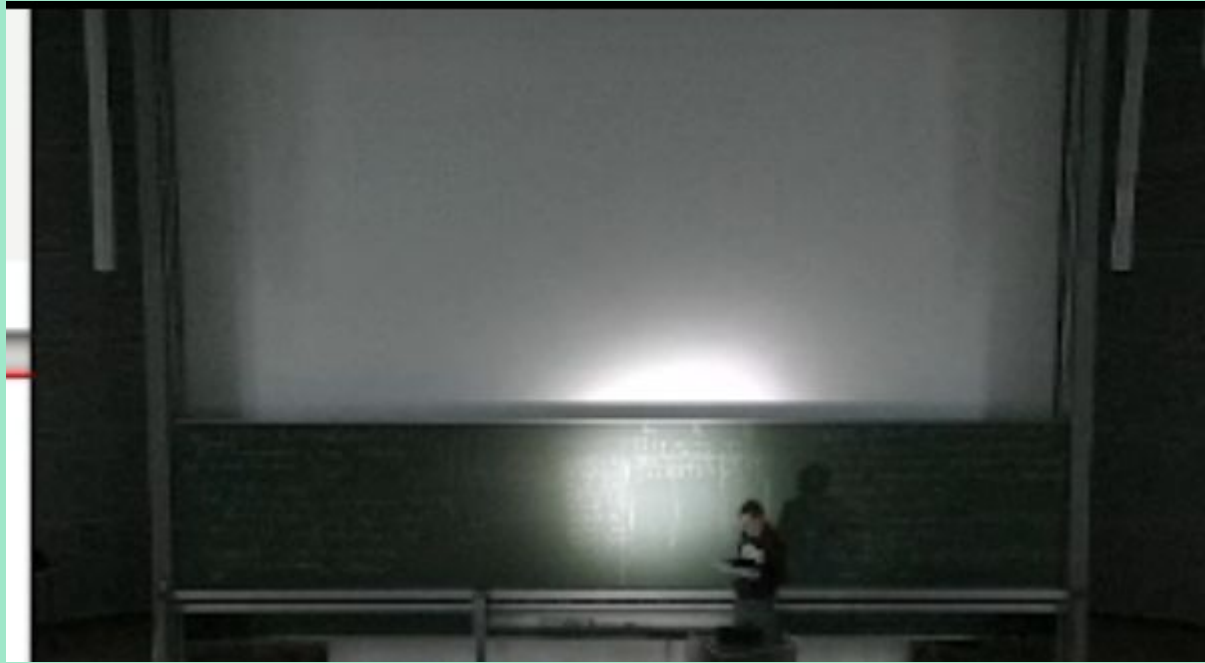




Willkommen zur elften Woche der ERA-Tutorials!



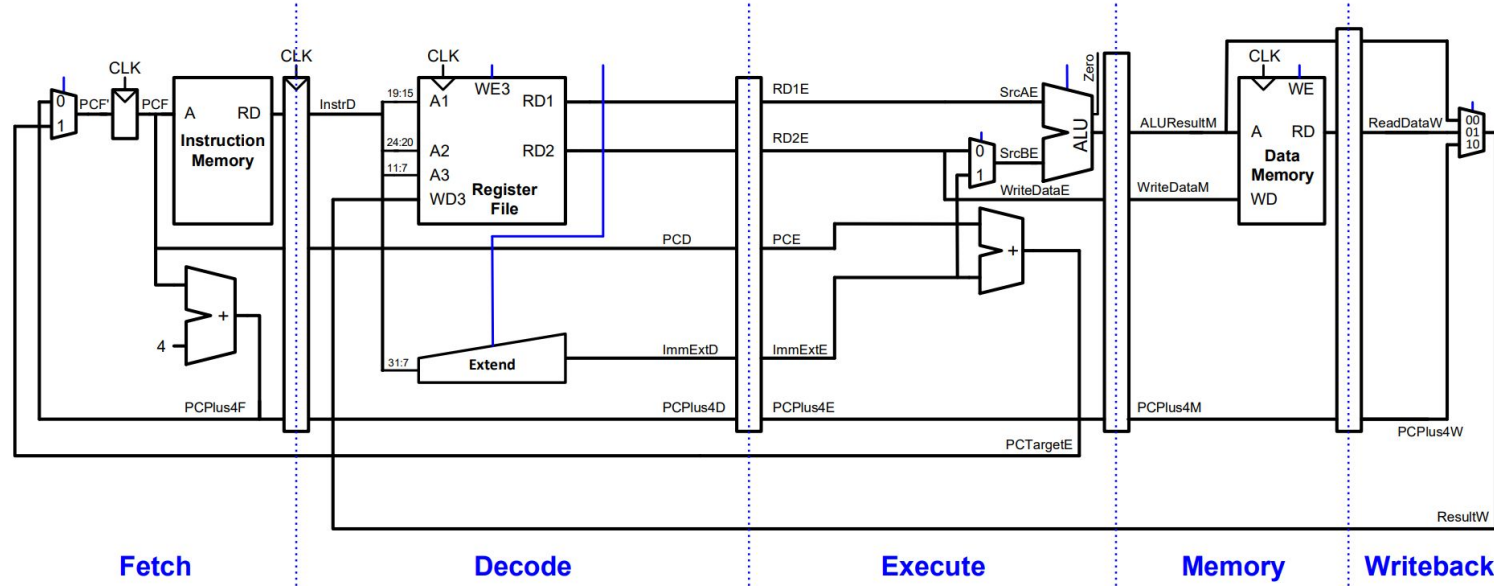
In dieser Woche geht es um Pipelining, den Hazard Unit und Datenabhängigkeiten.



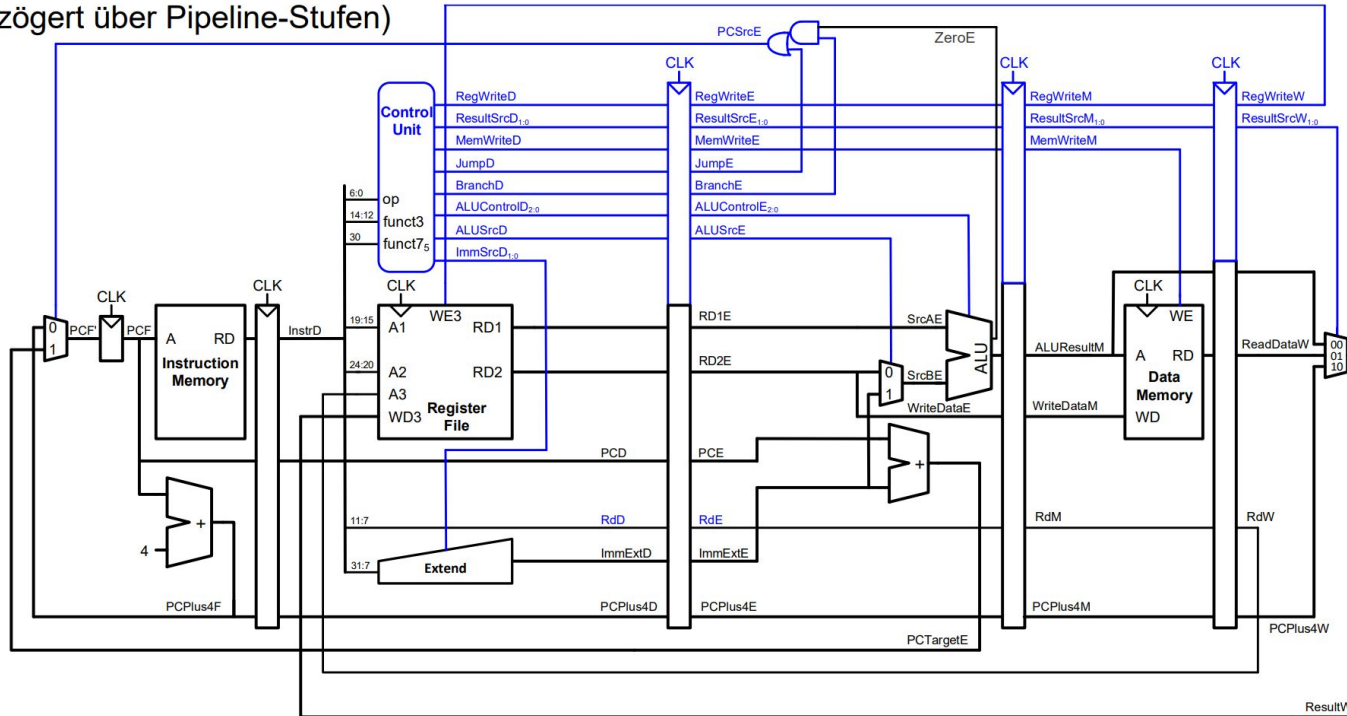
Aufteilung der Befehlsabarbeitung in Phasen

■ Beispiel: Abarbeitung in 5 Schritten:

1. Befehls-Holphase (*Fetch*)
2. Dekodierphase/Lesen von Operanden aus Registern (*Decode*)
3. Ausführung/Adressberechnung (*Execute*)
4. Speicherzugriff (*Memory*)
5. Abspeicherphase (*Writeback*)



- Steuerwerk identisch zu Single-Cycle Prozessor, aber **Steuersignale** laufen mit Instruktionen mit (also verzögert über Pipeline-Stufen)



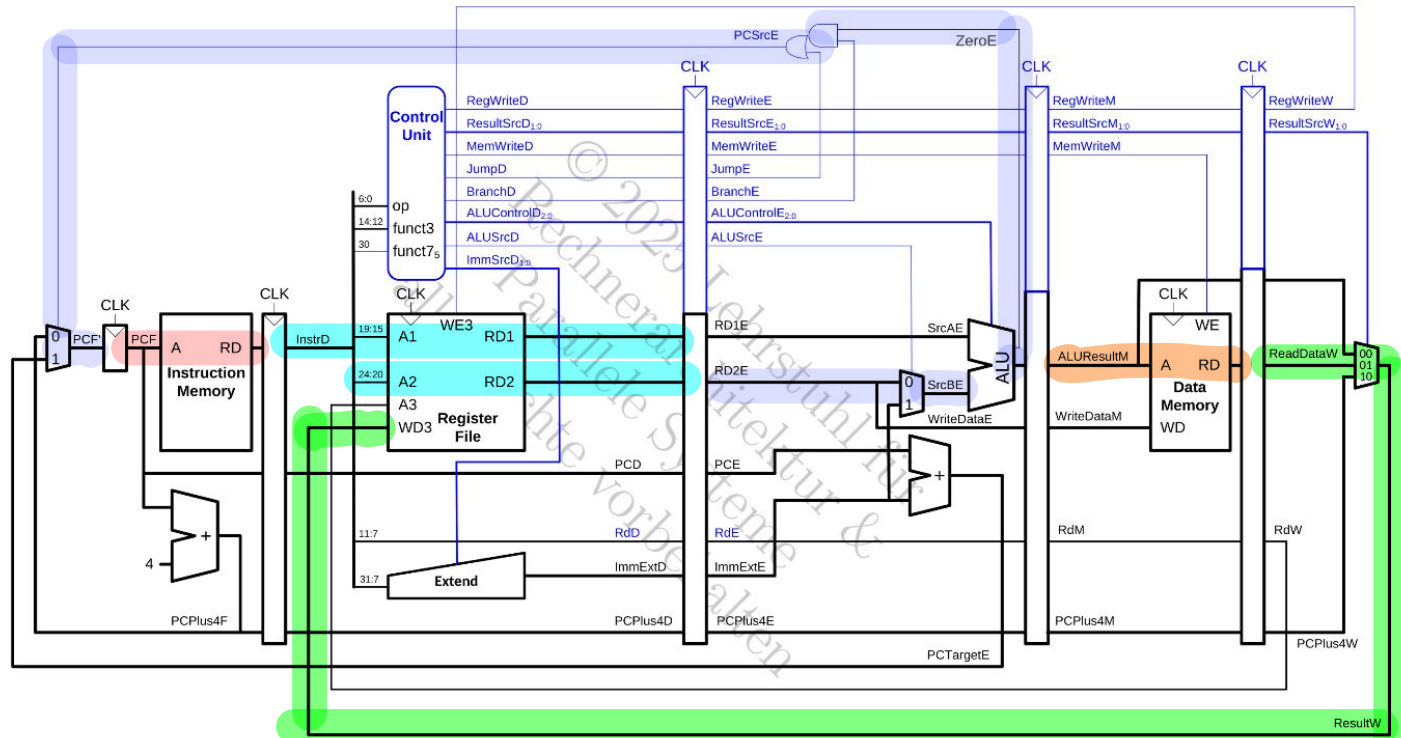
1 Pipelining Speedup

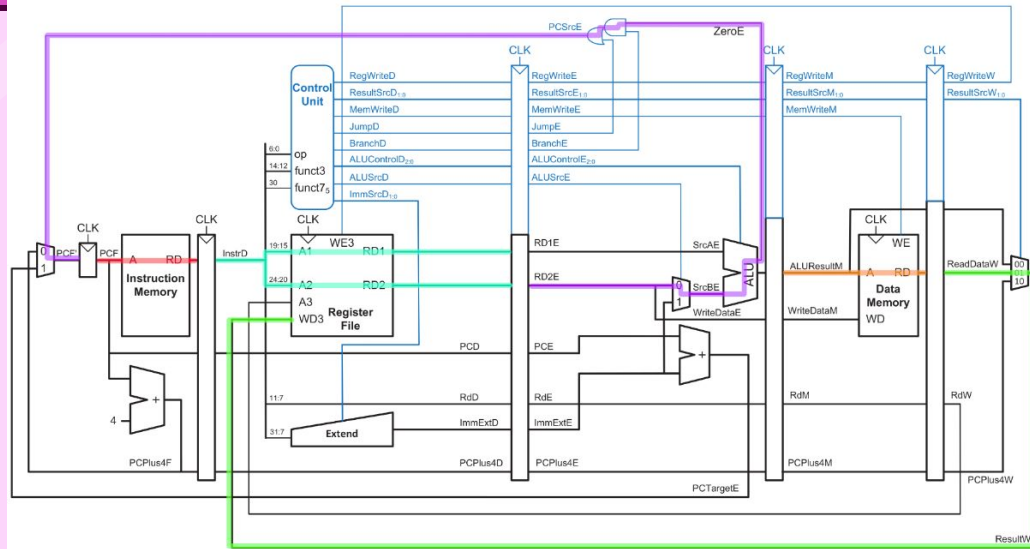
Element	Parameter	Delay (ps)
Register read	t_{RegRead}	40
Register setup	t_{RegSetup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{\text{AND-OR}}$	20
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{MemRead}	200
Register file read	t_{RFRead}	100
Register file setup	t_{RFSetup}	60

Tabelle 1: Propagation Delays

Nimm an, dass die Komponenten des RISC-V Prozessors die in Tabelle 1 angegebenen Verzögerungen haben (eine Verzögerung gilt zwischen jedem Eingang und Ausgang der Komponente). Für nicht gelistete Komponenten soll eine Verzögerung von 0ps angenommen werden.

- a) Zeichne die *kritischen Pfade* aller Pipelining-Stufen des RISC-V-Prozessors in Abbildung 1 ein und bestimme deren Länge (in ps).





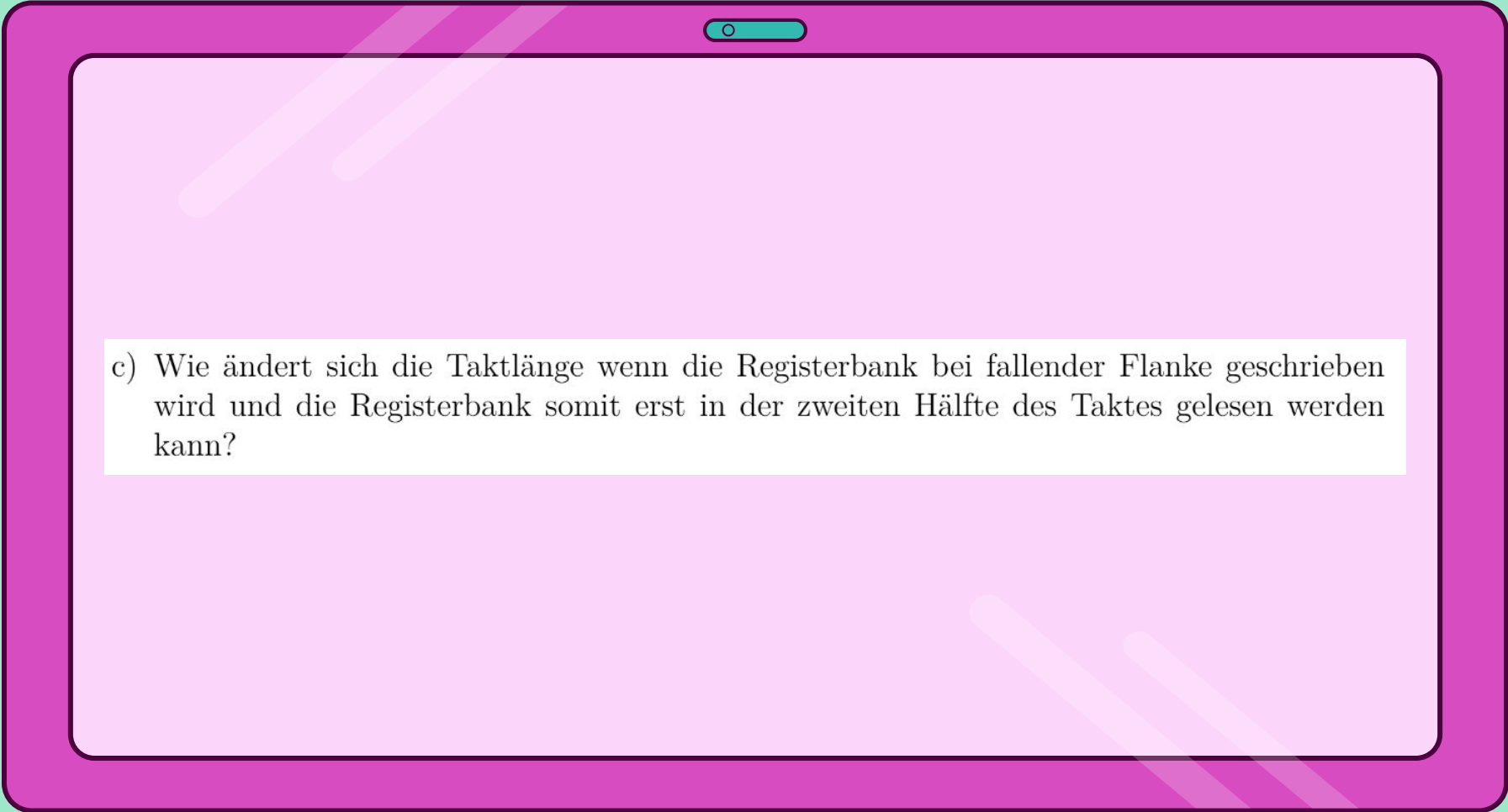
- **Fetch:** $t_{\text{RegRead}} + t_{\text{MemRead}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Decode:** $t_{\text{RegRead}} + t_{\text{RFRead}} + t_{\text{RegSetup}} = 190\text{ps}$
- **Execute:** $t_{\text{RegRead}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{mux}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Memory:** $t_{\text{RegRead}} + t_{\text{MemRead}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Writeback:** $t_{\text{RegRead}} + t_{\text{mux}} + t_{\text{RFSetup}} = 130\text{ps}$

- b) Bestimme die minimale Länge des Taktzyklus für den RISC-V Prozessors mit fünfstufiger Pipeline (Abbildung 1). Welcher Speed-up kann im Vergleich zum Prozessor ohne Pipeline (Abbildung 3, vgl. kombinatorischer Prozessor vom letzten Übungszettel) theoretisch erreicht werden? Warum ist dieser Speed-up nicht gleich der Anzahl der Pipeline-Stufen?

Die Taktlänge ergibt sich durch die längste Pipeline-Stufe. In diesem Falle sind das die *Fetch*-, *Execute*- und *Memory*-Stufe mit 290ps.

Erinnerung: Taktlänge des single-cycle RISC-V Prozessors war 750ps. Damit ergibt sich ein Speedup von $\frac{750\text{ps}}{290\text{ps}} = 2.59$.

Theoretisch könnte ein Speedup von (nahezu) 5 erreicht werden. Allerdings haben nicht alle Stufen der Pipeline die selbe Länge.



c) Wie ändert sich die Taktlänge wenn die Registerbank bei fallender Flanke geschrieben wird und die Registerbank somit erst in der zweiten Hälfte des Taktes gelesen werden kann?

Writeback und Decode dürfen nur einen halben Taktzyklus benötigen. Anders gesagt, muss das Delay dieser Stufen zweimal in einen Taktzyklus passen. Beachte, dass das Ergebnis der Registerbank erst gelesen werden kann, wenn es nach dem halben Takt geschrieben wurde. Deshalb kann das Delay für das Lesen der Instruktion aus dem Register in der Decode Stage ignoriert werden.

Dadurch ergeben sich also folgende Delays:

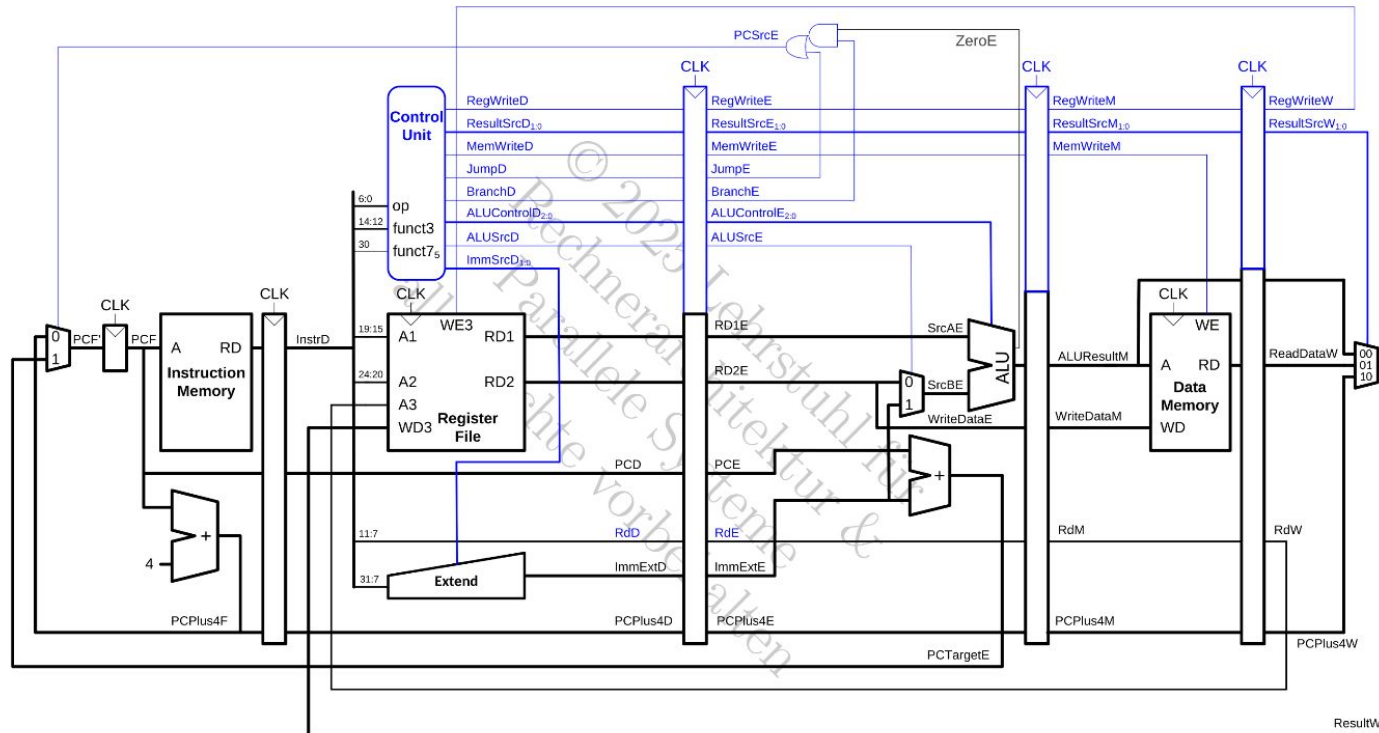
- **Fetch:** $t_{\text{RegRead}} + t_{\text{MemRead}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Decode:** $2 \cdot (t_{\text{RFRead}} + t_{\text{RegSetup}}) = 300\text{ps}$
- **Execute:** $t_{\text{RegRead}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{AND-OR}} + t_{\text{mux}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Memory:** $t_{\text{RegRead}} + t_{\text{MemRead}} + t_{\text{RegSetup}} = 290\text{ps}$
- **Writeback:** $2 \cdot (t_{\text{RegRead}} + t_{\text{mux}} + t_{\text{RFSetup}}) = 260\text{ps}$

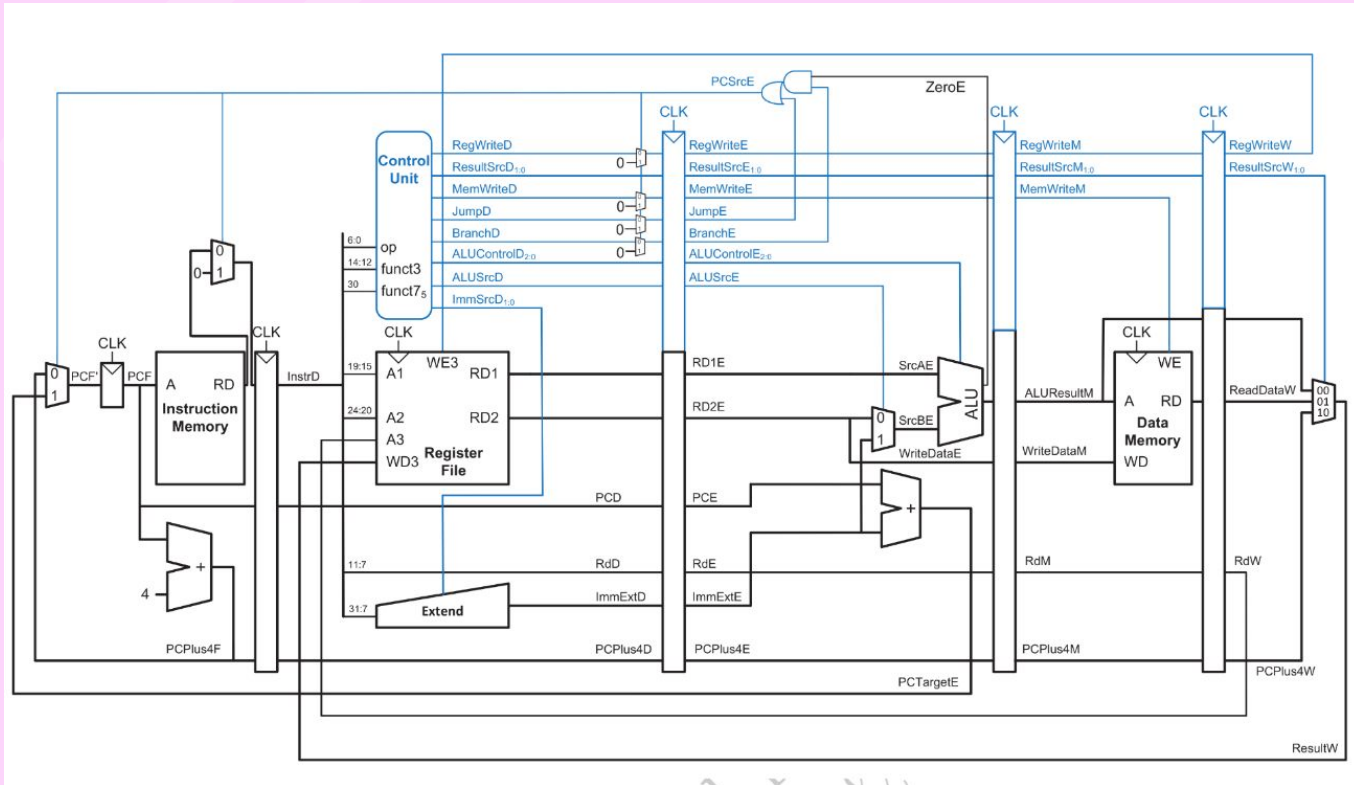
Tatsächlich erhöht sich damit die Taktlänge zu 300ps.

2 Flushen der Pipeline (ohne Hazard Unit)

Aus der Vorlesung wissen wir, dass eine Stage der Pipeline geflushed werden kann indem man rücksetzbare Register zwischen Pipelinestages verwendet. In dieser Übung wollen wir Flushing umsetzen *ohne* die Register selbst zu verändern.

Erweitere dazu den RISC-V Prozessor in Abbildung 1, sodass das Flushen der Pipeline bei den Befehlen `beq` und `jal` unterstützt wird. Füge dazu möglichst wenig Logik zum Datenpfad des Prozessors hinzu. Es darf angenommen werden, dass (zusätzlich zu `addi zero, zero, 0`) eine Instruktion die nur aus Nullen besteht, einem NOP entspricht. Bestehende Komponenten dürfen nicht verändert werden.





Datenabhängigkeiten

■ Read After Write (RAW)

S1: add t0, t1, t2
S2: sub t3, t0, t4

■ Write After Read (WAR)

S1: add t0, t1, t2
S2: sub t1, t5, t4

■ Write After Write (WAW)

S1: add t0, t1, t2
S2: sub t0, t3, t4

- Data Hazards entstehen durch Datenabhängigkeiten
 - Nur RAW Abhängigkeiten können zu Konflikten führen (müssen aber nicht)
 - Abhängigkeit ≠ Konflikt
- Control Hazards entstehen durch Änderung des Kontrollflusses
 - Bedingte und unbedingte Sprünge

- Zwischen zwei direkt aufeinander folgenden Befehlen mit RAW-Abhängigkeit müssen sich **mindestens 3 andere Befehle** befinden

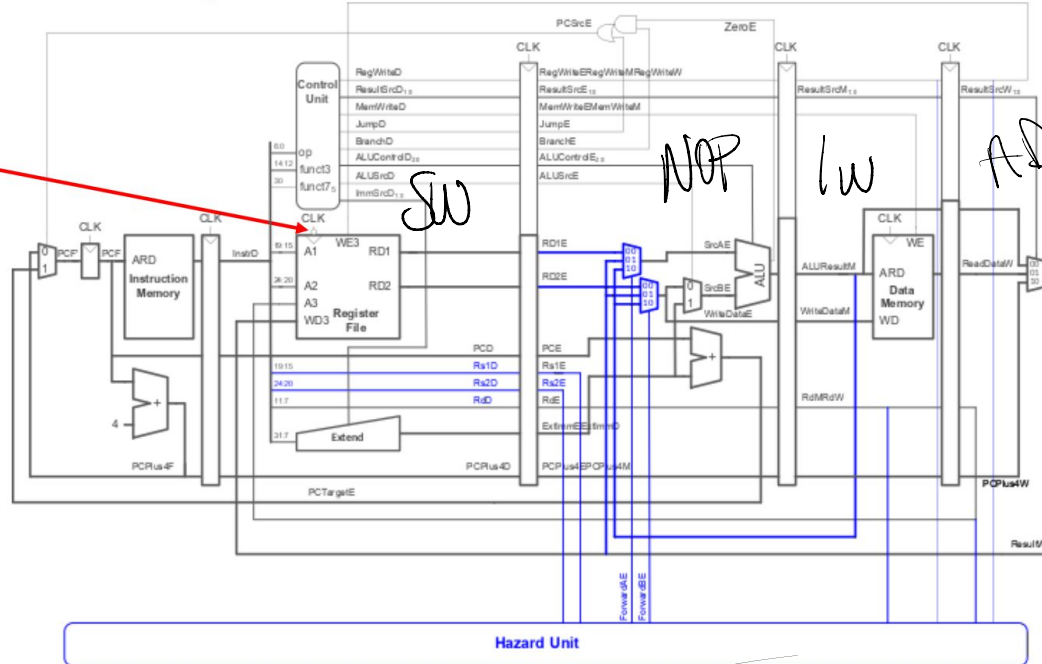
Data Hazards Lösen

- Einfügen von NOPs (NOP = no operation); Stalling
- nop = addi zero, zero, 0

Hazard Unit: Forwarding

```
S1 addi s8, s4, 5
S2 sub s2, s8, s3
S3 or s9, t6, s8
S4 and s7, s8, t2
```

- Hazard Unit prüft ob Quellregister des Befehls in Execute mit dem Zielregister in Memory **oder Writeback** übereinstimmt
- Muss für beide Register geprüft werden
- Kein Forwarding von Memory zu Decode nötig

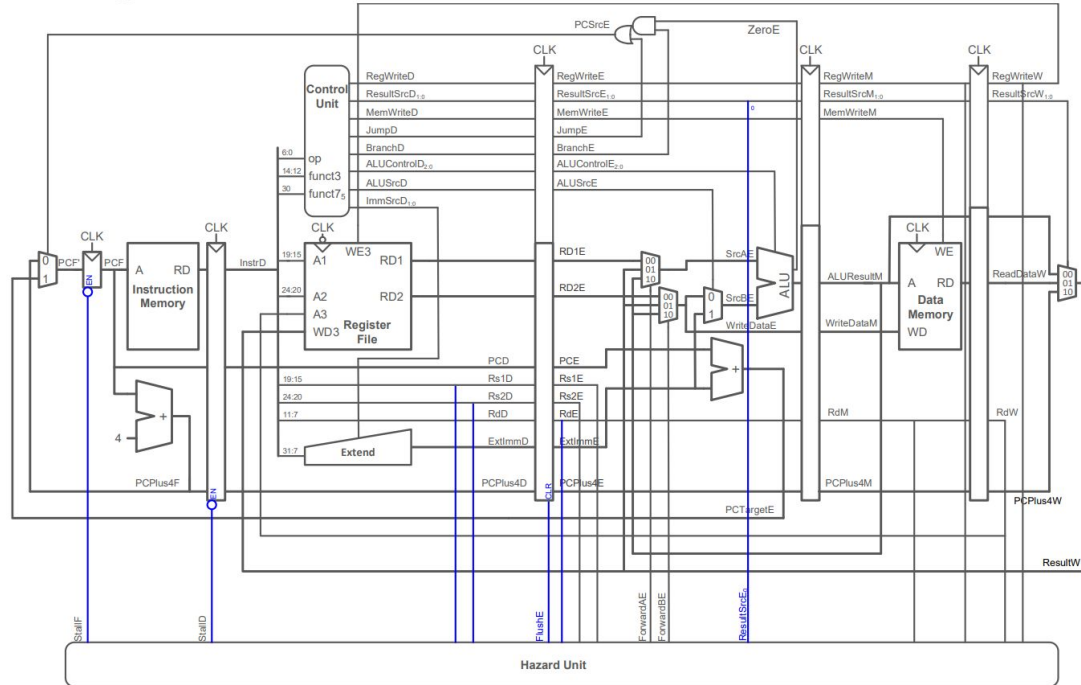


Hazard Unit: Stalling



```
S1 lw    s8, s4, 5
S2 sub   s2, s8, s3
S3 or     s9, t6, s8
S4 and    s7, s8, t2
```

- Überprüfe ob sich `lw` in Execute befindet
- Überprüfe ob der nächste Befehl aus dem Zielregister des `lw` liest
- Verhindere Laden des nächsten Befehls
- Wenn `lw` in Execute ist wird Decode geflushed
 - ☐ Im nächsten Zyklus wird somit ein NOP in Execute geladen



3 Pipelining mit Hazard Unit

Der RISC-V Prozessor mit Pipelining und Hazard Unit aus Abbildung 2 führt den folgenden Programmcode aus.

s1 addi t1, zero, 52 Forwarding
s2 addi t0, t1, -4 Mem to EX
s3 lw t3, 16(t0) Forwarding von Mem to EX
s4 sw t3, 20(t0) Stalling + Forwarding von Writeback zu
s5 xor t2, t0, t3 Execute
s6 or t2, t2, t3 Forwarding

- a) Erläre welche Konflikte in dem Programm auftreten. Welche davon können durch Forwarding gelöst werden und von welcher Pipeline-Stufe muss geforwarded werden? Welche Konflikte müssen durch Stalling gelöst werden?

- Die beiden `addi` Befehle haben eine RAW Abhängigkeit bezüglich `t1`. Also muss das Ergebnis des ersten `addi` aus Memory an Execute geforwarded werden.
- Der zweite `addi` Befehl und der `lw` Befehl haben eine RAW Abhängigkeit bezüglich `t0`. Also muss das Ergebnis des `addi` aus Memory an Execute geforwarded werden.
- Der `lw` Befehl und der `sw` Befehl haben eine RAW Abhängigkeit bezüglich `t3`. Da `lw`

hier schreibt, ist das Ergebnis erst am Ende von Memory verfügbar. Also muss die Pipeline einen Takt gestalled werden. Nach dem gestalled wurde, ist `lw` in Writeback und `sw` in Execute also muss hier wiederum geforwarded werden.

- Der `xor` Befehl und der `or` Befehl haben eine RAW Abhängigkeit bezüglich `t2`. Also muss das Ergebnis des `xor` aus Memory an Execute geforwarded werden.

b) Wie viele Taktzyklen sind erforderlich um alle Befehle in die Pipeline zu laden?

```

addi t1, zero, 52
addi t0, t1, 4
lw t3, 16(t0)
sw t3, 20(t0)
xor t2, t0, t3
or t2, t2, t3
  
```

Zeitschritt	F	D	E	M	W
1	addi				
2	addi	addi			
3	lw	addi	addi		
4	NOP	lw	addi	addi	
5	sw	NOP	lw	addi	addi
6	xor	sw	NOP	lw	addi
7	or	xor	sw	NOP	lw

Hier ist der Programmablauf skizziert.

Zeitschritt	F	D	E	M	W
1	addi	•	•	•	•
2	addi	addi	•	•	•
3	lw	addi	addi	•	•
4	STALL	lw	addi	addi	•
5	sw	STALL	lw	addi	addi
6	xor	sw	STALL	lw	addi
7	or	xor	sw	STALL	lw

Insgesamt benötigt es also 7 Zyklen um alle Instruktionen zu laden.

c) Mit den Delays aus Tabelle 1 hat der Prozessor aus Abbildung 2 eine Taktlänge von 350ps.

Auf dem Prozessor wird ein Programm ausgeführt das aus 25% `lw`, 10% `sw`, 11% `beq`, 2% `jal` und 52 % R- oder I-Typ ALU Instruktionen besteht. Nimm an, dass 40% der `lw` von Befehlen gefolgt werden die das Ergebnis direkt verwenden und 50% der Branches genommen werden. Bei einer falschen Sprungvorhersage müssen 2 Befehle geflushed werden.

Was ist die vorraussichtliche Laufzeit wenn das Programm aus 10^{11} Instruktionen besteht? Das initiale Laden der Pipeline kann ignoriert werden.

Sobald die Pipeline initial geladen ist, wird jeden Takt eine Instruktion abgearbeitet.

Loads benötigen 2 Zyklen wenn ein RAW Konflikt mit einem direkt folgenden Befehl besteht. Also benötigt es im Mittel $0.6 \cdot 1 + 0.4 \cdot 2 = 1.4$ Zyklen pro lw.

Branches benötigen einen Takt wenn nicht gesprungen wird und drei Takte wenn gesprungen wird da zwei falsch geladene Befehle aus der Pipeline geflushed werden müssen. Also benötigen `beq` im Mittel $0.5 \cdot 1 + 0.5 \cdot 3 = 2$ Zyklen. Jumps brauchen immer drei Taktzyklen.

Alle anderen Befehle verwenden einen Zyklus pro Instruktion.

Insgesamt benötigt jede Instruktion des Programms durchschnittlich

$$0.25 \cdot 1.4 + 0.1 \cdot 1 + 0.11 \cdot 2 + 0.02 \cdot 3 + 0.52 \cdot 1 = 1.25$$

Zyklen. Damit ergibt sich eine Laufzeit von $1.25 \cdot 10^{11} \cdot 350\text{ps} = 44\text{s}$.

Beachte: Auf Blatt 9 wurde die Ausführungszeit dieses Benchmarks bereits für Single-Cycle (75s und Multi-Cycle (155s) berechnet.