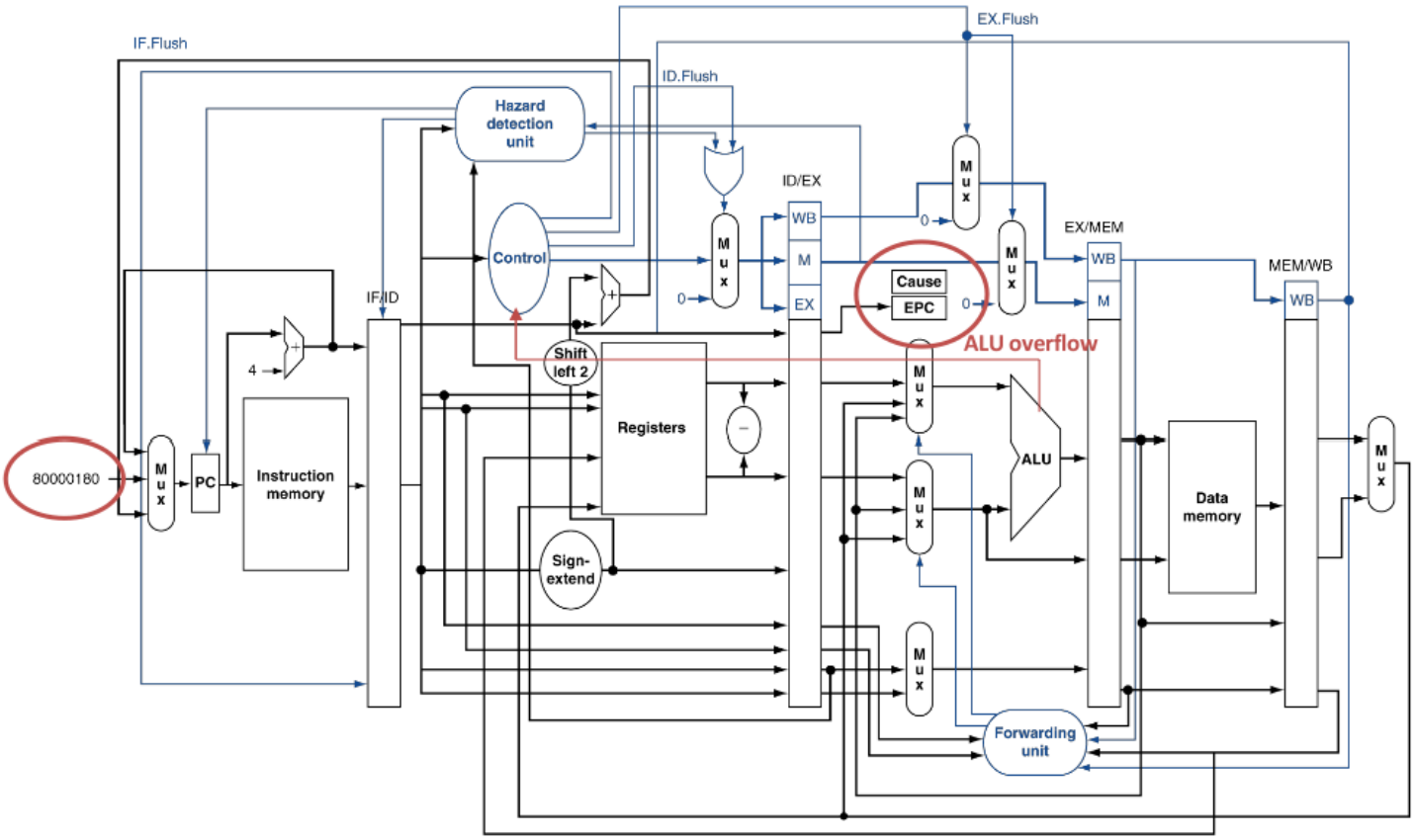
MFG, Thompson

7.1 Pipelining: Exceptions

Angenommen, die MIPS-Instruktion sub $8, $9, $8 werde von der Pipeline lt. VO-Folie 3-118 ausgeführt und löse eine „arithmetic overflow“-Exception aus.



a.) i) Bestimmen Sie die Werte der für die Exception-Behandlung relevanten Steuersignale für jede Pipeline-Stufe, die diese Instruktion durchläuft.

Exception Handling General → Execute the following:
- Execute previous instructions
- Prevent register from being written
- Flush SUB and subsequent instructions – Set them to nops
- Set Cause and EPC register Values

→ EPC: saves current PC+4 (Adjusted by Handler using -4)
→ CAUSE: Read Cause and transfer to corresponding Handler
- Called Handler assumes control

0’ Takt der Exception:
– Execute previous instructions
Instructions in MEM & WB-Flags remain the same, pending on previous instructions
– Prevent $8 from being written
EX.Flush set to 1
Set Flags for MEM & WB to 0, in order to prevent the write completely
– Flush SUB and subsequent instructions – Set them to nops
ID.Flush and IF.Flush set to 1

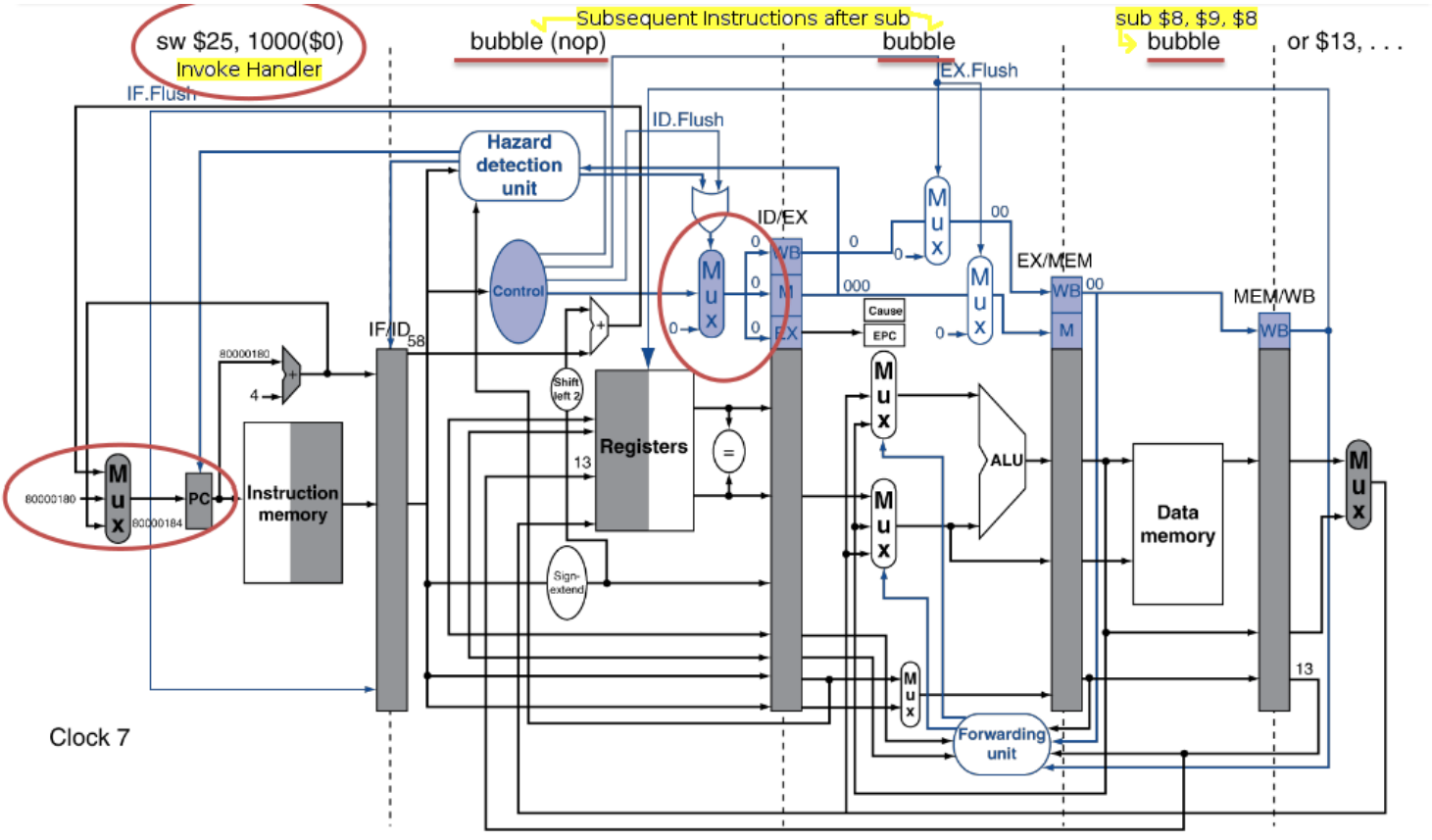
– Set Cause and EPC register Values
Cause ← 1 – Predefined value for “overflow”
EPC ← PC+4
– Called Handler assumes control
Invoke Handler in next tact as IF has command

→ Table:

Takt 0	IC_A	IC_B	sub	IC_X	IC_Y
Operating...	Sub triggers Exception → Invoke Exception handling				
Takt 0’	Flush!	Flush!	Flush!	IC_X	IC_Y

→ Relevant Signals:
- IF.Flush: 1
- ID.Flush: 1
- EX.Flush: 1
- Muxer: EX/MEM: 0
- Disable Forwarding Unit (?)

<u>1’ Takt der Exception</u>:
Invoke Exception handler as ECP registered fault:



→Table:

	IF	ID	EX	MEM	WB
Tact 1	Handler	Flushed	Flushed	Flushed	IC_X

– Der Handler liest in weiterer Folge dann EPC aus um festzustellen, an welcher Adresse der Fehler entstand und terminiert mit einem Error Code.

→ Relevant Signals:
PC ← Handler in Adresse 0x80000180
ii) Erklären Sie zudem, warum manche Steuersignale im ID/EX-Pipeline-Register gespeichert werden, während andere direkt in die EX-Pipeline-Stufe geführt werden.
– ?
Steuersignale wie x.Flush brauchen nicht gespeichert werden, da diese nur bei unmittelbaren Exceptions benötigt werden.
Abhängig von der Pipe können Signale gespeichert werden. Etwa können bei einem Exception Call Parameter wie die Eingänge von EX/WM/WB = 0 gespeichert werden.

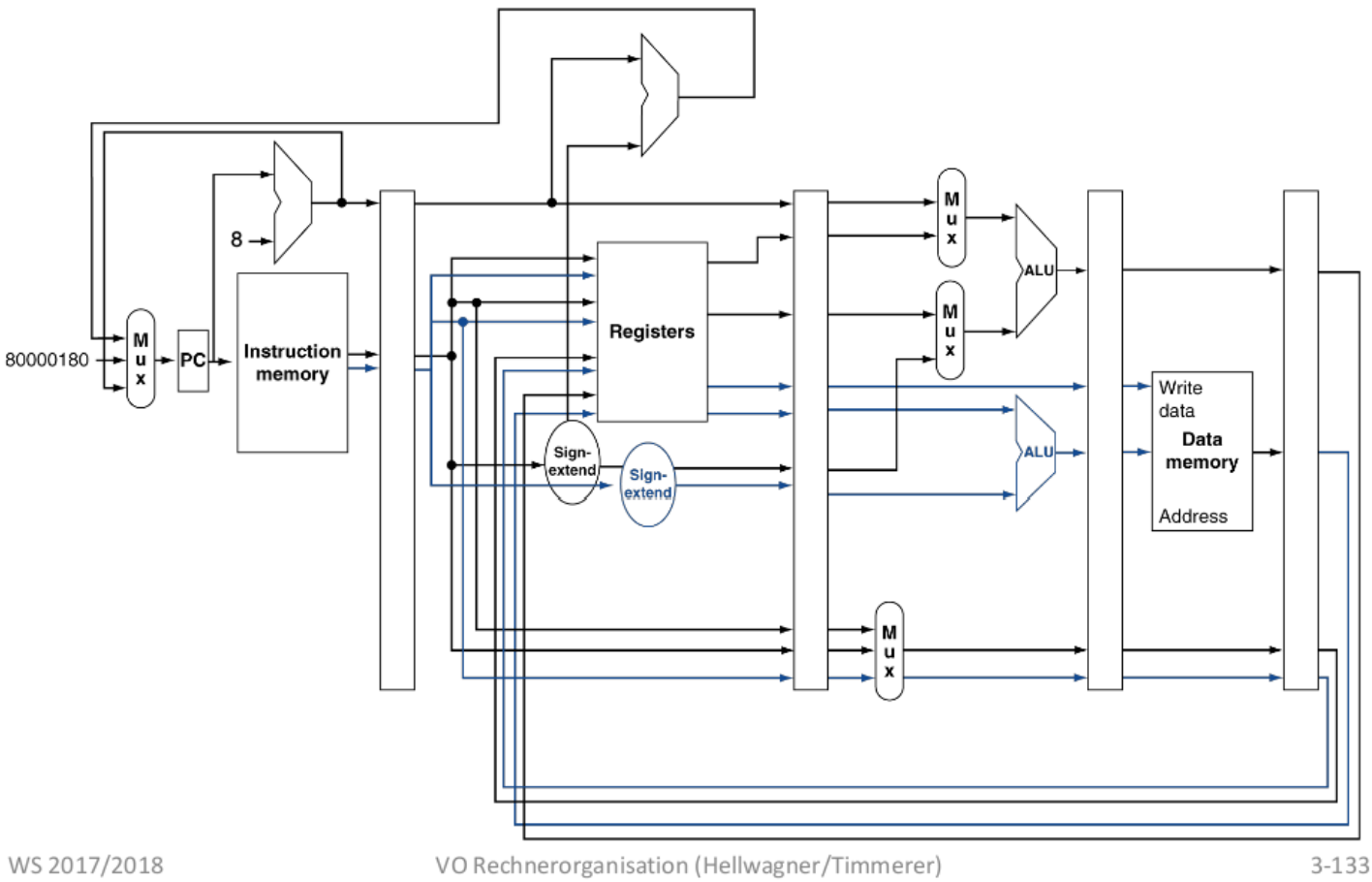
b.) Die Latenz der EX-Pipeline-Stufe könnte verringert werden, wenn die Behandlung der Exception erst in der nachfolgenden Pipeline-Stufe erfolgt. Erklären Sie anhand der gegebenen Instruktion die Nachteile dieses Ansatzes.
→ In nachfolgender Pipeline-Stufe ↔ Weiterer Stall vor Terminierung
→ In MemAccess wird vom Prinzip her Speicher verändert ↔ Bei einer Exception mehr als wie ungewollt.
→

7.2) Statische „Dual Issue“ Prozessoren

Gegeben sei ein statischer „Dual-Issue“-Prozessor (vgl. VO-Folie 3-133) mit fünf Pipeline-Stufen, auf dem ein Programm mit folgenden Befehlshäufigkeiten ausgeführt wird:

Befehl	Häufigkeit
ALU	40%
BEQ – Prediction correct	10%
BEQ – Prediction incorrect	5%
lw	30%
sw	15%

Nehmen Sie an, dass der Prozessor stets zwei beliebige Instruktionen im gleichen Takt ausführen kann (mit Ausnahme von Branch-Befehlen), die Sprungvorhersage in der ersten Pipeline-Stufe erfolgt und Sprünge für „branch“-Befehle in der zweiten Pipeline-Stufe ausgeführt werden. Im Programm sind Branch-Befehle nicht unmittelbar hintereinander angeordnet, die Häufigkeit der Branch-Befehle an geraden bzw. ungeraden Wortadressen ist gleich, Leertakte aufgrund von Datenabhängigkeiten treten nicht auf, und „delay slots“ werden nicht verwendet.



a) Bestimmen Sie den CPI-Wert für die Ausführung dieses Programms.
– Genereller CPI = ½ = 0,5 – 1 Takt ist in der Lage 2 Befehle abzuarbeiten
– Ohne Hazards jeglicher Art

→ Relevant sind die Predictions

BEQ	Prediction correct	Prediction Incorrect
Pipe 1	0,1 / 2 * 0,5 = 0,025	0,05 / 2 * 1,5 = 0,0375
Pipe 2	0,1 / 2 * 0 → Jump here = 0	0,05 / 2 * 1 = 0,025

→ If Incorrect: Consider the ‘+1’ stall for miscalculation

CPI = Genereller CPI + BEQs
CPI = 0,5 + 0,025 + 0 + 0,0375 + 0,025 = 0,5875

b.) Welcher Speedup im Vergleich zu (a) würde erreicht, wenn die Sprungvorhersage perfekt wäre?
Perfekte Sprungprediction → Es gilt genereller CPI als Maß
CPI_old / CPI_new = 0,5875 / 0,5 = 1,175 ↔ 17,5% schneller

c.) Angenommen, der Prozessor habe nur ein Write-Port in der Registereinheit, d.h. es können nicht zwei Befehle parallel in die Registereinheit schreiben (vgl. VO-Folie 3-133). Welcher Speedup wird erreicht, wenn ein zweiter Write-Port hinzugefügt wird?
→ Beeinflussung aller ALU/LW-Befehlen. SW unbetroffen, da nicht in Registereinheit geschrieben wird
* CPI aller ALU+LW = 40%+30% = 70% per Pipe
* CPI aus a.)
→ CPI = 0,5875 + 0,7 * 0,7 = 1,0775 CPI

7.3 Schleifenabrollen, Superskalare Prozessoren

Gegeben sei das untenstehende Code Fragment. Ordnen Sie den Code der zweimal abgerollten Schleife so an, dass er optimal auf einem superskalaren Prozessor mit „Delayed Branching“ entsprechend VO-Folie 3-154 ausgeführt werden kann. Es gelten die Latenzen zwischen abhängigen Befehlen, wie in der am Ende des Übungsblattes stehenden Tabelle angegeben. Wie viele Takte werden pro Ergebniselement benötigt?

Code – Dependencies:

```
0’ Loop:    l.d     $f2, 0($t0)
1’          add.d   $f2, $f2, $f8
2’          l.d     $f12, 0($t1)
3’          add.d   $f12, $f12, $f0
4’          div.d   $f10, $f2, $f12
5’          add.d   $f2, $f10, $f12
6’          mul.d   $f4, $f2, $f10
7’          s.d     $f4, 0($t2)
8’          addi    $t0, $t0, 8
9’          addi    $t1, $t1, 8
10’         addi    $t2, $t2, 8
11’         bne     $t0, $t3, loop
12’         nop
```

Latenzen für Befehlsabhängigkeiten (für Ü 7.3 und Ü 7.5):

Erzeugender Befehl (schreibt Register $x)	Benutzender Befehl (liest Register $x)	Latenz / Zwischentakte (um Leerzyklen zu vermeiden)
FP ALU operation	FP ALU operation	3
FP ALU operation	Store FP double	2
Load FP double	FP ALU operation	1
Load FP double	Store FP double	0
Load integer	Integer operation	1
Load integer	Branch	2
Integer operation	Integer operation	0
Integer operation	Branch	1

Code – Abgerollt & Dependencies

```
0’ Loop:    l.d     $f2,   0($t0)
0.1’        l.d     $f2.1, 8($t0)
            Stall
            Stall
1’          add.d   $f2,   $f2,   $f8
1.1’        add.d   $f2.1, $f2.1, $f8.1
2’          l.d     $f12,  0($t1)
2.1’        l.d     $f12.1, 8($t1)        // Stall
3’          add.d   $f12,  $f12,  $f0     // Stall
3.1’        add.d   $f12.1, $f12.1, $f0.1
            Stall
```

```
            Stall
4’          div.d   $f10,   $f2,   $f12
4.1’        div.d   $f10.1, $f2.1,  $f12.1
            Stall
            Stall
5’          add.d   $f2,    $f10,   $f12
5.1’        add.d   $f2.1,  $f10.1, $f12.1
            Stall
            Stall
6’          mul.d   $f4,    $f2,    $f10
6.1’        mul.d   $f4.1   $f2.1   $f10.1
            Stall
7’          s.d     $f4, 0($t2)
7.1’        s.d     $f4.1,  8($t2)
8’          addi    $t0, $t0, 16            // 8+8
9’          addi    $t1, $t1, 16
10’         addi    $t2, $t2, 16
11’         bne     $t0, $t3, loop
12’         nop                             // Branch delay
```

→ Superskalar gemäß

Parallel instruction execution:

Type	*Pipe Stages*					
INT instruction	IF	ID	EX	MEM	WB	
FP instruction	IF	ID	EX	MEM	WB	
INT instruction		IF	ID	EX	MEM	WB
FP instruction		IF	ID	EX	MEM	WB
INT instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

→ 2 Concurrent instructions: 1 Floating Point (ALU?), 1 Anything else

Code: Geordnet im Superskalaren

Takt \ Pipe	INT	FP
0	l.d $f2 0($t0)	
1	l.d $f2.1 8($t0)	
2	l.d $f12 0($t1)	
3	l.d $f12.1 8($t1)	
4	Addi $t0, $t0, 16	Add.d $f2, $f2, $f8
5	Addi $t1, $t1, 16	Add.d $f2.1 $f2.1, $f8.1
6	Addi $t2, $t2, 16	Add.d $f12, $f12, $f0

7		Add.d $f12.1 $f12.1 $f0.1
8		
9		
10		Div.d $f10, $f2, $f12
11		Div.d $f10.1, $f2.1, $f12.1
12		
13		
14		Add.d $f2, $f10, $f12
15		Add.d $f2.1 $f10.1, $f12.1
16		
17		
18		Mul.d $f4, $f2, $f10
19		Mul.d $f4.1 $f2.1 $f10.1
20	s.d $f4, 0($t2)	
21	Bne $t0, $t3, loop	
22	s.d $f4.1 8($t2)	

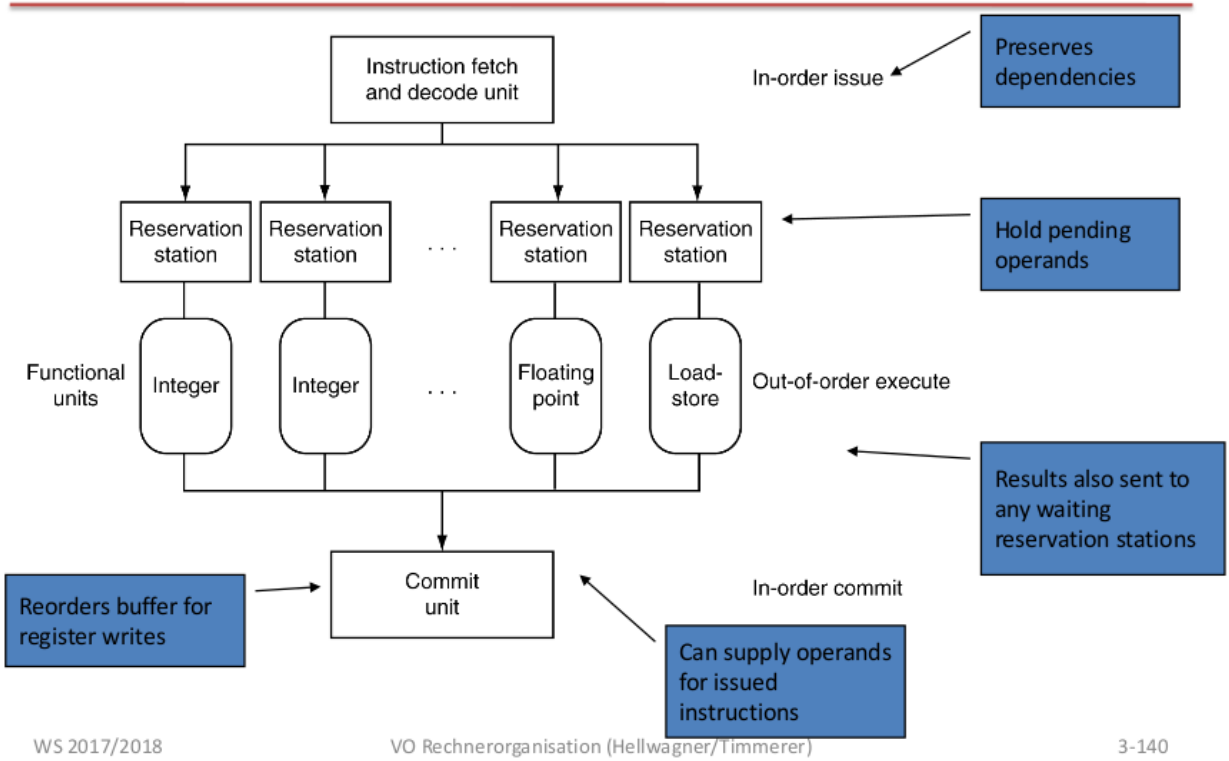
=> Mit 22 Taktzyklen bei 2fachem Abrollen entsteht im Schnitt 1 Ergebnis per 11 Zyklen

7.4 Dynamisches Pipeline-Scheduling

Stellen Sie die Ausführung des (nicht abgerollten und nicht umgeordneten) Codes aus Ü 6.5 auf dem superskalaren Beispielprozessor mit „Dynamic Scheduling“ und „Branch Prediction“ wie auf VO-Folie 3-156 dar. Wie viele Takte werden pro Ergebniselement benötigt?

Code:

```
loop:  l.d     $f4, 0($t0)
       sub.d   $f6, $f4, $f0
       l.d     $f8, 0($t1)
       mul.d   $f10, $f6, $f8
       add.d   $f12, $f10, $f2
       s.d     $f12, 0($t2)
       addi    $t2, $t2, -8
       addi    $t1, $t1, -8
       addi    $t0, $t0, -8
       bne     $t0, $t4, loop
       nop
```



BSP-Prozessor:
→ Es gibt 2 Issue-Pipelines, sodass die Bearbeitung von Instruktionen direkt begonnen wird – Je 1 FP und je 1 'Else'-Befehl
→ Keine Beschränkung seitens Reservierungsstationen oder Funktionseinheiten
→ Zwischen Dependencies gibt es Forwarding, sodass Bearbeitung direkt im selben Takt erfolgen kann.
→ Out-of-order-Completion: Die Commits, also tatsächlichen Write Backs, müssen nicht mit der Issue-Order übereinstimmen
→ Execute-Phasen für verschieden Befehlsgruppen sei wie folgt gegeben; Dauer von Branch und Store-Befehlen ist irrelevant und Branch prediction sei immer richtig.

Befehlsgruppe	FP Load	FP Arithmetik	Übrige
Executes/Takte	2	3	1

Tact	Instruction		Issue		Execute		Commit	
1	loop: l.d	$f4, 0($t0)	INT:	1	INT:	2	INT:	4
2	sub.d	$f6, $f4, $f0	FP:	1	FP:	2	FP:	5
3	l.d	$f8, 0($t1)	INT:	2	INT:	4	INT:	6
4	mul.d	$f10, $f6, $f8	FP:	2	FP:	5	FP:	8
5	add.d	$f12, $f10, $f2	FP:	3	FP:	8	FP:	11
6	s.d	$f12, 0($t2)	INT:	3	irrelevant			
7	addi	$t2, $t2, -8	INT:	4	INT:	6	INT:	7
8	addi	$t1, $t1, -8	INT:	5	INT:	8	INT:	9
9	addi	$t0, $t0, -8	INT:	6	INT:	10	INT:	11
10	bne	$t0, $t4, loop	INT:	7	irrelevant			

=> ??? Per 11 Zyklen 1 Element, muss I-wo was falsch sein

7.5 VLIW

Ordnen Sie den Code der sechsmal abgerollten Schleife aus Ü 6.5 so an, dass er optimal auf einem VLIW-Prozessor entsprechend VO-Folien 3-158 ausgeführt werden kann. Nehmen Sie

an, dass der Prozessor 64 FP-Register besitzt und Branch Prediction implementiert. Die Latenzen zwischen abhängigen Befehlen entnehmen Sie bitte untenstehender Tabelle.

Code:

```
1’      loop:   l.d     $f4, 0($t0)
2’              sub.d   $f6, $f4, $f0
3’              l.d     $f8, 0($t1)
4’              mul.d   $f10, $f6, $f8
5’              add.d   $f12, $f10, $f2
6’              s.d     $f12, 0($t2)
7’              addi    $t2, $t2, -8
8’              addi    $t1, $t1, -8
9’              addi    $t0, $t0, -8
10’             bne     $t0, $t4, loop
11’             nop
```

Gemäß 3-158:

Pipe \ Takt	Mem-Ref	Mem-Ref	FP1	FP2	INT op/branch
1’	l.d $f4.0 0($t0)	l.d $f4.1 8($t0)			
2’	l.d $f4.2 16($t0)	l.d $f4.3 24($t0)			
3’	l.d $f4.4 32($t0)	l.d $f4.5 40($t0)	Sub.d $f6.0, $f4.0, $f0.0	Sub.d $f6.1, $f4.1, $f0.1	
4’	l.d $f8.0, 0($t1)	l.d $f8.1, 8($t1)	Sub.d $f6.2, $f4.2, $f0.2	Sub.d $f6.3, $f4.3, $f0.3	
5’	l.d $f8.2, 16($t1)	l.d $f8.3, 24($t1)	Sub.d $f6.4, $f4.4, $f0.4	Sub.d $f6.5, $f4.5, $f0.5	Addi $t2, -48
6’	l.d $f8.4, 32($t1)	l.d $f8.5, 40($t1)	* Stall due F6		Addi $t1, -48
7’			mul.d $f10.0 $f6.0 $f8.0	mul.d $f10.1 $f6.1 $f8.1	Addi $t0, -48
8’			mul.d $f10.2 $f6.2 $f8.2	mul.d $f10.3 $f6.3 $f8.3	
9’			mul.d $f10.4 $f6.4 $f8.4	mul.d $f10.5 $f6.5 $f8.5	
10’			* Stall due F10		
11’			add.d $f12.0 $f10.0 $f2.0	add.d $f12.1 $f10.1 $f2.1	
12’			add.d $f12.2 $f10.2 $f2.2	add.d $f12.3 $f10.3 $f2.3	
13’			add.d $f12.4 $f10.4 $f2.4	add.d $f12.5 $f10.5 $f2.5	
14’	s.d $f12.0, -40($t2)	s.d $f12.1, -32($t2)			
15’	s.d $f12.2, -24($t2)	s.d $f12.3, -16($t2)			Bne $t0, $t4
16’	s.d $f12.4, -8($t2)	s.d $f12.4, 0($t2)			

(a) Wie viele Takte werden pro Ergebniselement benötigt?

In 16 Zyklen bei 6-fachen Abrollen:

16/6 = 2,67 Takte per Ergebniselement, im Schnitt.

(b) Bestimmen Sie für die Ausführung dieses Codes die Effizienz der Prozessorauslastung und den IPC-Wert.

Prozessorauslatung:
Mit 5 Pipes die über 16 Takten Ergebnisse liefern, ist die Auslastung per Pipe:
Pipe(MemRef_1) = 16 Instruktionen – 100 % ↔ 9 Instruktionen – 56,25%
Pipe(MemRef_2) = 16 Instruktionen – 100% ↔ 9 Instruktionen – 56,25%
Pipe(FP1) = ↔ 9 Instruktionen – 56,25%
Pipe(FP2) = ↔ 9 Instruktionen – 56,25%
Pipe(Int/Branch) = ↔ 4 Instruktionen – 25%
→ Gesamtauslastung = (56,25 * 4 + 25) / 5 = <u>50%</u>

IPC:
IPC = #Befehle / #Takte = (9+9+9+9+4) / 16 = 40/16 = <u>2,5</u>