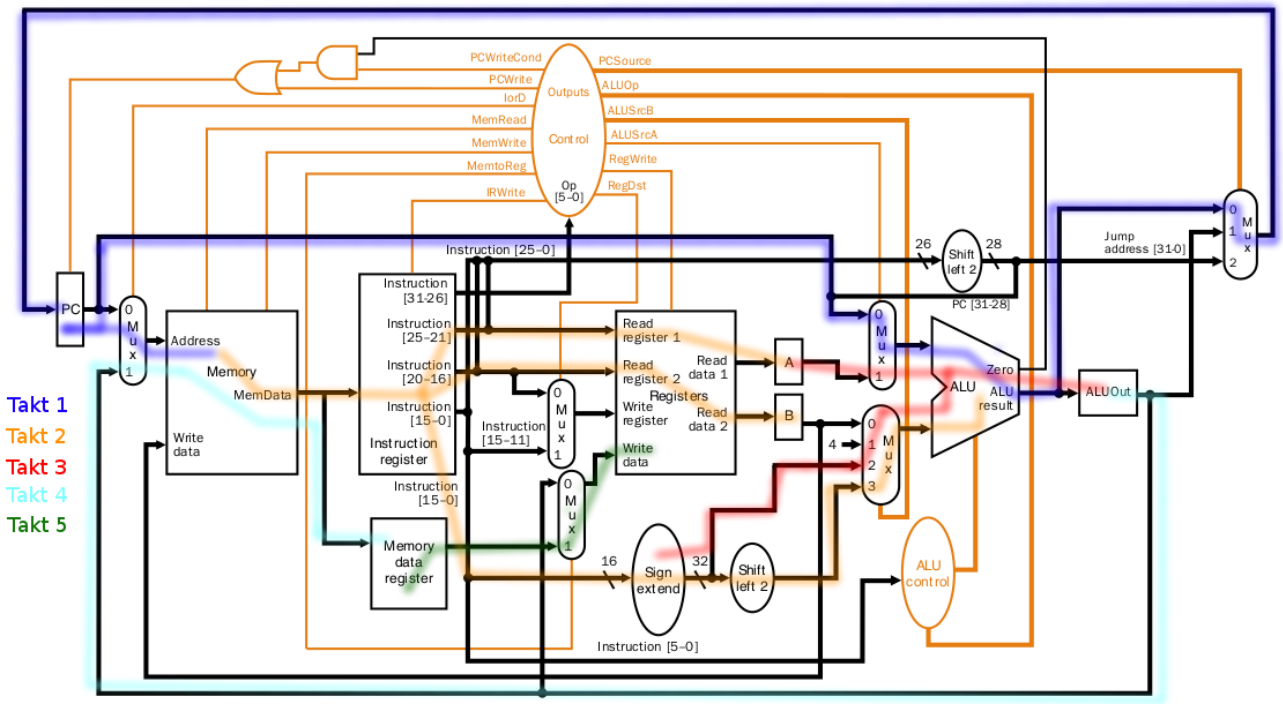
5.1 Multi-Cycle Datenpfad: lw-Instruktion

Erläutern Sie die Ausführung der lw-Instruktion (load word) für den Multi-Cycle-Datenpfad. Welche Vorteile bietet der Multi-Cycle-Datenpfad gegenüber dem Single-Cycle-Datenpfad? Diskutieren Sie dies anhand dieses Befehls.

+ Schnellere serielle Ausführung von Befehlen aufgrund von pipelinen einzelner Befehle
+ Unterschiedliche Befehlsdauer <-> Orientierung am langsamsten
- "Make the common case fast" wird nicht angewandt
- Längste Instruktion bestimmt Taktrate

Mutlicycle: 5 Takte -- Siehe 3-34

1. Instruction Fetch - Same for all instructions
   - 1.1: Fetch IC :: IR = MEM[PC]
   - 1.2: Next Address :: PC += 4
2. Instruction Decode and Operand Fetch from Reg - All Instructs
   - 2.1: Read RS, RT
     - A = REG[IR[25-21]]
     - B = REG[IR[20-16]]
   - 2.2: Compute Branch target address
     - ALUOut = PC + (SE(IR[15-0]) << 2)
3. Execute - Different per Instruction
   - Pending on OP, ONE of following is executed
   - 3.1: ALU-EXEC (R-Type)
     - ALUOut = A ° B
   - 3.2: Mem-Reference
     - ALUOut = A + SE(IR[15-0])
   - 3.3: Branch Executable
     - if A==B: PC = ALUOut # FIN: Beq
   - 3.4: Jump
     - PC = PC[IR[31-28]] || (IR[25-0] << 2) # FIN: J
4. Memory Access - Different per Instruction
   - Pending on OP, ONE of following is executed
   - 4.1: ALU-Exec (R-Type)
     - REG[IR[15-11]] = ALUOut # FIN: R
   - 4.2: Mem-Reference
     - Load: MDR = Memory[ALUOut]
     - Store: MEM[ALUOut] = B # FIN: Store
5. Store Result/Write Back
   - 5.1: Exclusive for loads
     - Load: Reg[IR[20-16]] = MDR;



5.3) **Pipelining**
a.) Bestimmen Sie die Anzahl der Pipelinestufen, die Taktdauer und die Taktfrequenz der Beispiel-Pipeline unter Annahme der Angaben auf VO-Folie 3-44 (Ausführungszeiten der Funktionseinheiten). Wie lange dauert die Ausführung eines einzigen Befehls auf der Beispiel-Pipeline?

→ Pipes = 5

→ Taktfrequenz: 1/200 = 0,005 = 5*10^9 = 5GHz
... *Fetch - Decode - Execute - MemAccess - Write Back* – Jeder Befehl muss diesen Zyklus durchlaufen.

→ Taktdauer: 200ps
... kürzere Befehle besitzen kurze Leerläufe in deren Ausführungeb, welcher jedoch notwendig ist.

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
  - Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

b.) Angenommen es treten keine Leertakte (stalls) auf, welchen Speedup erreicht die Beispiel-Pipeline aus a) gegenüber einem Single-Cycle Datenpfad, der aus den gleichen Funktionsregistern besteht?

If all stages are balanced
- i.e., all take the same time

$$Time\ between\ Instructions_{pipelined} = \frac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ stages}$$

$$Time\ between\ Instructions_{pipelined} = \frac{Time\ between\ instructions_{nonpipelined}}{Number\ of\ Stages}$$

$$Time\ between\ Instructions_{pipelined} * Number\ of\ Stages = Time\ between\ Instructions_{nonpipelined}$$

$$1000\ ps * 5\ Stages = Time\ between\ instructions_{nonpipelined} = 5000\ ps$$

c.) Auf der Pipeline wird folgende Befehlssequenz ausgeführt:

```
and   $10, $2, $3
sw    $11, 4($3)
```

Stellen Sie die Ausführung der oben angeführten Befehlssequenz durch die Beispiel-Pipeline wie auf VO-Folie 3-45 grafisch dar (untere Abbildung). Achten Sie insbesondere auf die zeitliche Anordnung der Zugriffe auf die Registereinheit! Wie lange dauert die Ausführung der Befehlssequenz?

Befehl	Befehlspipe – Derzeitige Ausführung					
and	IF - PC+4	ID - A = $2 - B = $3	Exec - A & B = ALUOut	MemAcc - C = ALUOut	WB/SR - Ignored	
sw		IF - PC+4	ID - A = $3 - B = $11	Exec - A+SE()	MemAcc - mem[] = B	WB/SR - Ignored
Timestamp	200ps	200ps	200ps	200ps	200ps	200ps
Total Time:	200ps	400ps	600ps	800ps	1000ps	1200ps

→ Extime_nonpiped / ExTime_piped = 800/200 = 4 ICs ↔ 4x Schneller

---

5.4 Pipelining: Daten- und Kontrollabhängigkeiten
Gegeben sei folgendes Code-Fragment:

```
        addi $t5, $t5, 42
        addi $t0, $a0, 516
loop:   addi $t0, $t0, -4
        lw   $t1, 0($t0)
        add  $t2, $t1, $t5
        sw   $t2, 0($t0)
        bne  $t0, $a0, loop
        nop
```

a.) Identifizieren Sie alle Daten- und Kontrollabhängigkeiten, die Leerzyklen (Stalls) verursachen (Annahme: ohne Forwarding-Einheit). Wie viele Taktzyklen werden für die Ausführung des gesamten Codes bzw. pro Ergebniselement (d.h. nur die Schleife) benötigt?

Konflikte:

```
        addi $t5, $t5, 42
        addi $t0, $a0, 516
loop:   addi $t0, $t0, -4
        lw   $t1, 0($t0)
        add  $t2, $t1, $t5
        sw   $t2, 0($t0)
        bne  $t0, $a0, loop
        nop
```

Dementsprechend der Code, ohne Forwarding-Einheit:

1’		addi $t5, $t5, 42	
2’		addi $t0, $a0, 516	
3’		**stall**	
4’		**stall**	**– We know result after another 2 Tacts**
5’	loop:	addi $t0, $t0, -4	
6’		**stall**	
7’		**stall**	**– We know result after another 2 Tacts**
8’		lw $t1, 0($t0)	
9’		**stall**	
10’		**stall**	
11’		add $t2, $t1, $t5	
12’		**stall**	
13’		**stall**	**– We know result after another 2 Tacts**
14’		sw $t2, 0($t0)	
15’		bne $t0, $a0, loop	
16’		**stall**	
17’		**stall**	**– We know result after another 2 Tacts**
18’		nop	

Entsprechend unserer Pipe mit:

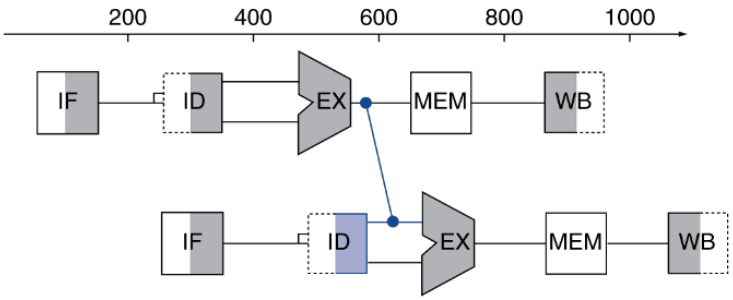
IF	ID	EXEC	MemAcc	WB/SR				
	IF	ID	EXEC	MemAcc	WB/SR			
		IF	ID	EXEC	MemAcc	WB/SR		
			IF	ID	EXEC	MemAcc	WB/SR	

#Tacts – Schleifendurchläufe: 516 / 4 = 129
[2 + 2] + [(5+8) * 129] + [4] = 1811 Zyklen werden durchlaufen.

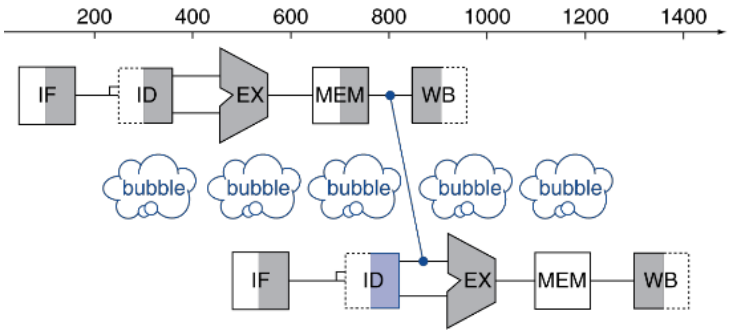
1. Instrukionen vor Loop
2. Instruktionen im Loop – Hier mit der Annahme dass man für den BNE weitere Stalls einführen muss (Eigene Hardware-Optimierung)
3. “Pipe-Drain” → Die Abarbeitung der allerletzten Befehle erfordern weiterhin ihre Zyklen, hier ‘nop’, was per se wiederum den allgemeinen Taktzyklus durchlaufen muss.

(b) Welche aus den Datenabhängigkeiten resultierenden Pipeline-Konflikte können durch eine Forwarding-Einheit gelöst werden? Wie viele Taktzyklen benötigt die Ausführung der Befehlssequenz mit Forwarding-Einheit für die Ausführung des gesamten Codes bzw. pro Ergebniselement (d.h. nur die Schleife)? Wo und warum muss die Hardware Leerzyklen (d.h. Stalls) einfügen?

Forwarding-Einheit – Eliminiert Stalls bei meisten Ops:



Forwarding-Einheit bei LW – Immernoch mit 1 Stall:



Dementsprechend der Code:

1’		addi $t5, $t5, 42	
2’		addi $t0, $a0, 516	
~~3’~~		~~**stall**~~	
~~4’~~		~~**stall**~~	
5’	loop:	addi $t0, $t0, -4	
~~6’~~		~~**stall**~~	
~~7’~~		~~**stall**~~	
8’		lw $t1, 0($t0)	
~~9’~~		~~**stall**~~	
~~10’~~		~~**stall**~~	
11’		**stall**	**– 1 Stall remaining, see above**
12’		add $t2, $t1, $t5	
~~13’~~		~~**stall**~~	
~~14’~~		~~**stall**~~	
15’		sw $t2, 0($t0)	
16’		bne $t0, $a0, loop	
~~17’~~		~~**stall**~~	
~~18’~~		~~**stall**~~	
19’		nop	

Thus:
[2 + 0] + [(5+1) * 129] + 4 = 777 Zyklen für das Programm
- Berechnung analog wie für a.), sprich:
Instruktionen vor der Schleife + Instruktionen in der Schleife + Pipe-Drain

Der eine Stall-Zyklus ergibt sich aus dem Load-Word-Hazard, welcher trotz Forwarding-Unit einen Stall erzeugt. Grund ist der aktuelle Parameterfetch im 4. Zyklus, welcher nicht früher weitergegeben werden kann. (Siehe obrige Grafik)

(c) Ordnen Sie den Code so um, dass er auf einer modifizierten Beispiel-Pipeline mit Unterstützung für „Delayed Branching“ (d.h. Branch-Delay-Slot) möglichst rasch ausgeführt wird (und die Semantik erhalten bleibt). Wie viele Taktzyklen werden in diesem Fall für die Ausführung des gesamten Codes bzw. pro Ergebniselement (d.h. nur die Schleife) benötigt? Wie hoch ist der erzielte Speedup relativ zu bzw. b)?

2’		addi $t0, $a0, 516	
1’		addi $t5, $t5, 42	- Swap 1’ and 2’
5’	loop:	addi $t0, $t0, -4	
8’		lw $t1, 0($t0)	
12’		add $t2, $t1, $t5	
15’		sw $t2, 0($t0)	
16’		bne $t0, $a0, loop	
19’		nop	

“Nein, man kann hier mehr optimieren. lw z.B. verursacht stalls, die man durch Umordnung reduzieren kann (konkret kann man das lw vorziehen, muss dann aber die Indexberechnung anpassen, damit die Semantik erhalten bleibt). Ebenso kann man ausnützen, dass bei Delayed Branching der Befehl nach dem Sprungbefehl _immer_ ausgeführt wird. Man kann also etwas "Sinnvolles" anstelle des "nop" Befehls ausführen. Unter Berücksichtigung von Forwarding und Delayed Branching können durch Umordnung alle stalls entfernt werden.”

Übungsblatt 6

Durch eine Erweiterung der MIPS-Beispiel-Pipeline (vgl. VO-Folie 3-102 ff.), kann die Latenz von bedingten Sprüngen verringert werden. Insbesondere können durch zusätzliche Hardware die Auswertung der Sprungbedingung und die Berechnung der Sprungzieladresse bereits in der zweiten Pipelinestufe erfolgen.

(a) Welche positive Auswirkung hat die genannte Hardware-Erweiterung auf die Latenz von Branch-Befehlen? Welchen negativen Effekt gibt es für die Latenz von Datenabhängigkeiten zu vorausgehenden ALU- bzw. Load-Befehlen? Wie groß sind diese Latenzen (in Takten)?
– Latenz = #Pipes / Taktrate
→ Berechnung der Zielsprungadresse in der 2ten Pipestufe: Branches brauchen nicht mehr 3 sondern 1 Stall.
:: Jump in ID möglich, statt in MEM

(b) Wie viele Takte benötigt die Ausführung des folgenden MIPS-Code-Fragments auf der gegebenen Pipeline? (Ergebnis auf 0,1% genau)

Gesucht: #Takte.
→ Ohne Ohne Stalls:

```
1’              xor     $s0, $zero, $zero
2’              addi    $t0, $a0, 600
3’      outer:  addi    $t1, $a1, 200
4’      inner:  addi    $s0, $s0, 1
5’              addi    $t1, $t1, -1
6’              bne     $t1, $a1, inner
7’              addi    $t0, $t0, -2
8’              bne     $t0, $a0, outer
```

Die Innere Schleife wird genau 200x durchlaufen.
Die Äußere Schleife genau 300x, welche per Durchlauf die Innere Schleife zurücksetzt, dementsprechend wird die Innere Schleife insgesamt 60.000 durchlaufen.
→ [1’+2’] + [3’+7’+8’]*(300 Wiederholungen) + [4’+5’+6’] * (60.000 Wiederholungen)
→ 2+ 3*300 + 3*60.000 + 4 = 2+900+180.000+4 = 180.906 Zyklen,

→ Mit Stalls:

```
1’              xor     $s0, $zero, $zero
2’              addi    $t0, $a0, 600
3’      outer:  addi    $t1, $a1, 200
4’      inner:  addi    $s0, $s0, 1
5’              addi    $t1, $t1, -1
6’              Stall durch DD
7’              bne     $t1, $a1, inner
8’              Stall durch BNE – NOP <!-- There already is a stall.
9’              addi    $t0, $t0, -2
10’             Stall durch DD
11’             bne     $t0, $a0, outer
12’             Stall durch BNE – NOP <!-- There already is a stall.
```

Per innere Schleife gibts einen Stall, generiert durch R-Type/Branch-Dependency.
Per äußere Schleife gibts einen Stall, genau wie bei der inneren Schleife.
→ 2 + 600/2 * (200 * (3 + 1 + 1) -1 + 3 + 1 + 1) -1 = 301.201

```
2                               – XOR, ADD
600/2                           – Outer
        200                     – Inner
                3+1+1           – addi,addi,bne + Stall + Branch prediction
        - 1 + 3 + 1 + 1         – Letzte Prediction richtig, Befehle Outer, Stall, Prediction Wrong
-1                              – Letzter Sprung richtig
```

6.3 Branch Prediction
Das MIPS-Code-Fragment aus Ü 6.2 wird auf der dort beschriebenen Beispiel-Pipeline mit Unterstützung für Branch Prediction ausgeführt. Die Pipeline besitzt eine Branch History Tabelle (BHT) und einen Branch Target Buffer (BTB) , die beide in der ersten Pipelinestufe ausgelesen werden. Das PC-Register enthält somit am Beginn der zweiten Pipelinestufe die vorhergesagte Sprungadresse (oder PC+4, falls BHT oder BTB keine gültigen Einträge für die aktuelle Befehlsadresse enthalten). Beantworten Sie folgende Fragen:

a.) Wie viele Taktzyklen werden benötigt, wenn eine 1-Bit BHT implementiert ist? (Ergebnis auf 0,1% genau)

→ Bei einer 1-Bit BHT generiert man das Bit je nach vorherigen Sprung.
Unter der Annahme dass das BHT-Bit 0, also Falsch, gesetzt wird, ergibt sich folgendes:

```
1’              xor     $s0, $zero, $zero
2’              addi    $t0, $a0, 600
3’      outer:  addi    $t1, $a1, 200
4’      inner:  addi    $s0, $s0, 1
5’              addi    $t1, $t1, -1
6’              Stall durch DD
7’              bne     $t1, $a1, inner
8’              Stall durch BNE – NOP <!-- There already is a stall. Would be failed twice
9’              addi    $t0, $t0, -2
10’             Stall durch DD
11’             bne     $t0, $a0, outer
12’             Stall durch BNE – NOP <!-- There already is a stall. Would be failed twice
```

Taktzyklen: 2 + 600/2 * (200 * (3+1) + 2 + 3 + 1) + 2 = 241.804 Zyklen

b.) Wie viele Taktzyklen werden benötigt, wenn eine 2-Bit BHT laut VO-Folie 3-109 implementiert ist? (Ergebnis auf 0,1% genau)
→ BHT: 00 ↔ 01 ↔ 10 ↔ 11 <==> ¬J ↔ ¬J ↔ J ↔ J

Taktzyklen: 2 + (600/2) * (200 * (3+1) + 1 + 3+1)+1+4 = 241.507

c.) Verwenden Sie den MARS BHT Simulator (siehe Howto im Moodle), um die Anzahl der richtig bzw. falsch vorhergesagten Sprünge in a) und b) zu verifizieren.
BHT: 1 Bit

Initialize with ¬Take Jump:	correct |	¬correct
	59400	| 600
	298	| 2
Initialize with Take Jump:	59401	| 599
	299	| 1

BHT: 2 Bit

Initialize with ¬Take Jump:	correct |	¬correct
	59698	| 302
	297	| 3
Initialize with Take Jump:	59700	| 300
	299	| 1

– Die Adresse wird masked und in die BHT gespeichert.

6.4) Pipelining bedingter Move
Geben Sie eine möglichst kurze Sequenz von MIPS-Assembler-Befehlen an, welche das Maximum der Werte der Register $t0 und $t1 im Register $t2 ablegt. Verwenden Sie dabei keine Pseudo-Befehle. Die Befehlssequenz soll auf der Beispiel-Pipeline mit Branch Prediction laut Ü 6.3 ausgeführt werden. Nehmen Sie an, dass das Maximum mit der gleichen Wahrscheinlichkeit in $t0 bzw. in $t1 zu finden ist und dass die Branch Prediction in der Hälfte aller Fälle richtig ist. Wie viele Takte benötigt die Ausführung dieser Befehlssequenz im Mittel?

```
main:                           ||      max:
        li $t0, 70              ||              slt $t3, $t0, $t1
        li $t1, 50              ||              add $v0, $zero, $t0
        jal max                 ||              beq $zero, $t3, end
                                ||              add $v0, $zero, $t1
        move $a0, $v0           ||---------------------------------------
        li $v0, 1               ||      end:
        syscall                 ||              jr $ra
                                ||
        li $v0, 10              ||
        syscall                 ||
```

… für die Subfunktion max :: $t2 = $v0;
→ Branch Prediction: Init True
=> 3 Takte wenn BEQ wahr
=> 4 Takte wenn BEQ nicht wahr

→ Branch Prediction: Init False
=> 4 Takte wenn BEQ wahr

=> 5 Takte wenn BEQ nicht wahr

b.) Angenommen, der MIPS-Befehlssatz sei um einen Befehl movz $rd, $rs, $rt (kein Pseudo-Befehl) erweitert worden, der den Wert des Registers $rs nach $rd kopiert, falls $rt den Wert 0 hat; anderenfalls hat dieser Befehl keinen Effekt. Benutzen Sie diesen Befehl, um eine effizientere Lösung für a) ohne Verwendung von Verzweigungs- oder Sprungbefehlen anzugeben. Um welchen Prozentsatz sinkt die Ausführungszeit der Befehlssequenz auf der Beispiel-Pipeline gegenüber a)? Wodurch wird die Leistungs-steigerung verursacht?

Movz := [rd ← rs; if rt==0]
slt $t3, $t1, $t2
add $t2, $t0, $zero
movz $t2, $t1, $t3

→ Branch Prediction N/A, Fixer Wert von 3 Takten per Programmaufruf.
→ (a) Leistungssteigerung = (3+4+5)/3 = 4/3 = 1.33 → 33% schnellere Programmausführung.
5.) Gegeben sei folgendes MIPS-Assemblercodefragment, das auf einem Pipelined-Prozessor mit „Delayed Branching“ (1 Takt Branch Delay) ausgeführt wird. Die Latenzen zwischen abhängigen Befehlen sind in untenstehender Tabelle angegeben (FP = floating point).

```
      # initialize c as $f0, d as $f2 – not shown
loop: l.d   $f4, 0($t0)       # load x[i]
      sub.d $f6, $f4, $f0     # x[i] - c
      l.d   $f8, 0($t1)       # load y[i]
      mul.d $f10, $f6, $f8    # (x[i] – c) * y[i]
      add.d $f12, $f10, $f2   # (x[i] - c) * y[i] + d
      s.d   $f12, 0($t2)      # store result element z[i]
      addi  $t2, $t2, -8
      addi  $t1, $t1, -8
      addi  $t0, $t0, -8
      bne   $t0, $t4, loop
      nop
```

a.) Identifizieren sie alle Daten- und Kontrollabhängigkeiten, die Leerzyklen (Stalls) verursachen. Wie viele Takte werden für ein Ergebniselement (durch s.d gespeicherter Wert) benötigt?

```
      # initialize c as $f0, d as $f2 – not shown
loop: l.d   $f4, 0($t0)       # load x[i]
      sub.d $f6, $f4, $f0     # x[i] - c
      l.d   $f8, 0($t1)       # load y[i]
      mul.d $f10, $f6, $f8    # (x[i] – c) * y[i]
      add.d $f12, $f10, $f2   # (x[i] - c) * y[i] + d
      s.d   $f12, 0($t2)      # store result element z[i]
      addi  $t2, $t2, -8
      addi  $t1, $t1, -8
      addi  $t0, $t0, -8
      bne   $t0, $t4, loop
      nop
```

[1.1] f4 → 2’ generiert 1 Stall
[2.1] f6 → 4’ generiert 2-1 Stalls
[3.1] f8 → 4’ generiert 1 Stall

[4.1,4.2,4.3] f10 → 5’ generiert 3-0 Stalls
[5.1,5.2] f12 → 6’ generiert 2-0 Stalls

[9.1] t0 → 10’ generiert 1 Stall

Code:

1’	l.d	$f4, 0($t0)
1.1	stall	
2’	sub.d	$f6, $f4, $f0
2.1	stall	
3’	l.d	$f8, 0($t1)
3.1	stall	
4’	mul.d	$f10, $f6, $d8
4.1	stall	
4.2	stall	

```
4.3     stall
5’      add.d   $f12, $f10, $f2
5.1     stall
5.2     stall
6’      s.d     $f12, 0($ts)
7’      addi    $t2, $t2, -8
8’      addi    $t1, $t1, -8
9’      addi    $t0, $t0, -8
9.1     stall
10’     bne     $t0, $t4, loop
11’     nop
```

=> 11 + 1 + 1 +1 + 3 + 2 + 1 = 20 Takte
!! Nop immer mitzählen wegen Branch-Delayer

b.) Optimieren Sie den Code durch Umordnen von Befehlen so, dass er auf dem gegebenen Prozessor möglichst schnell ausgeführt wird. Wie viele Takte werden für die Verarbeitung eines Ergebniselements (z[i]) durchschnittlich benötigt?

```
1’      l.d     $f4, 0($t0)
9’      addi    $t0, $t0, -8
3’      l.d     $f8, 0($t1)
2’      sub.d   $f6, $f4, $f0
8’      addi    $t1, $t1, -8
4’      mul.d   $f10, $f6, $d8
4.2     stall
4.3     stall
5’      add.d   $f12, $f10, $f2
5.1     stall
5.2     stall
6’      s.d     $f12, 0($t2)
7’      addi    $t2, $t2, -8
10’     bne     $t0, $t4, loop
11’     nop
```

=> 11 Takte + 4 Stalls = 15 Takte

Optimalste form: ld; ld; addi; addi; sub.d; addi; mul.d; add.d; s.d; bne; s.d;
→ s.d nach bne möglich, wegen Branch Delay Slot :: bne; s.d;
=> Bester Fall: 4 Stalls => 14 Takte

c.) Rollen Sie die Schleife zweimal ab (zwei Kopien des Code-Fragments in einer Schleifeniteration), und ordnen Sie den Code so um, dass er auf dem gegebenen Prozessor möglichst schnell ausgeführt wird. Wie viele Takte werden nun pro Ergebniselement benötigt?

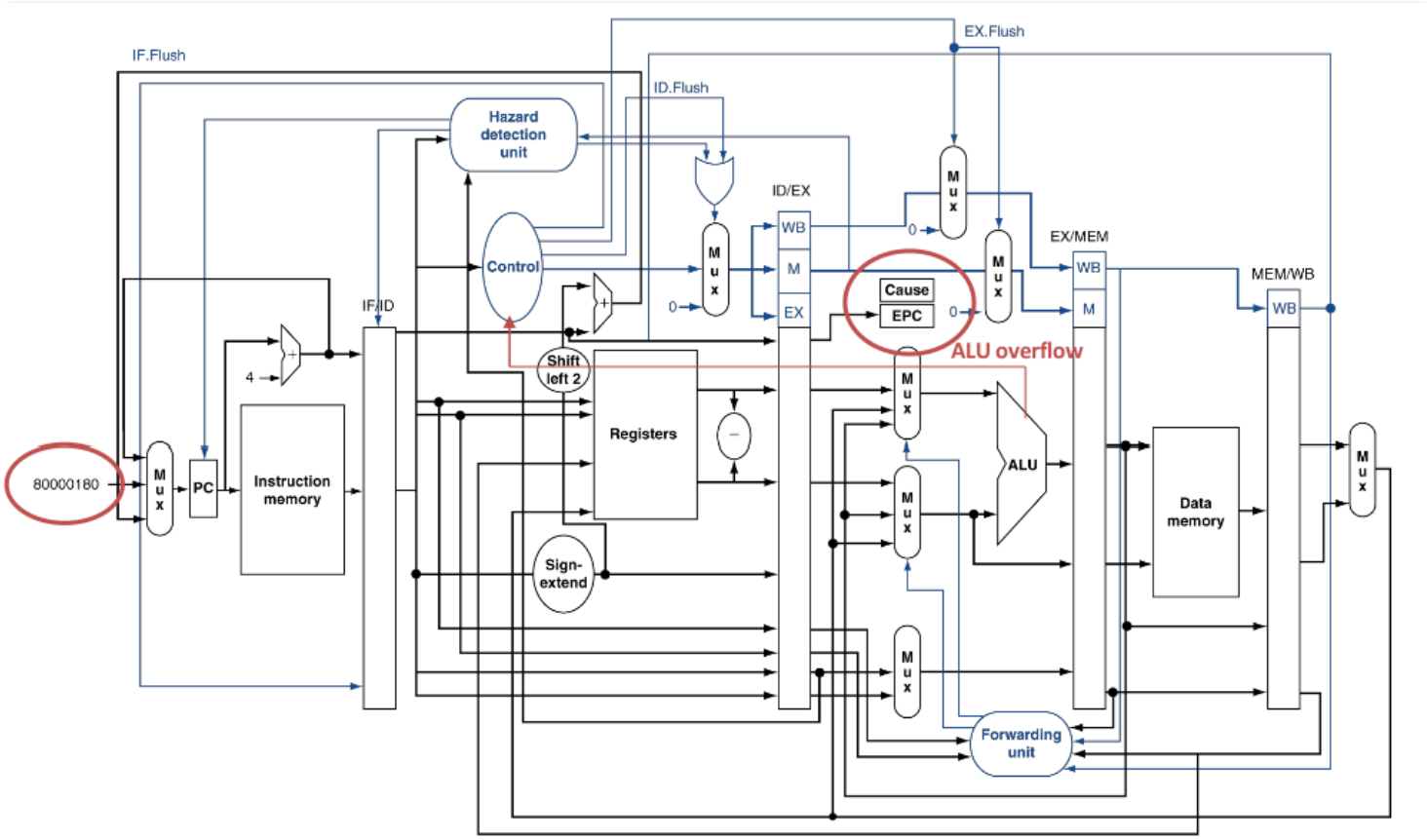
```
### Optimiert:
ld      $f4, 0($t0)         # x[i]
ld      $f4.1, -8($t0)      # x[i - 1]
sub.d   $f6, $f4, $f0
sub.d   $f6.1, $f4.1, $f0.1
ld      $f8, 0($t1)
ld      $f8.1, -8($t1)
mul.d   $f10, $f6, $f8
mul.d   $f10.1, $f6.1, $f8.1
addi    $t2, $t2, -16
addi    $t1, $t1, -16
add.d   $f12, $f10, $f2
add.d   $f12.1, $f10.1, $f2.1
sd      $f12, 16($t2)
addi    $t0, $t0, -16
bne     $t0, $t4, loop
sd      $f12.1, 8($)
```

→ 16/2 Takte = 8 Takte

d.) Wodurch wird die Leistungssteigerung beim Abrollen von Schleifen im Allgemeinen erreicht?
Duplikat → Mehrfache Ergebnisberechnung => Zyklen / #Unrolls bei erweiterten Möglichkeiten Stalls zu Elimineren

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

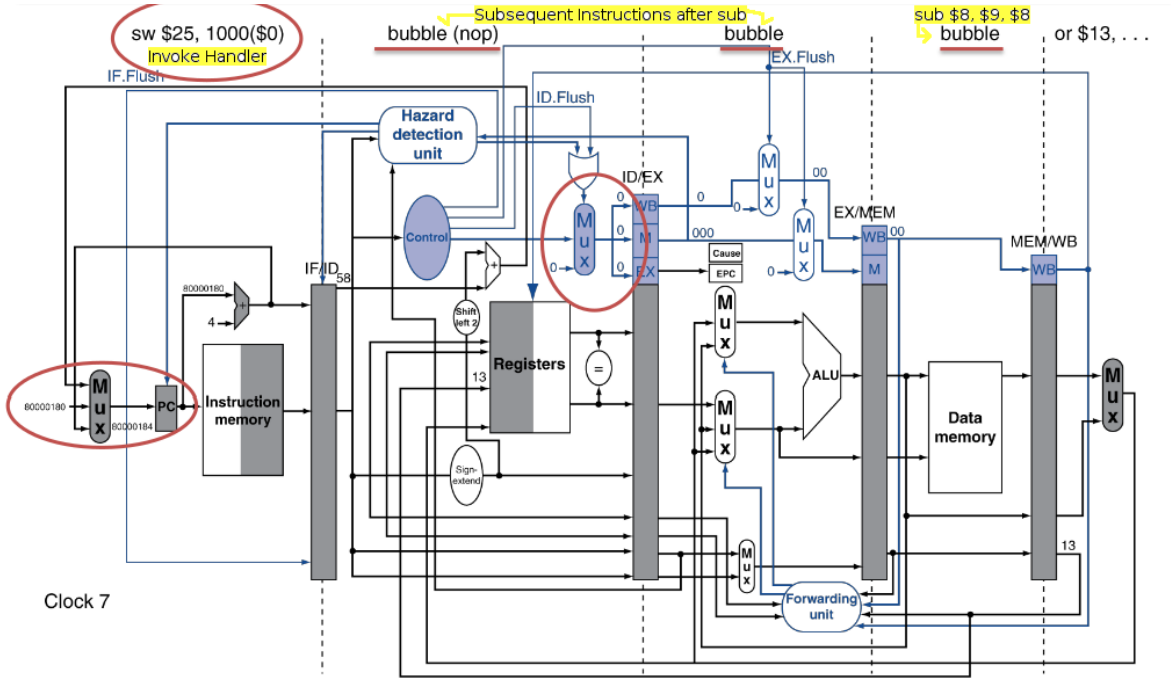
0’ Takt der Exception:
– Execute previous instructions
Instructions in MEM & WB-Flags remain the same, pending on previous instructions
– Prevent $8 from being written
EX.Flush set to 1
Set Flags for MEM & WB to 0, in order to prevent the write completely
– Flush SUB and subsequent instructions – Set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

1' Takt der Exception:
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 = 0,5875 + 0,49 = 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

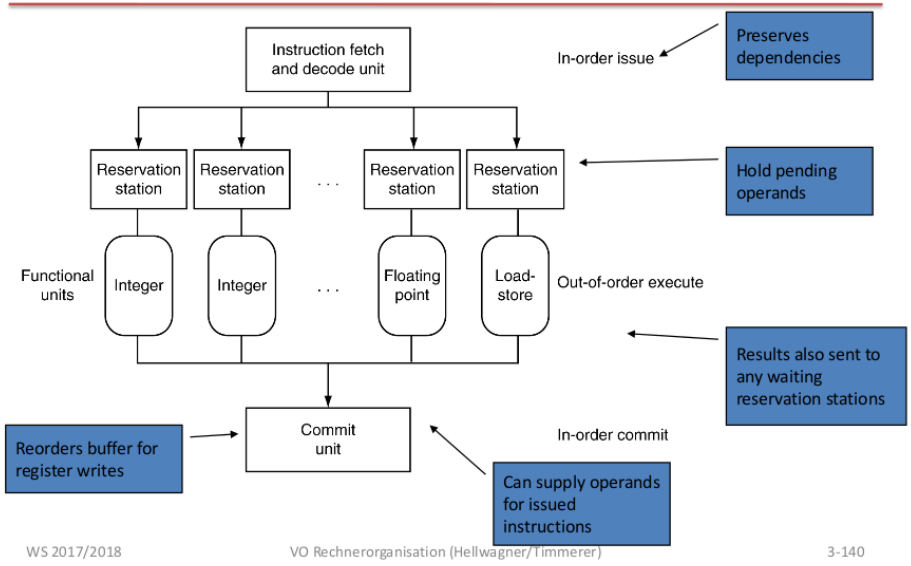
→ 2 Concurrent instructions: 1 Floating Point, 1 Anything else

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

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:	5	FP:	8
3		l.d	$f8, 0($t1)	INT:	2	INT:	3	INT:	5
4		mul.d	$f10, $f6, $f8	FP:	2	FP:	8	FP:	11
5		add.d	$f12, $f10, $f2	FP:	3	FP:	11	FP:	14
6		s.d	$f12, 0($t2)	INT:	3		15		16
7		addi	$t2, $t2, -8	INT:	4	INT:	5	INT:	6
8		addi	$t1, $t1, -8	INT:	5	INT:	6	INT:	7
9		addi	$t0, $t0, -8	INT:	6	INT:	7	INT:	8
10		bne	$t0, $t4, loop	INT:	7		8		9

=> 7 Zyklen per Result – Direkt ablesbar in der Issue.
=> Dynamisch: Zur Laufzeit von der Hardware ↔ Code wird von dieser Optimiert
– Man weiß nicht wann/wo/was fertig wird.

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:

Mem-Ref	Mem-Ref	FP1	FP2	INT op/branch
l.d $f4.0 0($t0)	l.d $f4.1 8($t0)			
l.d $f4.2 16($t0)	l.d $f4.3 24($t0)			
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	
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	
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
l.d $f8.4, 32($t1)	l.d $f8.5, 40($t1)	* Stall due F6		Addi $t1, -48
		mul.d $f10.0 $f6.0 $f8.0	mul.d $f10.1 $f6.1 $f8.1	Addi $t0, -48
		mul.d $f10.2 $f6.2 $f8.2	mul.d $f10.3 $f6.3 $f8.3	
		mul.d $f10.4 $f6.4 $f8.4	mul.d $f10.5 $f6.5 $f8.5	
		* Stall due F10		
		add.d $f12.0 $f10.0 $f2.0	add.d $f12.1 $f10.1 $f2.1	
		add.d $f12.2 $f10.2 $f2.2	add.d $f12.3 $f10.3 $f2.3	
		add.d $f12.4 $f10.4 $f2.4	add.d $f12.5 $f10.5 $f2.5	
s.d $f12.0, -40($t2)	s.d $f12.1, -32($t2)			
s.d $f12.2, -24($t2)	s.d $f12.3, -16($t2)			Bne $t0, $t4
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>

<u>Rechnerorganisation – Übungsblatt 8</u>
8.1 Caches
Ein C-Programm definiert ein Array int a[5000] und erzeugt nacheinander folgende Schreibzugriffe auf die Array-Elemente: a[15], a[4097], a[0], a[2048].

Nehmen Sie für die folgenden Aufgaben an, dass das Array im Speicher lückenlos ab Adresse 0 abgelegt ist. Eine Speicheradresse und der int Datentyp seien jeweils 32 Bit breit. Der Cache enthalte zu Beginn der Programmausführung nur ungültige Blöcke.

Gegeben sei ein **direkt abbildender write-back** Cache der Größe 16 KiB mit 128 Bytes Blockgröße. Bestimmen Sie für jeden Zugriff der gegebenen Folge: Adresse, Cache Tag, Cache Index, Byte Select (jeweils als Dezimalwerte), in den Speicher geschriebene Array-Elemente (falls zutreffend), in den Cache geladene Array-Elemente (falls zutreffend) und den Wert des Dirty-Bits. Kennzeichnen Sie jeden dieser Zugriffe als Hit, Compulsory Miss bzw. Conflict Miss.

Element	Addresse	Tag	Index	BS	Hit/Miss	Geschrieben	Geladen	Dirty
a[15]	60 = 00000000**0**0111100	0x00000	0	0111100 = 60	Compulsory Miss	none	A[0] ..a[31]	0
a[4097]	16388 = 01**0**00000000000100	0x00001	0	0000100 = 4	Conflict Miss	A[0]..a[31]	a[4097]… a[4128]	1
a[0]	0 = 00**0**00000000000000	0x00000	0	0000000 = 0	Conflict Miss	a[4097]… a[4128]	A[0] ..a[31]	1
a[2048]	8192 = 00**1**00000000000000	0x00000	127	0000000 = 0	Compulsory Miss	A[0] ..a[31]	A[2048] ..a[2081]	1

Kib = KibiBIT
KiB = KibiBYTE

* Blockzeilen => Cache :: 16384 / 128 = 128 Cache-Blöcke
  ↔ 32 Bit Speicheradressen => 4 Byte Speicherzellenbesetzung per Element ↔ 128/4 = 32 Elemente per Index
* Addresse
  => Index of array * 4 Bit ↔ a[**15**] * 4 Bit = 15*4 = 60
  … Transmutation to Binary necessary

* Byte Select = ld(Blockgröße) = ld(128) = 7
  => 2^M … M is the parameter to get the block size.
  … For 128' Sizes, it's 2^7 → M = 7 ↔ The last 7 Bits indicate Byte Select
* Index = ld(Blockzeilen) = ld(128) = 7
* Tag = Restbit = Speicheradressgröße – Byte Select – Index = 32 – 7 – 7 = 18 Bit
  => The uppermost (X-N) bits define the Cache Tag
* Hit/Miss
  Hit: Data appears in some block in the upper level
  ### Wenn TAG und INDEX gleich sind ↔ HIT
  Miss: Data needs to be retrieved from a block in the lower level.
  => Compuslory Miss ↔ First Data writes invokes compulsory misses
  => Conflict Miss ↔ Dataoverwrites invoke conflict misses.
* Geschrieben
  Data that is written from inside the cache to the lower level Memory
* Geladen
  Data that is written into cache at the time of execution
* Dirty
  IF Set: Write memory back to hardware later
  => Write-Back: Written only to cache. If space is required, that data is saved to lower layer
  => Write-Through: Written to cache & Sublayer

8.2 Caches
Wie 8.1 für einen **2-fach satzassoziativen write-back** Cache gleicher Größe (16 KiB) und Blockgröße (128 Bytes) mit LRU (least recently used) Ersetzungsstrategie. Geben Sie zudem für jeden Zugriff auch den verwendeten Satz (0 oder 1) an.

Element	Addresse	Tag	Index/ Satz	BS	Hit/Miss	Geschrieben	Geladen	Dirty
a[15]	60 = 00000000**0**0111100	0x00000	0/0	0111100 = 60	Compulsory Miss	invalid	A[12] … [23]	0
a[4097]	16388 = 01**0**00000000000100	0x00010	0/1	0000100 = 4	Compulsory Miss	invalid	A[4091] …a[4103]	0
a[0]	0 = 00**0**00000000000000	0x00000	0/0	0000000 = 0	Conflict Miss	A[12] … [23]	A[0] … a[11]	1
a[2048]	8192 = 00**1**00000000000000	0x00001	0/1	0000000 = 0	Conflict Miss	A[4091] …a[4103]	A[2039] … a[2051]	1

* Blockzeilen = Cache Größe/Blockgröße*n = 16384 / 128*2 = 64 Blöcke zu 2 Sätzen
  ↔ 32 Bit Speicheradressen => 4 Byte Speicherzellenbesetzung per Element ↔ 64/4 = 12 Elemente per Index
* Byte Select: ld(BlockGröße) = ld(128) = 7 Bit
* Cache Index = ld(Blockzeilen) = ld(64) = 6 Bit
  → Satzbestimmung: Least recently used will be overwritten.
* Adresstag = Speicheradresse – Byte Select – Cache Index = 32-7-6 = 19 Bit

8.3 Caches
Wie 8.1 für einen **vollassoziativen write-back** (fully-associative) Cache gleicher Größe und Blockgröße mit LRU (least recently used) Ersetzungsstrategie

Element	Addresse	Tag	Index bzw. Block	BS	Hit/Miss	Geschrieben	Geladen	Dirty
a[15]	60 = 00000000000111100	0x00000	0	0111100 = 60	Compulsory Miss	none	A[0] ..a[31]	0
a[4097]	16388 = 01000000000000100	0x000F0	1	0000100 = 4	Compulsory Miss	none	a[4097]… a[4128]	0
a[0]	0 = 00000000000000000	0x00000	2	0000000 = 0	Compulsory Miss	none	A[0] ..a[31]	0
a[2048]	8192 = 00100000000000000	0x000C0	3	0000000 = 0	Compulsory Miss	none	A[2048] ..a[2081]	0

WÄHLE DORT WO FREI ↔ Es gibt keine eindeutige Cache-Addresse mehr.

* Blockzeilen => Cache :: 16384 / 128 = 128 Cache-Blöcke
  ↔ 32 Bit Speicheradressen => 4 Byte Speicherzellenbesetzung per Element ↔ 128/4 = 32 Elemente per Index
* Byte Select: ld(BlockGröße) = ld(128) = 7 Bit
* Cache Index = N/A bei vollassoziativen Caches.
* Adresstag = Speicheradresse – Byte Select – Cache Index = 32-7-0 = 25 Bit

8.4 Speicherhierarchien
Gegeben sei eine Rechnerarchitektur mit drei Speicherebenen laut untenstehender Tabelle, die die durchschnittliche Hitrate und die Verzögerung (Wartezeit der CPU) im Fall eines erfolgreichen Zugriffs auf die jeweilige Speicherebene angibt. Der durchschnittliche CPI-Wert ohne Berücksichtigung der Speicherzugriffe betrage 1,3 (idealer CPI). Nehmen Sie an, dass durchschnittlich 20% aller Instruktionen auf den Speicher zugreifen.

Speicherebene	Hitrate	Verzögerung/Takte
L1	85%	4
L2	75%	12
Hauptspeicher - RAM	100%	236

a) Berechnen Sie den durchschnittlichen CPI-Wert der vorgestellten Architektur unter Berücksichtigung der Speicherzugriffe.

CPI = Durchschnittlicher CPI + Summe(Zugriffswahrscheinlichkeit * Taktverzögerung)
= 1,3 + L1 + L2 + RAM

→ L1: (20%*85%) * 4 = 0,2*0,85*4 = 0,17 * 4 = 0,68
→ L2: [20%* (100%-85%) * 75%] *12 = [0,2*0,15*0,75] *12 = 0,0225 * 12 = 0,27
→ RAM: [20%*(100%-85%) * (100%-75%) * 100%] * 236
= [0,2*0,15*0,25*1] * 236 = 0,0075*236 = 1,77

CPI = 1,3+0,68+0,27+1,77 = 4,02

UE BSP

Alle	HFG: 1	Zyklen: 1,3	CPI: 1,3
L1	0,2*0,85 = 0,17	4	0,68
L2	0,2*0,15*0,75 = 0,0225	12	0,27
RAM	0,2*0,15*0,25*1 = 0,0075	236	1,77
			4,02

… 4,02 = Zyklen MIT allem drum und dran.

b) Wie viel Prozent der Ausführungszeit eines Programms muss der Prozessor durchschnittlich auf Speicherzugriffe warten?
– Durchschnittlicher CPI = 1,3
– Speicherzugriffs CPI = 4,02
=> Delta CPI = 4,02-1,3 = 2,72 … mit 2,72/4,02 = 0,676616915 hat der Prozessor im Schnitt 67% auf Speicherzugriffe warten.

c) Auf welchen Wert muss die Hitrate des L1-Caches erhöht werden, um einen durchschnittlichen CPI-Wert von 3 zu erhalten (bei unveränderten Hitraten der anderen Speicherebenen)?
0,8 * 1,3 + 0,2*(x*4 + (1-x) * (0,75 * 12 + 0,25 * 236)) = 3

x * 4 + (1 – x) * 68 = (3 – 0,8 * 1,3) / 0,2
x = -(9.8 – 68) / 64
x ~~ 0.91… 92,17%

8.5 Speicherhierarchien
Gegeben sei eine Rechnerarchitektur mit zwei Speicherebenen laut untenstehender Tabelle, die die durchschnittliche Hitrate und die Verzögerung (Wartezeit der CPU) im Fall eines erfolgreichen Zugriffs auf die jeweilige Speicherebene angibt. Der durchschnittliche CPI-Wert mit Berücksichtigung der Speicherzugriffe betrage 7,5. Nehmen Sie an, dass durchschnittlich 30% aller Instruktionen auf den Speicher zugreifen.

Speicherebene	Hitrate	Verzögerung/Takte
L1 Cache	80%	3
Hauptspeicher	100%	90

a) Berechnen Sie den durchschnittlichen CPI-Wert der vorgestellten Architektur ohne Berücksichtigung der Speicherzugriffe (idealer CPI-Wert). Wie viel Prozent seiner Ausführungszeit muss der Prozessor auf Speicherzugriffe warten?

CPI-Total = CPI + L1 + Hauptspeicher CPI:=x
7,5 = x + L1 + Hauptspeicher
→ L1: 30%*80%*3 = 0,3*0,8*3 = 0,72
→ HMem: 30%*100%*(1-80%)*90 = 0,3*(0,2)*90 = 5,4

=> 7,5 = x + 0,72 + 5,4
→ x = 7,5 – 0,72 – 5,4 = 1,38 = CPI

b) Durch Modifikation des Befehlssatzes ändert sich der durchschnittliche ideale CPI-Wert der Rechnerarchitektur auf 1,1. Nun soll ein L2 Cache eingebaut werden, der eine durchschnittliche Hitrate von 90% erwarten lässt. Wie groß darf die Verzögerung (in Takten) dieses L2-Caches maximal sein, damit der CPI-Wert der erweiterten Architektur den Wert von 3,5 nicht überschreitet?

CPIneu = 1,1
CPItotal = 3,5
→30% sind Speicherzugriffe

Zugriffe	p	Takte	CPI
Alle	1	1,1	1,1
L1	0,3 * 0,8	3	0,72
L2	0,3*(1-0,8)*0,9 = Anzahl Zugriffe von L2	X → 21	0,054*x → 1,134
RAM	0,3*(1-0,8)*(1-0,9)	90	0,54

SUMME: 2,36+0,054*x
… soll <= 3,5

x→ → 3,5 >= 2,36 + 0,054x
x <= 21,1

Übungsblatt 9
9.1 Caches
Gegeben sei folgendes Fragment eines C-Programms, welches auf drei Arrays operiert.

```
#define N 2048
int arrA[N];
int arrB[N];
int arrX[N];

int i, s;

for (i=0; i!=N; i++) {
arrA[i]=i;
arrB[i]=N-i;
}

s = 0;

for (i=N-1; i>=0; i--) {
s = s + arrA[i] - arrB[i];
```

arrX[i] = s;
}

Nehmen Sie an, dass die Arrays beginnend mit arrA lückenlos ab Adresse 0x10010000 im Speicher abgelegt sind. Der Datentyp int belege 32 Bit. Die beiden Variablen i und s werden vom Compiler in Registern abgelegt, d.h. deren Verwendung verursacht keinen Speicherzugriff. Das Fragment wird auf einem System mit einem 4 KiB großen 2-fach satz-assoziativen write-back Datencache mit LRU-Ersetzungsstrategie ausgeführt. Die Größe eines Cache-Blocks beträgt 32 Byte. Gehen Sie davon aus, dass der Cache zu Beginn nur ungültige Einträge enthält.

a) Bestimmen Sie die Hitrate des Caches für die Ausführung des gesamten Fragments, und beurteilen Sie die erzielte Performance.

→ Cachegröße = 4096 Byte
→ Blockgröße = 32 Byte
→ Data = 32 Bit
→ N = 2 … 2-fach Satzassoziativ, LRU

=> Anzahl an Blöcken = Cachegröße / Blockgröße * N
= 4096 / 32 * 2 = 4096 / 64 = 64 Indexierungen á 0 oder 1
=> Speichergrößen: 1 Byte = 8 Bit : 32 Byte = 256 Bit
=> Blockinhalt: 8 Wörter
=> Byteselect = ld(Blockgröße) = ld(32) = 5
=> Cache Index = ld(Blockzeilen) = ld(64) = 6
=> Tag = Blockgröße – Byteselect – Index = 32 – 5 – 6 = 21

=> 1 Cache Eintrag = 32 Bit an Data = 4 Byte ↔ 32 Byte Blockgröße => 8 Einträge per Block
=> Offset zwischen Arrays = Arraygröße * Data = 2048 * 32 = 65536 = x10000 Offset zwischen Arrays

2048 Wörter pro Array: 2048 * 4 = 8192 = Offset von 0x2000, NICHT 0x10000
For 1:

Element	Addresse	Tag	Index/Satz	BS	Hit/Miss	Geschrieben	Geladen	Dirty
arrA[0]	0x10010000	0x10010	0/0	0	Comp Miss	Invalids	ArrA[0] … arrA[7]	1
arrA[1]	0x10010004			x4	Hit	ArrA[1]		
arrA[7]	0x10010020			x20	Hit	ArrA[7]		
ArrA[8]	0x10010024			x24	Comp Miss (Conflict?)			
ArrA[9]	0x10010028			x28				
arrB[0] – Analog	0x10020000	0x10020	0/0	0	Comp Miss	invalid	ArrB[0] … arrB[7]	1

Ad b.) Arrayinhalt von [N-i] irrelevant
→ 0-7 => 1 Miss :: Element 0 = Miss ↔ Rest ist Hit
→ 8–15 => 1 Miss
….
→ 2041-2048 => 1 Miss
=> #Misses per Array = #Schreibzugriffe per Array = 2048 Datas / 8 Einträg = 256 Misses per Array ↔ Bei 2048 Zugriffen 256 erzeugte Misses
→ 1792
=> Totale #Misses = 256*2 = 512 Misses insgesamt.
=> Total #Hits = 1792*2 = 3584 Hits insgesamt

Load-Schema bleibt immer gleich

A[0]...A[7] || A[512]...	B[0]...b[7]	0
…	...	1….62
...A[511] || … || … A[2047]	...B[511]	63

→ Miss auf ersten Save, Hits auf Folgeloads

For 2:
i = 2047 … 0
s += arrA[i] – arrb[i];
arrX[i] = s;

Element	Addresse	Tag	Index/Satz	BS	Hit/Miss	Geschrieben	Geladen	Dirty
ArrA [2047]	0x10021FFC	0x10021	111111	11100	Hit	X	A2040...2047	0
ArrB [2047]	0x10011FFC	0x10011	111111	11100	Hit	X	B2040...2047	0
ArrX [2047]	0x10031FFC	0x10031	111111	11100	Comp Miss	X	X2040...2047	1
A..B..X					H..H..M			0..0..1
A..B..X					M..M..M			1..1..1
ArrA [2039]					Conf Miss	2040...2047	A2031...2039	0
ArrB [2039]					Conf Miss	2040...2047	B2031...2039	0
ArrX [2039]					Conf Miss	2040...2047	X2031...2039	1

→ Wir verändern A/B nicht ↔ Dirty 0

Zugriff auf A und B sind bereits im Cache → Die ersten 64 sind schon drin → Hits
→ Die X ersetzt A, A ersetzt dann B → Folgeschema nach LRU
=> 2 Hits, 1 Miss bei 1. Aufruf der Schleife
=> 3 Misses * 7 Aufrufe d. Schleife = 21, für die ersten 64 Elemente → 128 Hits || 1408 Mis
#
Restliche nach den 64: Immer Miss.
→ 8*3 Misses * 64 * 3 = 4608 Misses
Hits Gesamt: 3584 + 128 = 3712
Miss Gesamt: 512+1408+4608 = 6528 Misses
á 10240 Aufrufe.
→ 4712 / 10240 = 0.3625 → 36,25% Hitrate

Satz 0-1 :: A-B ↔ Mit X wird jedesmal ein Conflict Miss erzeugt durch identen ByteSelect und Index.
=> Durchlauf des Cache – LRU :: Swappe abwechselnd

HIT:A at 0 – HIT:B at 1 – MISS:X => Swapp into 0 # I = 1
MISS:A => Swapp into 1 – MISS:B => Swapp into 0 – MISS:X: Swap into 1 # I = 2
… repeat until fin.

→ 2048 Elemente per Array :: Schleife 1 und 2 ↔ 5 Zugriffe per Iteration
2048 * 5 = 10240 Zugriffe

→ 2 Hits und 1 Miss per 1. Aufruf
→ 3 Misses für Folgeaufrufe per Speicherung
→ 64 Hits zum 2fachen Satz <=> 128 Hits

Hitrate = Hits / Zugriffe
= 3712 / 10240
= 0,3625 → 36.25% Hitrate

b) Im Moodle-Kurs finden Sie ein entsprechendes MIPS-Assembler-Programm des obigen Fragmentes. Verwenden Sie den MARS Data Cache Simulator, um Ihr Resultat aus a) zu verifizieren.

9.2 Caches
Gegeben seien das Code-Fragment und der Datencache aus Aufgabe 9.1.

a) Kann ein Compiler mit Hilfe von Array-Padding eine höhere Hitrate ermöglichen? Wenn ja, wie hoch wäre näherungsweise die verbesserte Hitrate für das gegebene Fragment?
→ Arraypadding via Minimalem Datensatz: 8 Byte oder 32 Bit

A[2048]
B[2048]
X[2056] ↔ Padding um 32 Bit

Schleife 1:

Addresse	Tag	Index/Satz	BS	Hit/Miss	Geschrieben	Geladen	Dirty
0x10010000	0x10010	0/0	0	Comp Miss	Invalids	ArrA[0] … arrA[7]	1
0x10010004			x4	Hit	ArrA[1]		
0x10010020			x20	Hit	ArrA[7]		
0x10010024			x24	Comp Miss (Conflict?)			
0x10010028			x28				
0x10020000	0x10020	0/0	0	Comp Miss	invalid	ArrB[0] … arrB[7]	1

Diese bleibt gleich.

Schleife 2:

Addresse	Tag	Index/Satz	BS	Hit/Miss	Geschrieben	Geladen	Dirty
0x10021FFC	0x10021	111111	11100	Hit	X	A2040...2047	0
0x10011FFC	0x10011	111111	11100	Hit	X	B2040...2047	0
0x10031FFC	0x10031	111111	11100	Comp Miss	X	X2040...2047	1
				H..H..M			0..0..1
				M..M..M			1..1..1
				Conf Miss	2040...2047	A2031...2039	0
				Conf Miss	2040...2047	B2031...2039	0
				Conf Miss	2040...2047	X2031...2039	1

→ Nun ist der Index jedesmal so verschoben, dass 1 Array 1 Index weiter gespeichert wird. Dementsprechend ist die Hitrate stark erhöht.

→ á 2048 Elemente auf 3 Arrays werden erzeugt:
→ 1. Zugriff: 1 Miss, 2 Hits
→ Folgezugriffe 2...8: 3 Hits
=> 1 Miss und 23 Hits á Arrayspeicherung = 63 Misses

X: Erzeugt 7 Hits und 1 Comp Miss.
→1792 Hits, 256 Misses
AB: Erzeugen auf die ersten 0..63 Blöcke stets Hits
=> 504 Hits
Für darauffolgende Iterationsspeicherungen wird 1 Miss und 7 Hits erzeugt
= [#Durchläufe-Misses] * 1-Missrate
→2048-504 = 1544 * 0.875 = 1351 Hits

Es ergeben sich damit für die zweite Schleife 5502 Hits und 642 Misses.

→ Hitrate = Hits / Zugriffe
= 9086 / 10240
= 0,89. = 89% Hitrate

Insgesamt:
Zugriffe: 4096 + 6144 = 10240
Hits: 3584 + 5502 = 9086
Misses: 512+642 = 1154

B wird vergrößert

A[0]...a[7]	B[0]...b[7]	n

X[0]...x[7] || a[8]..a[15]	B[8]...b[15]	n+1

Bei 63 startet X wieder bei 0 – Im Ersten Fall sind die Werte bereits so drinnen. Erst NACH 63 Iterationen werden die Werte geladen.

b) Wie würde sich die Hitrate des Caches aus Aufgabe 9.1 (ohne Array-Padding) verändern, wenn der Datencache 4-fach satz-assoziativ organisiert wäre? Bestimmen Sie die Hitrate näherungsweise.

1. Schleife bleibt gleich
→ 3584 Hits, 512 Misses

2. Schleife besitzen für die ersten 32 Blöcke immer 100 Hits
→ 32*8 = 256 Hits
Anschließend für jeden Block zunächst 1 Miss und anschließend 7 Hits
→ 2048-256 = 1792 * 0875 = 1792 Hits und 256 Misses

=> Daraus folgt:
256+1568 * 2 + 1792 = 5440 Hits
704 Misses

4-Fach:

A	B	X	-	0
A	B	X	-	1
A	B	X		31

Suche via TAG
A[0] = Tag 2
B[0] = Tag 3
X[0] = Tag 5
→ Index 0 :: Zeile
→ Da steht A und B drin = 4 Vergeiche @Tag ↔ 5 = 2 ? Nein → 2 ist nicht X.
↔ 5 = 3 ? Nein → 3 ist nicht X.
↔ 5 = 5 ? Ja, an X ist X.

Cache:

Tag	Index	Offset
	0-31	0-31

→ 2048 – 256 = 1792 ↔ 7/8 Hit% = 1568 Hits für 1 Array :: A und B.
→ X:: 1 Zugriff Miss, Rest Hits = 1792
→ [(256 + 1568) * 2 + 1792 + 3584] / 10240 = 0,88

9.3 Caches
Gegeben sei folgender Pseudocode:

```
int array[100, 100000];
for each element array[i][j] {
array[i][j] = array[i][j] * 2;
}
```

Schreiben Sie zwei Java-Programme, die diesen Pseudocode implementieren: eines soll das Array zeilenweise durchlaufen, das andere spaltenweise. Die benötigte Ausführungszeit für den Schleifendurchlauf soll ausgegeben werden. Zum Messen der Ausführungszeit kann java.lang.System.currentTimeMillis() verwendet werden. Führen Sie beide Programme mehrmals hintereinander aus und vergleichen Sie die minimalen Ausführungszeiten. Was sagt Ihnen die unterschiedliche Cache-Performance über die Anordnung von Java-Arrays im Speicher?
Hinweis: Verringern Sie die Zeilenanzahl (1. Dimension) des Arrays, falls Ihre virtuelle Java-Maschine (JVM) nicht genügend Speicher zur Verfügung hat. Oder vergrößern Sie alternativ die maximale Speichergröße der JVM (z.B. mit java –Xmx128m).

Übungsblatt 10
10.1 Virtuelles Speichersystem
Nehmen Sie ein virtuelles Speichersystem mit folgenden Eigenschaften an:
• Wortgröße: 4 Bytes

• Seitengröße: 4 KiB
• Virtuelle Adresse: 32 Bits
• Physische Adresse: 32 Bits
• Zweistufige Seitenumsetzung (die Seitentabelle auf der 1. Stufe umfasst genau eine Seite).
• Jeder Seitentabelleneintrag (page table entry, PTE) muss an einer Wortgrenze beginnen (word-aligned sein).
• 4 Bits Zusatzinformationen (valid, dirty, reference, protection) pro Seitentabelleneintrag.
• Es gibt keinen TLB (translation look-aside buffer).

a) Geben Sie die Größe des virtuellen bzw. physischen Adressraums in GiB an.

Der Adressraum ist abhängig von den jeweiligen Adresslängen.
Virtueller Adressraum 2^32 = 4.294.967.296 ~~ 4GiB
Physischer Adressraum 2^32 = 4.294.967.296 ~~ 4GiB

b) Bestimmen Sie die Größe eines Seitentabelleneintrags (page table entry, PTE) und
die Anzahl der Einträge pro Seitentabelle. Leiten Sie daraus ab, wie die Bits der virtuellen Adresse 0x12345678 interpretiert werden (Indizes, Page Offset).
Virtual Addresses point towards physical addresses/page frames or disk-Address.

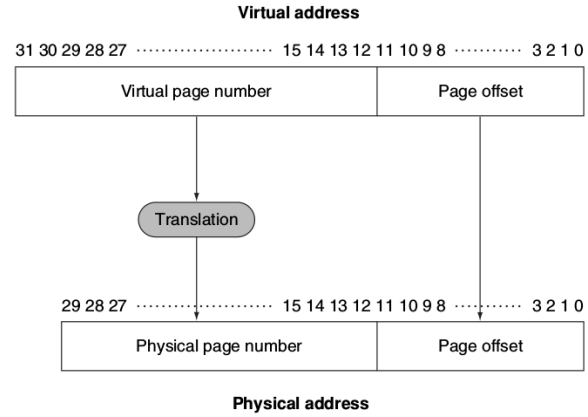


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is 2^{12} = 4 KiB. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

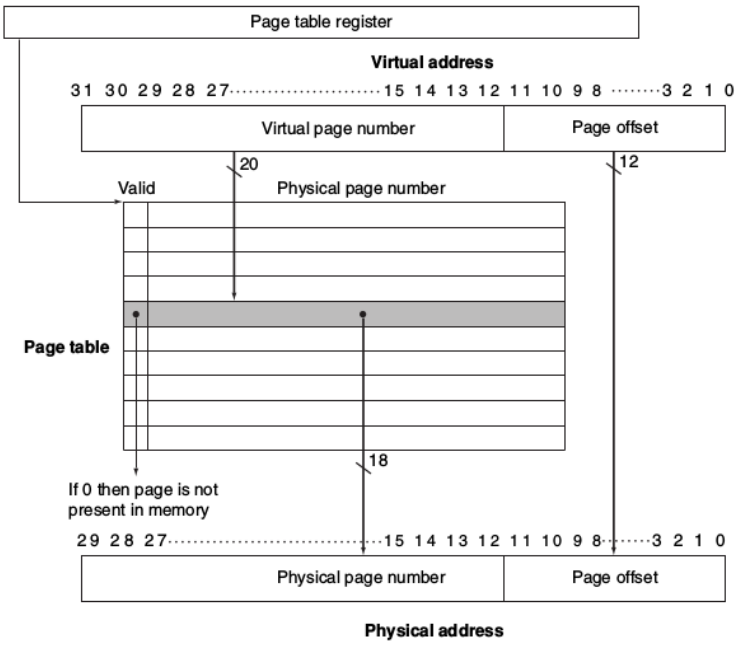


FIGURE 5.27 The page table is indexed with the virtual page number to obtain the corresponding portion of the physical address. We assume a 32-bit address. The page table pointer gives the starting address of the page table. In this figure, the page size is 2^{12} bytes, or 4 KiB. The virtual address space is 2^{32} bytes, or 4 GiB, and the physical address space is 2^{30} bytes, which allows main memory of up to 1 GiB. The number of entries in the page table is 2^{20}, or 1 million entries. The valid bit for each entry indicates whether the mapping is legal. If it is off, then the page is not present in memory. Although the page table entry shown here need only be 19 bits wide, it would typically be rounded up to 32 bits for ease of indexing. The extra bits would be used to store additional information that needs to be kept on a per-page basis, such as protection.

=> #PTE = 2^#Bits of Virtual Adress(32) / 2^#Pages (4KiB)
= 2^32 / 2^12
= 2^20

=> Seitengröße berrechnet sich aus mehreren Parametern. (Abb.: 5.27)
1. Visual Page Number = 32 – Page Offset = 20 Bit
2. Page Offset := ld(Seitengröße) → Seitengröße = 4KiB = 2^12
→ 12 Bit.
3. Zusatzinformationen! 4 Bit.
→ VPN + 4 Bit = 24 Bit
4. "Jeder Seitentabelleneintrag (page table entry, PTE) muss an einer Wortgrenze beginnen, word-aligned sein."
→ Es braucht ein Padding um auf 32 Bit zu kommen.
24 Bit + 8 Bit Padding = 32 Bit.

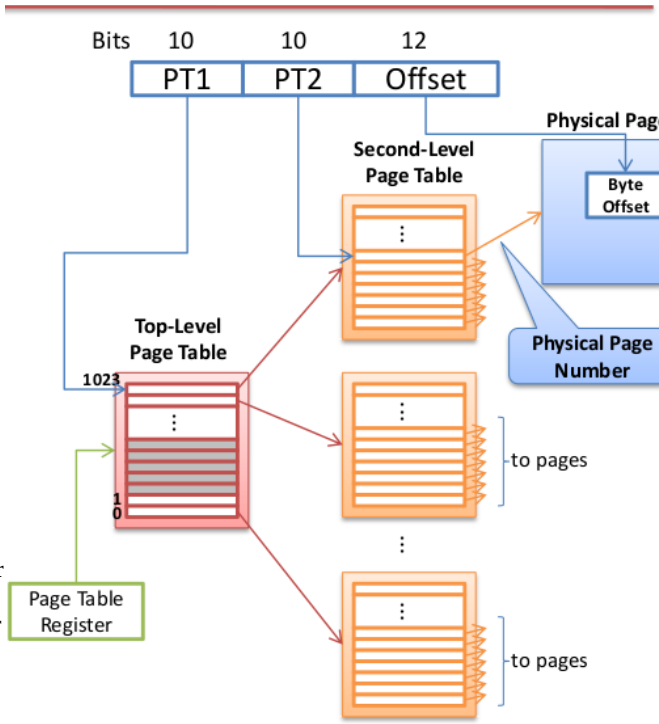
→ Ein Page Table Entry ist damit 32 Bit oder 4 Byte breit.

=> Anhand der Seitengröße von 4 KiB passen:
4KiB / 4B = 1024 Einträge
in das erste Level: PT1

=> Die Indexierung des PT1 errechnet sich aus ld(#Einträge) = ld(1024) ~ 2^10 = 10 Bit.

Wir haben eine 2-stufige Seitenumsetzung:

Der Second-Level Page Table, PT2 besitzt die gleichen Parameter wie PT1.
– "PTE stores the physical Page number" if page is present in memory.
– 32-Bit Virtual address partiitoned into
10-bit PT1 → Index to top-level page table ↔ Contains page number for second-level
10-bit PT2 → Index to second-level page table ↔ Physical page number
12-bit Offset → Byte offset within physical page



für die Adressberechnung von **der virtuellen Adresse 0x12345678** betrachtet man nun die Bitstellen.

1	2	3	4	5	6	7	8
0001	0010	0011	0100	0101	0110	0111	1000
0001001000			1101000101			011001111000	
PT1			PT2			OFFSET	

Die Addressierung in PT1 ist folglich:

PT1 = 00 0100 1000 = 0x048
PT2 = 11 0100 0101 = 0x345
Offset = 0110 0111 1000 = 0x678

c) Stellen Sie die Umsetzung einer virtuellen Adresse in eine physische Adresse grafisch dar.

VPN → PPN → Memory

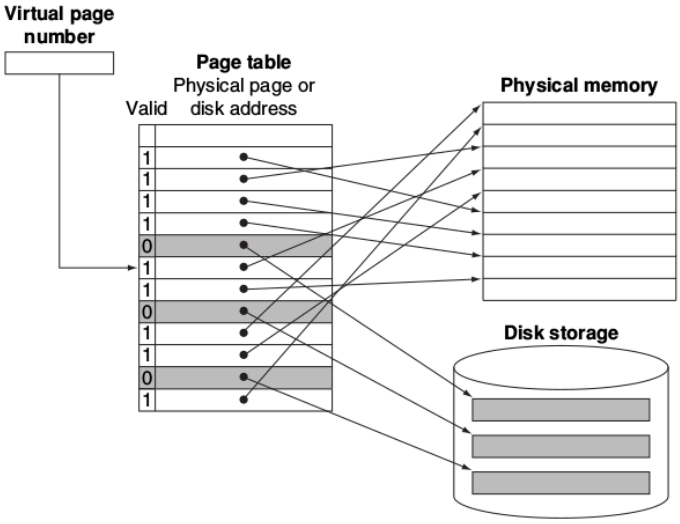


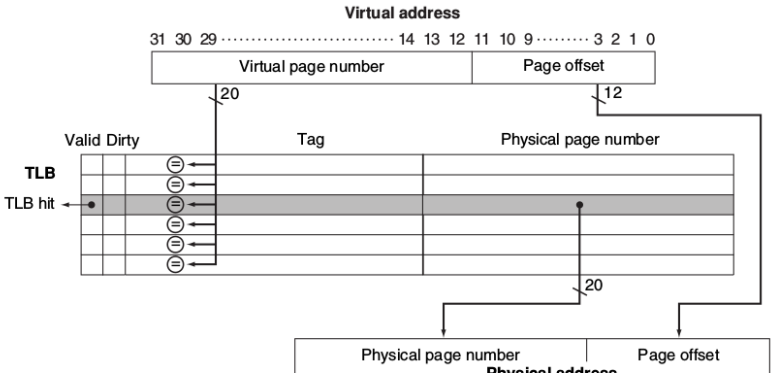
FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

10.2 Translation Look-aside Buffer (TLB)
Das virtuelle Speichersystem aus Ü 10.1 soll durch einen 4-fach satz-assoziativen (4-way set-
associative) TLB mit insgesamt 128 Einträgen erweitert werden.

a) Stellen Sie die Umsetzung der virtuellen in physische Adressen mit Hilfe des TLB
grafisch dar und geben Sie die Breite aller Felder und Signale an.
=> 128 Einträge zu je 4' Sätzen → 32 Einträge mit je 4 Sätze

→ Visual Page Number = 32 – Page Offset = 20 Bit
Accounts for each virtual and physical page
→ Page Offset := ld(Seitengröße) → Seitengrö0e) 4KiB = 2^12 → Offset = 12 Bit.
→ Zusatzinformation: 4 Bit

b) Welche Bits einer virtuellen Adresse werden vom TLB als Cache Tag bzw. als Cache
Index verwendet?



Indexierung im TLB: 32 Einträge ↔ 0...31 Index.
→ ld(32) = 2^5 = 5, 5 Bit des Tags werden für Indexierung benötigt.
→ der Cache Tag entspricht den 20 Bit der Page Number – Indexierungsbits
= 20 – 5 = 15 Bits.

c) Aus welchen Feldern besteht ein TLB-Eintrag? Bestimmen Sie die Größe des TLB in Bytes.
TLB besteht aus

Zusatzinformation	Cache-Tag	Physische Seitennr	
4 Bit	15 Bit	20 Bit	… 39 Bit

Mit einem TLB von 32 Einträgen zu je 4 Sätzen, ergeben sich
39 Bit * 128 = 4992 Bit = 624 Byte

Wir greifen nicht mehr auf Speicher zu um Virtuelle/Physische zu übersetzen.

10.3 Virtuelles Speichersystem
Gegeben sei eine byte-adressierbare Maschine mit einem virtuellen Speichersystem mit 42-

Bit virtuellem Adressraum, einer Seitengröße von 1 KiB und 34-Bit physischem Adressbereich. Jede Seitentabelle (Page Table) soll gerade eine Speicherseite belegen. Pro Eintrag in der Seitentabelle (Page Table Entry) werden außer der physischen Seitennummer (Page Frame Number) 7 Bit Zusatzinformationen benötigt. Ein Seitentabelleneintrag belege eine ganze Anzahl von Worten (1 Wort = 4 Bytes).

– 42 Bit Virtueller Adressraum
– 34 Bit physischer Adressraum
– Seitengröße 1 KiB
– 7 Bit Zusatzinformation
– Jeder Seiteneintrage belege eine ganze Anzahl von Worten
– Jede Page Table per Speicherseite

a) Bestimmen Sie die Größe des physischen bzw. virtuellen Speicherbereichs sowie die Größe eines Seitentabelleneintrags.
Der Adressraum ist abhängig von den jeweiligen Adresslängen.
Virtueller Adressraum 2^42 = 549.755.813.888 = 550 Gb
Physischer Adressraum 2^34 = 17.179.869.184 = 17 Gb

###
2^42 Virtuelle
2^42B =2^32 KiB = 2^22 MiB = 2^12 GiB = 4 TiB
2^34 Physische
2^34B = 2^24 KiB = … = 4 GiB

b) Bestimmen Sie nachvollziehbar die minimale Anzahl von Stufen (Levels), die in einer mehrstufigen Seitentabellenhierarchie für die Adressumsetzung benötigt werden.
Generell gilt:

Seitengröße 2^10 – Offset = 10 Bit
1.→ 34-10 = 24 Bits für VPN
2.→ 7 Bit Zusatzinformation
3. => 1 Bit Padding

Minimale Anzahl Levels
Größe Seitentabelle / Größe PTE = 1 KiB / 32 Bits
= 2^10 B/2^2 B
= 2^8 B

… Bei 2^32 Seiten im Virtuellen Adressraum
=> Gesamtanzahl Seiten = Größe Virtueller Adressraum / Größe Seitentabelle
= 2^42 B / 2^10
= 2^32 Seiten.

Das Problem ist lösbar mit weiteren Stufen:
Gesamtanzahl Seiten <= (Einträge per Seite)^n

N = ceil(log(Gesamtanzahl Seiten) / log(Einträge pro Seite))
= ceil(log(2^32)) / log(2^8))
= ceil(32/8)
= 4 Pages

VA: 40 Bit
→ 30 Bit: Virtuelle Pages => 2^30 PTE
→ 10 Bit Offset, at

→ Berechnung Einträge
1 PTE = 4 Byte
Page Size = 1 KiB
#Page Table Entries / Page Table Size = 2048 / 4
= 512 Einträge

Beginne ganz Links auf der Adresse.
Die ersten 9 Bit zeigen auf Sublevel →
→ PT2 zeigt auf Subsublevel →

9 Bit	9 Bit	9 Bit	3 Bit

→ Direkt errechenbar aus #Einträge in Betracht zur Virtuellen Addresse.
=> 4 Levels, wobei PT4 mit 3 Bit besetzt ist. – Beachte OFFSET von VPN || OFFSET
=> Levels NUR IN VPN

30 #VPN / 9 = 3, … → Aufgerundet = 4

→ Dasselbe für die Physikalische Adresse á PPN || OFFSET

---

c) Auf welche Einträge (Indizes) wird bei Umsetzung der virtuellen Adresse 0x1122334455<<2 (40-Bit-Adresse wird um 2 Binärstellen nach links geschoben bzw. 2 Null-Bits werden angehängt, Ergebnis ist die 42-Bit-Adresse 0x4488CD1154) in eine physische Adresse in den einzelnen Seitentabellen zugegriffen?

Bei 4 Pages:

PT1	PT2	PT3	PT4	Offset
8 Bit	8 Bit	8 Bit	8 Bit	10 Bit
0001 0001	0010 0010	0001 0001	0010 0010	

d) Für einen Prozess muss virtueller Speicher im Ausmaß von 1000 Seiten in physische Adressen übersetzt werden. Wie viel Speicher wird im günstigsten Fall für die Seitentabellen benötigt?

… Eine Seitentabelle = 2^8 = 256 Einträge per Page.

→ Bei 1000/256 Einträgen → 4 Seitentabellen auf Stufe 4.
→ Auf allen weiteren Ebenen wird nur 1 Seite benötigt.
→ Für den Speicherbedarf ergeben sich dadurch 7 KiB.

Klausur: Wie gehabt

Mehrstufige Umsetzung VA → PA

Wir haben eine virtuelle Adresse mit 10 Bit Offset und 30 Bit Virtuelle Page Number.

30 Bit VPN	10 Bit Offset

Wir haben pro Page Table 512 Einträge (1Seite = 1 Page Table)
→ Nehme die 30 Bit VPN.
→ Mit ld(512) ergeben sich 9 Bit per Level

Aufteilung wie folgt:

1st Level – 9 Bit	2nd Level – 9 Bit	3rd Level – 9 Bit	3 Bit – Offset	(OFFSET, NOT ACCOUNTED)