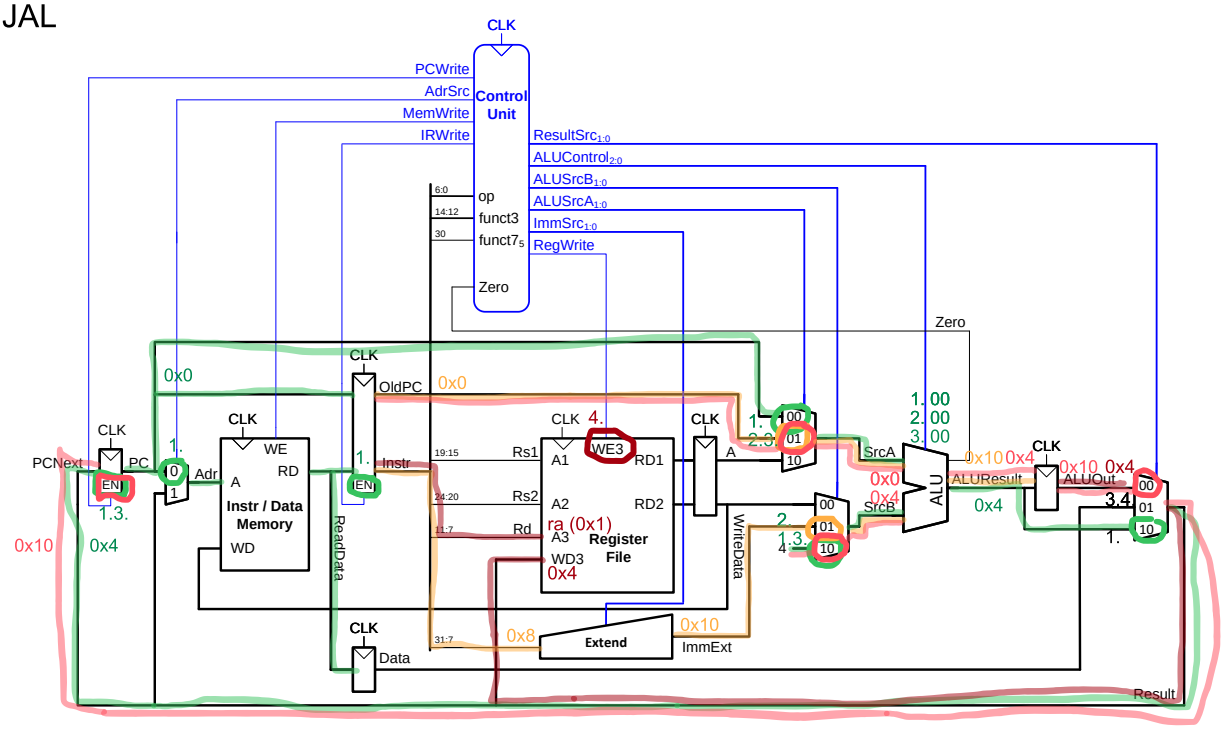
PC = 0x0
PCNext = 0x4
Adr = 0x0
ReadData = and t0, t0, t1 (0x0062F2B3)

t0=0x2
t1=0x3
Ergebnis der Alu wird ignoriert

ALUResult = 0x5

Instruction liegt immer noch im Flip-flop
Rd = t0
WD3 = 0x5

JAL



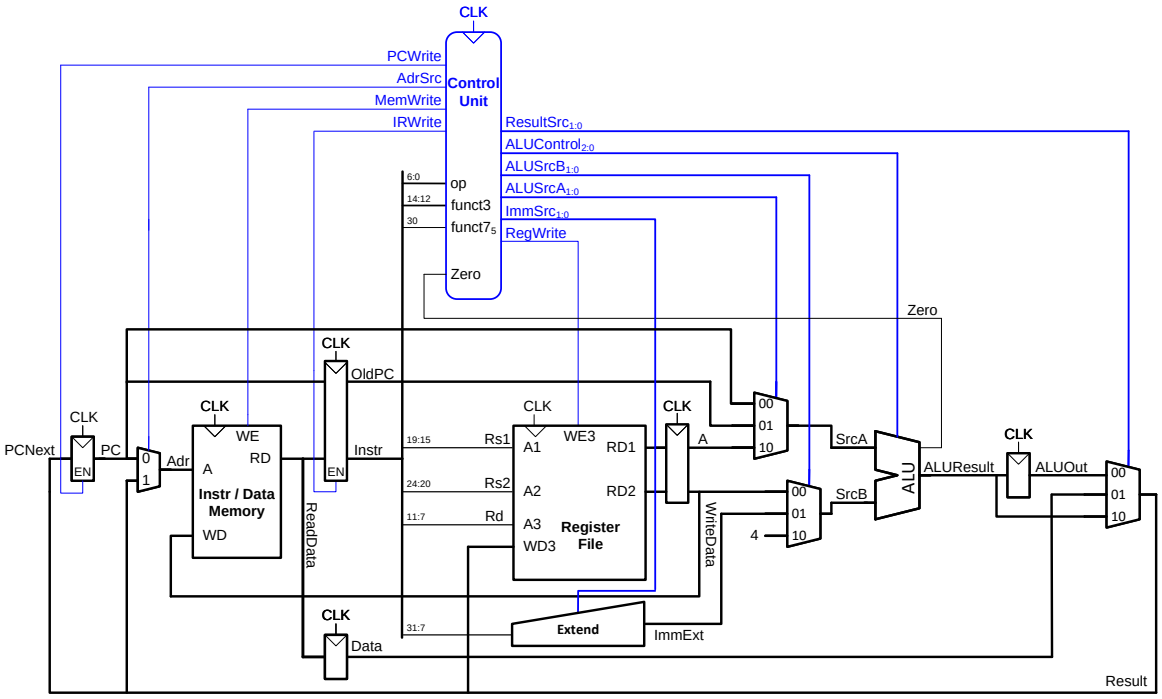
PC = 0x0
PCNext = 0x4
Adr = 0x0
ReadData = jal ra, 0x8 (0x000080EF)

Berechnen der JTA
OldPC = 0x0
ImmExt = 0x10
ALUResult = 0x10

Berechnen der return address
PC+4
und springen
JTA (0x10) to PCNext

Speichern der return address
ra = 0x4

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 μ	Standardformat
PCB2	einseitig	standard	SMD	70 μ	Standardformat
PCB3	einseitig	standard	SMD	35 μ	Sonderformat
PCB4	einseitig	standard	THT	70 μ	Standardformat
PCB5	doppelseitig	standard	SMD	70 μ	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.

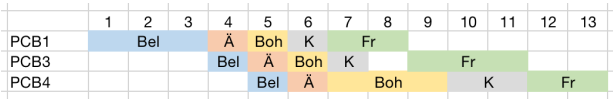
a) Wie lange dauert die Herstellung einer einzelnen Leiterplatte vom Typ PCB1?

$3 + 1 + 1 + 1 + 2 = 8$ ZE

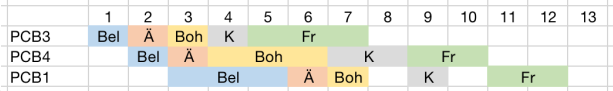
b) Wie lange dauert es, zehn Leiterplatten vom Typ PCB1 in Folge herzustellen?

$3*10 + 1 + 1 + 1 + 2 = 35$ ZE

c) Wie lange dauert die Herstellung der drei Leiterplatten PCB1, PCB3 und PCB4 in der Reihenfolge PCB1–PCB3–PCB4?

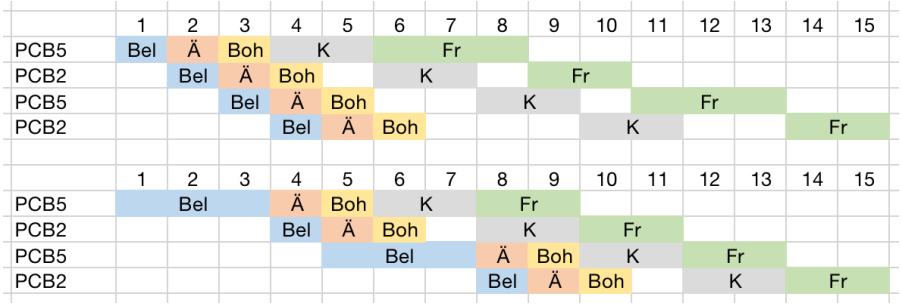


d) Kann die Gesamtdauer aus Teilaufgabe c) durch Umordnen der Reihenfolge verkürzt werden?



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 *i1*, *i2*, *i3* und *i4*. Die Dauer der Ausführung einer Verarbeitungsstufe, abhängig vom Typ der Instruktion, ist in folgender Tabelle angegeben:

Instruktionstyp	Fetch	Decode	Execute	Memory	Write Back	Summe
i1	55ns	50ns	150ns	125ns	0ns	380ns
i2	75ns	50ns	250ns	100ns	55ns	530ns
i3	95ns	75ns	150ns	175ns	120ns	615ns
i4	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.

615ns (längste Instruktion)

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?

250ns (längste Verarbeitungsstufe)

c) Berechnen Sie den theoretischen Durchsatz in MIPS für die Prozessoren aus Unteraufgabe a) und b).

a) 615 ns / instruction
1.6 MIPS
b) 250 ns / instruction
4 MIPS

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?

Durchsatz aus c) 4 MIPS
-25%
3 MIPS
1.5 Mio. pro 500 ms

e) Welche der folgenden Änderungen der Pipelinestruktur bringt eine Verbesserung hinsichtlich des theoretischen Durchsatzes?

(i) Zusammenfassen und Optimieren von *Decode* und *Execute*, sodass alle Instruktionen in der neuen Stufe *Decode & Execute* 150ns benötigen.

(ii) Auftrennen der Memory-Stufe in zwei Stufen *Memory1* und *Memory2*, wobei *i1* und *i2* jeweils 90ns in den beiden neuen Stufen brauchen, *i3* 100ns und *i4* 125ns.

(iii) Eine allgemeine Optimierung, die jede Stufe, die mehr als 100ns um 25ns verkürzt und Stufen mit mehr als 150ns sogar um 55ns.

i) Ja, da jetzt alle 200 ns eine neue Instruktion bearbeitet werden kann.
ii) Nein, da das Bottleneck bei Execute bestehen bleibt.
iii) Ja, alle Stufen (auch das Bottleneck) werden schneller.

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	sw a2, 0(sp)			
2	add a1, a2, a1	sw a2, 0(sp)		
3	div a5, a3, a4	add a1, a2, a1	sw a2, 0(sp)	
4	addi a1, a1 , 1	div a5, a3, a4	add a1, a2, a1	sw a2, 0(sp)
5	sub a6, a5, a1	addi a1, a1 , 1	div a5, a3, a4	add a1, a2, a1
6	mul a1, a5, a6	sub a6, a5, a1	addi a1, a1 , 1	div a5, a3, a4
7	mul a1, a5, a6	sub a6, a5, a1		addi a1, a1 , 1
8	lw a3, 0(sp)	mul a1, a5, a6	sub a6, a5, a1	
9	lw a3, 0(sp)	mul a1, a5, a6		sub a6, a5, a1
10		lw a3, 0(sp)	mul a1, a5, a6	
11			lw a3, 0(sp)	mul a1, a5, a6
12				lw a3, 0(sp)
13				
14	Legende:	stall		
15				

Falsch, es gibt kein forwardig, add muss komplett abgeschlossen sein, bevor addi in D kann.

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`
○ korrekt	◉ korrekt	○ korrekt	○ korrekt
◉ nicht korrekt	○ nicht korrekt	◉ nicht korrekt	◉ nicht korrekt

lw darf nicht vor div sein

12 Takte