8. *( _____ / 13 Punkte)* Pipelining, Control Hazards

Sie haben ein Programm in Pseudo-Code Notation gegeben. Das Programm enthält bedingte Sprünge und wird auf einem Prozessor mit Pipelining und Sprungvorhersage (*Branch Prediction*) ausgeführt. Die Pipeline besteht aus den vier Stufen *Instruction Fetch (F), Decode (D), Execute (E)* und *Memory & Write Back (M/W)*. Die Strategie zur Sprungvorhersage nimmt beim ersten Mal an, dass nicht gesprungen wird und ansonsten, dass die Entscheidung wie beim letzten Mal ausfällt. *1

Wandert also eine Instruktion mit bedingtem Sprung in die Stufe *D*, so wird zunächst die nächste Instruktion in *F* geladen. In Stufe *D* wird die Sprunginstruktion dann als solche erkannt und es kommt zu einer Sprungvorhersage. Sagt diese keinen Sprung voraus, so wird im folgenden Takt einfach weiter die nächste Instruktion in die Pipeline gefetched (als gäbe es keine Sprunginstruktion).

Wird jedoch ein Sprung voraus gesagt, so wird die Instruktion an der Sprungadresse in *F* geladen und anstatt der zuvor in *F* befindlichen Instruktion wird eine *nop Instruktion (no-operation)* in *D* geladen. *2

Erst wenn die Sprunginstruktion die Stufe *E* verlässt, steht fest ob tatsächlich gesprungen wird. *1 War dann die Sprungvorhersage falsch, so werden die Stufen *D* und *E* mit *nops* überschrieben und die Instruktion an der Sprungadresse wird in *F* geladen.

Visualisieren Sie den Ablauf der Pipelineverarbeitung für das gegebene Programm indem Sie die gegebene Tabelle vervollständigen. Setzen Sie die Darstellung solange fort, bis alle Instruktionen vollständig abgearbeitet wurden. Tragen sie auch *nops* ein. *Data-Hazards* müssen nicht berücksichtigt werden.

Programm:

| | |
|---|---|
| *i0:* | R1 ← -3 |
| *i1:* | R0 ← 2 |
| *i2:* | R1 ← R1+1 |
| *i3:* | R0 ← R0+R0 |
| *i4:* | (R1); If N goto i2 |
| *i5:* | R0 ← R0-1 |
| *i6:* | *nop* |

R1 = -3
R0 = 2
R1 = -3 + 1 = -2
R0 = 2 + 2 = 4
R1 = -2 + 1 = -1
R0 = 4 + 4 = 8
R1 = -1 + 1 = 0
R0 = 8 + 8 = 16
R0 = 16 - 1 = 15
nop

| Zeit ↓ | F | D | E | M/W | |
|---|---|---|---|---|---|
| 1 | *i0* | | | | |
| 2 | *i1* | *i0* | | | |
| 3 | *i2* | *i1* | *i0* | | |
| 4 | *i3* | *i2* | *i1* | *i0* | |
| 5 | *i4* | *i3* | *i2* | *i1* | |
| 6 | i5 | i4 | i3 | i2 | |
| 7 | i6 | i5 | i4 | i3 | *1 |
| 8 | i2 | nop | nop | i4 | *2 |
| 9 | i3 | i2 | nop | nop | |
| 10 | i4 | i3 | i2 | nop | |
| 11 | i5 | i4 | i3 | i2 | |
| 12 | i2 | nop | i4 | i3 | *1 |
| 13 | i3 | i2 | nop | i4 | |
| 14 | i4 | i3 | i2 | nop | |
| 15 | i5 | i4 | i3 | i2 | |
| 16 | i2 | nop | i4 | i3 | |
| 17 | i5 | nop | nop | i4 | * |
| 18 | i6 | i5 | nop | nop | |
| 19 | | i6 | i5 | nop | |
| 20 | | | i6 | i5 | |
| 21 | | | | i6 | |

7. *( _____ / 10 Punkte)* Pipelining, Data Hazards

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. Analog zur Vorlesung gilt, dass in der ersten Takthälfte geschrieben wird und in der zweiten Takthälfte gelesen. Die lesende Instruktion kann also in Stufe D verarbeitet werden, sobald die schreibende Instruktion in Stufe S verarbeitet wird.
Auf dem Prozessor wird folgendes Programm ausgeführt:

```
ADD   R4,  R2,  R1  # R2 und R1 addieren, Resultat in R4
SUB   R3,  R2,  R1  # R1 von R2 subtrahieren, Resultat in R3
MOV   R5,  R3       # R3 nach R5 kopieren
MUL   R3,  R3,  R4  # R3 und R4 multiplizieren, Resultat in R3
ARSH  R3            # Arithmetischer Rightshift von R3
PUSH  R3            # R3 auf den Stack legen
```

Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline am Beginn und am Ende leer ist. Kennzeichnen Sie *gestallte* Instruktionen durch Einklammern.

Gestallte Befehle sind in Klammer

| Zeit ↓ | F | D | E | S |
|---|---|---|---|---|
| 1 | ADD | | | |
| 2 | SUB | ADD | | |
| 3 | MOV | SUB | ADD | |
| 4 | (MUL) | (MOV) | SUB | ADD |
| 5 | MUL | MOV | | SUB |
| 6 | ARSH | MUL | MOV | |
| 7 | (PUSH) | (ARSH) | MUL | MOV |
| 8 | PUSH | ARSH | | MUL |
| 9 | | (PUSH) | ARSH | |
| 10 | | PUSH | | ARSH |
| 11 | | | PUSH | |
| 12 | | | | PUSH |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |

2018W

4. *( _____ / 22 Punkte)* Die Hardware des Micro16-Prozessors wurde erweitert, sodass der Prozessor Pipelining unterstützt. Die Pipeline besteht aus den vier Stufen *Instruction Fetch (F), Decode (D), Execute (E)* und *Memory & Write Back (M/W)*. **Werte die in *M/W* geschrieben werden, stehen im selben Takt in *D* bereits zur Verfügung.**

   Es ist ein Micro16-Programm $P$ gegeben, welches Datenabhängigkeiten enthält. Dadurch kann es zu **Hazards** kommen. Data Hazards werden durch **Stalling** von der Hardware verhindert. Weiters enthält das Programm **bedingte Sprünge**. Ob gesprungen wird oder nicht, steht fest, wenn die Instruktion die Stufe $E$ verlässt. Falls nicht gesprungen wird, dann läuft die Pipeline ungestört weiter. Wird allerdings gesprungen, so wird die Pipeline *geflushed*, wenn die Sprunginstruktion $E$ verlässt. D.h. die Stufe $D$ wird geleert, und in Stufe $F$ wird die Instruktion an der Zieladresse des Sprunges geladen (unabhängig davon, welche Instruktion in $F$ war).

   **Unbedingte Sprünge** werden sofort erkannt. Wenn der unbedingte Sprungbefehl in die Stufe $D$ weiter wandert, wird sofort die korrekte Folgeinstruktion in $F$ geladen.

   Programm $P$:

```
Zeile:
   0:  R2 ← 0
   1:  (R0); if Z goto 6          Wenn R0 0 ist, dann wird das Programm beendet.
   2:      (R0&1); if Z goto 4    Wenn das letzte Bit 0 ist, dann gehe zu Zeile 4
   3:        R2 ← R2+R1           R2 += R1
   4:      R1 ← lsh(R1)           Left-Shift R1
   5:  R0 ← rsh(R0); goto 1
   6:
```

   (a) Welche mathematische Operation führt das obige Programm aus? In welchen Registern stehen die Operanden und in welchem Register steht am Ende das Ergebnis der Operation?

   R2 <- R0 * R1

   (b) Welche zwei Arten von Hazards treten im obigen Programm auf? In welchen Zeilen treten die entsprechenden Hazards auf?

   Es existieren Structural-, Control- und Data-Hazards.

Structural Hazards:

- Zwei Pipeline-Stufen wollen auf dies selbe Ressource zugreifen. Wenn ich eine Von-Neumann Architektur bei meinem Prozessor habe und eine Pipeline Stufe will die nächste Instruktion holen und eine Pipeline-Stufe will auf Daten zugreifen. Bei einer Von-Neumann Architektur habe ich den selben Speicher für Instruktionen und Daten. Hier hätte ich ein Structural Hazard, da beide Pipeline-Stufen wollen auf die diesselbe Ressource zugreifen. Es kann aber nur ein Zugriff pro Takt auf den Speicher stattfinden.

In Zeile 1 und 2 haben wir bedingte Sprünge, da haben wir Control-Hazards.
Zeile 1/5: Tritt ein Read-After-Write Hazard auf.

(c) Zeichnen Sie den Ablauf der Pipelineverarbeitung für $P$, indem Sie angeben, welche Codezeile zu welchem Zeitschritt in welcher Pipelinestufe bearbeitet wird. Setzen Sie die Darstellung solange fort, bis die leere Instruktionen in Zeile 6 gefetched wurde. `R0` ist mit $(\mathbf{101})_2$ initialisiert. Instruktionen, die gestallt werden sind in Klammern zu setzen.

(d) Welche Taktfrequenz (in Hertz) ist notwendig, damit der Prozessor einen theoretischen Durchsatz von 100 MIPS erreicht?

100 * 10^6 Instruktionen pro Sekunde.

=> f = 100 Mhz

Beim theoretischen Durchsatz geh ich davon aus, dass bei jedem Takt eine fertige Instruktion aus der Pipeline kommt.
Pro Takt eine Instruktion.
Daher brauche ich 100 * 10^6 Takte pro Sekunde.

(e) Angenommen, die Stufe mit der längsten Verarbeitungsdauer benötigt 10 ns, um eine Instruktion vollständig verarbeiten zu können, während die Stufe mit der kürzesten Verarbeitungsdauer 2 ns benötigt. Wie groß ist der maximale theoretische Durchsatz der Pipeline in MIPS?

T = 10ns
=> 1/10ns = $1/10*10^{-9}$s = $10^9/10$ Hz = 100 MHz
=> 100 MIPS

| Takt ↓ | F | D | E | M/W | |
|---|---|---|---|---|---|
| 1 | **0** | | | | |
| 2 | **1** | **0** | | | |
| 3 | **2** | **1** | **0** | | |
| 4 | **3** | **2** | **1** | **0** | |
| 5 | **4** | **3** | **2** | **1** | |
| 6 | **5** | **4** | **3** | **2** | |
| 7 | **1** | **5** | **4** | **3** | |
| 8 | (**2**) | (**1**) | **5** | **4** | |
| 9 | **2** | **1** | | **5** | R0 ← 10 |
| 10 | **3** | **2** | **1** | | |
| 11 | 4 | 3 | 2 | 1 | |
| 12 | 4 | | | 2 | |
| 13 | 5 | 4 | | | |
| 14 | 1 | 5 | 4 | | |
| 15 | (2) | (1) | 5 | 4 | |
| 16 | 2 | 1 | | 5 | R0 ← 1 |
| 17 | 3 | 2 | 1 | | |
| 18 | 4 | 3 | 2 | 1 | |
| 19 | 5 | 4 | 3 | 2 | |
| 20 | 1 | 5 | 4 | 3 | |
| 21 | (2) | (1) | 5 | 4 | |
| 22 | 2 | 1 | | 5 | R0 ← 0 |
| 23 | 3 | 2 | 1 | | |
| 24 | 6 | | | 1 | |

(f) Angenommen, es würde die *Pipeline Freeze* Sprungvorhersage-Strategie verwendet werden: Sobald die bedingte Sprunginstruktion die Stufe $D$ verlässt, wird der Befehl in $F$ eingefroren bis die Sprunginstruktion die Stufe $E$ verlässt und das Sprungziel bekannt ist. Ist die in $F$ eingefrorene Instruktion korrekt, so wandert sie in Stufe $D$ weiter. Entspricht sie nicht dem Befehl an der Sprungadresse, so wird sie verworfen und der korrekte Befehl in $F$ geladen. Würde die *Pipeline Freeze* Strategie die Ausführung des obigen Programms beschleunigen? Begründen Sie ihre Antwort.

Falls kein Sprung ist: langsamer
Falls ein Spring: gleich schnell

Sprungvorhersage=Strategien machen nur Sinn für bedingte Sprünge

2018S

6. *( _____ / 11 Punkte)* Folgender eingerückter Text ist analog zu der Pipeline-Aufgabe in Übung 7:

> Sie haben ein Programm mit bedingten Sprüngen in Pseudo-Code Notation gegeben. Der Prozessor verwendet Pipelining mit *Branch Prediction*. Die Pipeline besteht aus den vier Stufen *Instruction Fetch (F), Decode (D), Execute (E)* und *Memory & Write Back (M/W)*. Werte, die in *M/W* geschrieben werden, stehen im selben Takt in *D* bereits zur Verfügung.
>
> Bedingte Sprünge werden in Stufe *D* erkannt. Ob gesprungen wird oder nicht, steht fest, wenn die Instruktion die Stufe *E* verlässt.
>
> Wandert also eine Instruktion mit bedingtem Sprung in die Stufe *D*, so wird zunächst die nächste Instruktion in *F* geladen. Danach wird, abhängig von der Sprungvorhersage, wieder die nächste Instruktion in die Pipeline geladen (kein Sprung), oder die Instruktion in *F* wird aus der Pipeline geworfen und es wird die Instruktion an der Sprungadresse in *F* geladen (Sprung).
>
> Verlässt die Instruktion mit bedingtem Sprung die Stufe *E* und war die Sprungvorhersage falsch, muss die Pipeline *geflushed* werden. Das bedeutet, die Stufen *F* und *D* werden geleert und *F* wird mit der nächsten korrekten Instruktion geladen.
>
> Die Strategie zur Sprungvorhersage nimmt beim ersten Mal an, dass *nicht* gesprungen wird. Bei jedem weiteren bedingten Sprung wird angenommen, dass die Sprungentscheidung so wie beim letzten Mal ausfällt.
>
> Zeichnen Sie den Ablauf der Pipelineverarbeitung für das gegebene Programm. Setzen Sie die Darstellung solange fort, bis alle Instruktionen vollständig abgearbeitet wurden.

Berücksichtigen Sie auch ***Read-After-Write* Data Hazards** und vermeiden Sie diese mittels *Stalling*. Gestallte Instruktionen müssen eingeklammert werden.

Programm *P*:

*L1:* P $\leftarrow$ *1*
*L2:* f $\leftarrow$ *3*
*L3:* P $\leftarrow P * f$
*L4:* f $\leftarrow f - 1$
*L5:* *if* $(f > 1)$ *goto L3*
*L6:* *NOP*
*L7:* *NOP*

Branch-Predication ist eine Sprungvorhersage

| Zeit ↓ | F | D | E | M/W |
|---|---|---|---|---|
| 1 | L1 | | | |
| 2 | | L1 | | |
| 3 | | | L1 | |
| 4 | | | | L1 |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| 11 | | | | |
| 12 | | | | |
| 13 | | | | |
| 14 | | | | |
| 15 | | | | |
| 16 | | | | |
| 17 | | | | |
| 18 | | | | |
| 19 | | | | |
| 20 | | | | |
| 21 | | | | |

5. *( _____ / 17 Punkte)* Sie arbeiten mit einem Prozessor, der eine fünfstufige Pipeline besitzt: *Instruction Fetch* (IF), *Instruction Decode and Read Registers* (ID), *Execute* (EXE), *Read or Write Memory* (MEM) und *Write Back Registers* (WB). Register werden in der ersten Takthälfte beschrieben und in der zweiten Takthälfte gelesen. Weiters besitzt der Prozessor Hardwareunterstützung für Post-/Pre- In-/Dekrement Operationen bei Speicherzugriffen. Das Ergebnis der Post-/Pre- In-/Dekrement Operation wird dabei in EXE berechnet und gegebenenfalls in WB in ein Register zurück geschrieben. Bedingt durch die Pipelinestruktur kann es zu *RAW Data-Hazards* kommen, welche durch Stalling vermieden werden. Auf dem Prozessor wird folgendes Programm ausgeführt:

```
01:  MOV  R0,  0x9F        # weise R0 die Konstante 0x9F zu
02:  LW   R1,  Mem[R0+]    # lade Wert aus Speicher in R1
03:  LW   R2,  Mem[0xA0]   # lade Wert aus Speicher in R2
04:  ADD  R3,  R1,   R2    # addiere R1 und R2, Ergebnis in R3
05:  SW   R3,  Mem[-R0]    # schreibe R3 in den Speicher
06:  DIV  R4,  R3,   2     # dividiere R3 durch 2, Ergebnis in R4
07:  SW   R4,  Mem[+R0]    # schreibe R4 in den Speicher
```

(a) Auf welche Speicheradressen in hexadezimaler Notation wird in den Zeilen `02`, `05` und `07` zugegriffen und welcher Wert steht in `R0` nach Ausführung des obigen Programms?

Zeile 02: 0x9F

Zeile 05: 0x9F

Zeile 07: 0xA0

R0= 0xA0

(b) Welche der nachfolgenden Addressierungs-Modi werden in Zeile `01` verwendet?
*Register Mode, Immediate Mode, Direct-Addressing Mode, Register-Indirect Mode, Indirect-Addressing Mode*

Immediate Mode

(c) Angenommen der Speicher ist wie folgt initialisiert. Wie sieht die Speicherbelegung nach Ausführung des obigen Programms aus? Befüllen Sie die leeren Stellen der Tabelle in hexadezimaler Notation, falls sich der Wert im Speicher geändert hat.

| Adresse | 9C | 9D | 9E | 9F | A0 | A1 | A2 | ... | 100 | 101 | 102 | 103 | ... | 10F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wert vorher | 69 | 32 | FD | 3 | B | 0 | 41 | ... | A | AF | 23 | B1 | ... | C7 |
| Wert nachher | 69 | 32 | F0 | E | 7 | 0 | 41 | ... | A | A | 23 | B1 | ... | C7 |

(d) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline am Beginn und am Ende leer ist. Berücksichtigen Sie *RAW Data-Hazards* durch Stalling. Klammern Sie gestallte Instruktionen ein.

| Zeit ↓ | IF | ID | EXE | MEM | WB |
|---|---|---|---|---|---|
| 1 | MOV | | | | |
| 2 | LW | MOV | | | |
| 3 | (LW) | (LW) | MOV | | |
| 4 | (LW) | (LW) | | MOV | |
| 5 | LW | LW | | | MOV |
| 6 | ADD | LW | LW | | |
| 7 | (SW) | (ADD) | LW | LW | |
| 8 | (SW) | (ADD) | | LW | LW |
| 9 | SW | ADD | | | LW |
| 10 | (DIV) | (SW) | ADD | | |
| 11 | (DIV) | (SW) | | ADD | |
| 12 | DIV | SW | | | ADD |
| 13 | SW | DIV | SW | | |
| 14 | | (SW) | DIV | SW | |
| 15 | | (SW) | | DIV | SW |
| 16 | | SW | | | DIV |
| 17 | | | SW | | |
| 18 | | | | SW | |
| 19 | | | | | SW |
| 20 | | | | | |
| 21 | | | | | |
| 22 | | | | | |

(e) Können die Instruktionen so umgeordnet werden, dass weniger Stalls auftreten als es bei der gegebenen Anordnung der Fall ist? Falls ja, wie?

```
01:  MOV  R0,  0x9F        # weise R0 die Konstante 0x9F zu
02:  LW   R1,  Mem[R0+]    # lade Wert aus Speicher in R1
03:  LW   R2,  Mem[0xA0]   # lade Wert aus Speicher in R2
04:  ADD  R3,  R1,   R2    # addiere R1 und R2, Ergebnis in R3
05:  SW   R3,  Mem[-R0]    # schreibe R3 in den Speicher
06:  DIV  R4,  R3,   2     # dividiere R3 durch 2, Ergebnis in R4
07:  SW   R4,  Mem[+R0]    # schreibe R4 in den Speicher
```

Zeile 2 & 3 vertauschen: 2 Hazards weniger

7. *( _____ / 11 Punkte)* Sie haben ein Programm mit bedingten Sprüngen in Pseudo-Code Notation gegeben. Der Prozessor verwendet Pipelining mit *Branch Prediction*. Die Pipeline besteht aus den vier Stufen *Instruction Fetch (F), Decode (D), Execute (E)* und *Memory & Write Back (M/W)*. Werte, die in *M/W* geschrieben werden, stehen im selben Takt in *D* bereits zur Verfügung.

   Bedingte Sprünge werden in Stufe *D* erkannt. Ob gesprungen wird oder nicht, steht fest, wenn die Instruktion die Stufe *E* verlässt.
   Wandert also eine Instruktion mit bedingtem Sprung in die Stufe *D*, so wird zunächst die nächste Instruktion in *F* geladen. Danach wird abhängig von der Sprungvorhersage, wieder die nächste Instruktion in die Pipeline geladen (kein Sprung), oder die Instruktion in *F* wird aus der Pipeline geworfen und es wird die Instruktion an der Sprungadresse in *F* geladen (Sprung).

   Verlässt die Instruktion mit bedingtem Sprung die Stufe *E* und war die Sprungvorhersage falsch, muss die Pipeline geflushed werden. Das bedeutet, die Stufen *F* und *D* werden geleert und *F* wird im darauf folgenden Takt mit der nächsten korrekten Instruktion geladen.

   Die Strategie zur Sprungvorhersage nimmt beim ersten Mal an, dass nicht gesprungen wird. Bei jedem weiteren bedingten Sprung wird angenommen, dass die Sprungentscheidung so wie beim letzten Mal ausfällt.

   Zeichnen Sie den Ablauf der Pipelineverarbeitung für das gegebene Programm. Setzen Sie die Darstellung solange fort, bis alle Instruktionen vollständig abgearbeitet wurden.

   Programm *P*:

   *i0:* n ← *3*
   *i1:* s ← *2*
   *i2:* n ← *n-1*
   *i3:* s ← *s+s*
   *i4:* *if (n≠0) goto i2*
   *i5:* *s ← s-1*
   *i6:* nop

| Zeit ↓ | F | D | E | M/W | |
|---|---|---|---|---|---|
| 1 | *i0* | | | | |
| 2 | *i1* | *i0* | | | |
| 3 | *i2* | *i1* | *i0* | | |
| 4 | *i3* | *i2* | *i1* | *i0* | |
| 5 | *i4* | *i3* | *i2* | *i1* | |
| 6 | i5 | i4 | i3 | i2 | n = 2 |
| 7 | i6 | i5 | i4 | i3 | |
| 8 | i2 | | | i4 | |
| 9 | i3 | i2 | | | |
| 10 | i4 | i3 | i2 | | |
| 11 | i5 | i4 | i3 | i2 | n = 1 |
| 12 | i2 | | i4 | i3 | |
| 13 | i3 | i2 | | i4 | |
| 14 | i4 | i3 | i2 | | n = 0 |
| 15 | i5 | i4 | i2 | i2 | |
| 16 | i2 | | i4 | i3 | |
| 17 | i5 | | | i4 | |
| 18 | i6 | i5 | | | |
| 19 | | i6 | i5 | | |
| 20 | | | i6 | i5 | |
| 21 | | | | i6 | |

3. *(10 Punkte)* Sie arbeiten mit einem Prozessor, der eine fünfstufige Pipeline besitzt: *Instruction Fetch* (IF), *Instruction Decode and Read Registers* (ID), *Execute* (EXE), *Read or Write Memory* (MEM) und *Write Back Registers* (WB). Bedingt durch die Pipelinestruktur kann es zu *RAW Data-Hazards* kommen, welche durch Einfügen von NOPs (No Operations) vermieden werden. Register werden in der ersten Takthälfte beschrieben und in der zweiten Takthälfte gelesen. Auf dem Prozessor wird folgendes Programm ausgeführt:

```
SUB  R4,  R5,  R6     # subtrahiere R6 von R5, Ergebnis in R4
SRL  R0,  R4,  1      # right shift von R4 um eine Stelle, Ergebnis in R0
LW   R1,  Mem[0x69]   # lade Wert aus Speicheradresse 0x69 in R1
SW   R1,  Mem[R4]     # schreibe R1 auf Speicheradresse in R4
DIV  R7,  R8,  R9     # dividiere R8 durch R9, Ergebnis in R7
```

(a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline am Beginn und am Ende leer ist. Vergessen Sie nicht an den entsprechenden Stellen genügend NOP Operationen einzufügen.

Gestallt wird nicht, denn der Compiler erkennt Abhaenigkeiten und fuegt NOPs ein.

Da die Register in der ersten Haelfte beschrieben wird und in der zweiten Haelfte gelesen wird, kann wenn die andere Instruktion bei WB die abhaengige Instruktion decoded werden.

| Zeit ↓ | IF | ID | EXE | MEM | WB |
|---|---|---|---|---|---|
| 1 | SUB | | | | |
| 2 | NOP | SUB | | | |
| 3 | NOP | NOP | SUB | | |
| 4 | SRL | NOP | NOP | SUB | |
| 5 | LW | SRL | NOP | NOP | SUB |
| 6 | NOP | LW | SRL | NOP | NOP |
| 7 | NOP | NOP | LW | SRL | NOP |
| 8 | SW | NOP | NOP | LW | SRL |
| 9 | DIV | SW | NOP | NOP | LW |
| 10 | | DIV | SW | NOP | NOP |
| 11 | | | DIV | SW | NOP |
| 12 | | | | DIV | SW |
| 13 | | | | | DIV |

(b) Ordnen Sie die Instruktionen so um, dass möglichst wenige NOP Operationen notwendig und alle Abhängigkeiten berücksichtigt sind.

SUB
LW
DIV
SRL
SW

Die Instruktionen so ordnen, dass moeglichst wenige NOPs enthalten sind.

3. *(12 Punkte)* 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. Analog zur Übung gilt, dass die lesende Instruktion erst dann in Stufe D verarbeitet wird, wenn die schreibende Instruktion Stufe S abgeschlossen hat.
Auf dem Prozessor wird folgendes Programm ausgeführt:

```
DIV   R3, R4, R8    # R4 durch R8 dividieren, Resultat in R3
SUB   R4, R2, R4    # R4 von R2 subtrahieren, Resultat in R4
ADD   R5, R3, R1    # R3 und R1 addieren, Resultat in R5
MULT  R6, R4, R4    # R4 quadrieren, Resultat in R6
SLL   R7, R6, 1     # Shift left von R6 um eine Stelle, Resultat in R7
PUSH  R5            # lege den Inhalt von R5 am Stack ab
```

(a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline am Beginn und am Ende leer ist!

| Zeit ↓ | F | D | E | S |
|---|---|---|---|---|
| 1 | DIV | | | |
| 2 | SUB | DIV | | |
| 3 | ADD | SUB | DIV | |
| 4 | (MULT) | (ADD) | SUB | DIV |
| 5 | MULT | ADD | | SUB |
| 6 | SLL | MULT | ADD | |
| 7 | (PUSH) | (SLL) | MULT | ADD |
| 8 | (PUSH) | (SLL) | | MULT |
| 9 | PUSH | SLL | | |
| 10 | | PUSH | SLL | |
| 11 | | | PUSH | SLL |
| 12 | | | | PUSH |
| 13 | | | | |
| 14 | | | | |

(b) Der Takt für die Pipeline betrage 5 ns. Wie hoch ist der theoretische Durchsatz in MIPS?

$\frac{1}{5\,ns} = \frac{1}{5 \cdot 10^{-9}} = \frac{10^9}{5}$ $0{,}2 \cdot 10^9 = 200 \cdot 10^6 = 200$ MIPS

(c) Angenommen, der reale Durchsatz liegt 25% unter dem theoretischen Durchsatz. Wie viele Instruktionen verlassen in 10 ms durchschnittlich die Pipeline?

200 MIPS · 0,75 = 150 MIPS = $150 \cdot 10^6$

1 s ... $150 \cdot 10^6$ ↘ : 100
0,01 s ... $1{,}5 \cdot 10^6$
10 ms

In einer Sekunde 150 MIPS die Pipeline und in 10ms (0.01s) verlassen 1,5 MIPS die Pipeline

4. *(8 Punkte)* 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. Auf dem Prozessor wird folgendes Programm ausgeführt:

```
MULT  R3, R4, R4      # R4 quadrieren, Resultat in R3
SUB   R4, R2, R4      # R4 von R2 subtrahieren, Resultat in R4
PUSH  R3              # R3 auf dem Stack ablegen
SLL   R6, R4, 1       # Shift left von R4 um eine Stelle, Resultat in R6
SRL   R5, R6, 1       # Shift right von R6 um eine Stelle, Resultat in R5
POP   R9              # oberstes Element vom Stack in R9 laden
```

(a) Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline am Beginn und am Ende leer ist!

*Hinweis:* Möglicherweise werden nicht alle Zeilen der Tabelle benötigt.

| Zeit ↓ | F | D | E | S |
|---|---|---|---|---|
| 1 | MULT | | | |
| 2 | SUB | MULT | | |
| 3 | PUSH | SUB | MULT | |
| 4 | (SLL) | (PUSH) | SUB | MULT |
| 5 | SLL | PUSH | | SUB |
| 6 | SRL | SLL | PUSH | |
| 7 | (POP) | (SRL) | SLL | PUSH |
| 8 | (POP) | (SRL) | | SLL |
| 9 | POP | SRL | | |
| 10 | | POP | SRL | |
| 11 | | | POP | SRL |
| 12 | | | | POP |
| 13 | | | | |
| 14 | | | | |

(b) Geben Sie zwei Maßnahmen zur Vermeidung von *Data Hazards* an!

Stall, NOP

---

Platz für Notizen:

1ns = 1 GHz = 1000 MIPS
2ns = 500 MHz = 500 MIPS

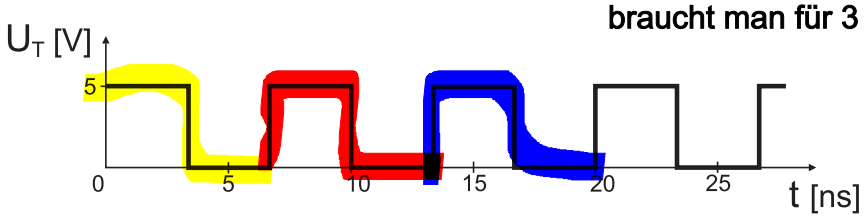
1mycroSe = 1MHz = 1 MIPS

1µs = 1MHz = 1 MIPS

5. *(12 Punkte)* Das unten dargestellte Diagramm beschreibt den Takt eines Prozessors. Der Prozessor besitzt eine Pipeline mit vier Stufen, die jeweils einen Takt benötigen um ihre Aufgabe zu erfüllen.

   *Hinweis:* Versuchen Sie nicht, die Dauer genau eines Taktes aus dem Diagramm auszulesen, sondern die Dauer von Vielfachen des Taktes und rechnen Sie damit!

Wie aus der Grafik hervorgeht, braucht man für 3 Takte 20ns.



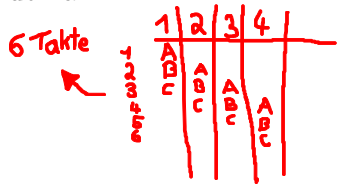

(a) Geben Sie die Taktrate des Prozessors in Megahertz (MHz) an!

3 Takte ..... 20ns
150 Takte ..... 1µs
:50
1µs
⇒ 150 MHz

Eine Mikrosekunde entspricht 1 MHz, weswegen man die 20ns auf eine Mikrosekunde aufrechnet. Dann gehen 150 Takte in 1 Mikrosekunde. Dadurch haben wir 150 MHz.

(b) Wie lange dauert es, drei Instruktionen zu verarbeiten, wenn die Pipeline am Beginn und am Ende leer ist? Geben Sie Ihr Ergebnis in Takten und in Nanosekunden an!

6 Takte ≙ 40ns




(c) Wie hoch ist der theoretische Durchsatz des Prozessors in MIPS?

*Hinweis:* Bei der Berechnung des theoretischen Durchsatzes wird das Ein- und Auslaufen der Pipeline nicht berücksichtigt!

150 MIPS = 150 MHz

(d) Um wie viel Prozent verringert sich der reale Durchsatz gegenüber dem theoretischen, wenn durchschnittlich jede neunte Instruktion einen Takt lang mittels *stall* verzögert werden muss?

real: 9 Instr. ..... 10 Takte
theo.: 10 Instr. ..... 10 Takte
→ 10 %

(e) Angenommen, durch ein Redesign der Prozessorarchitektur könnte eine vierstufige Pipeline wie folgt auf eine achtstufige Pipeline erweitert werden: Jede Stufe wird auf zwei neue Stufen mit exakt der halben Ausführungszeit der ursprünglichen Pipelinestufe aufgeteilt. Welche Konsequenzen hätte dies? Überprüfen Sie folgende Aussagen auf Richtigkeit und kreuzen Sie entsprechend an!
*(korrekte Antwort: +1 Punkt, inkorrekte Antwort: -1 Punkt, keine Antwort: 0 Punkte; Minuspunkte werden gegebenenfalls auf Teilaufgaben a) bis d) übernommen!)*

Darüber kann man keine Aussage treffen, da man nicht Weiss, wie viel gestallt wird

| | | | | |
|---|---|---|---|---|
| Der reale Durchsatz würde sich halbieren. | ❍ | richtig | ☒ | falsch |
| Der theoretische Durchsatz würde sich verdoppeln. | ☒ | richtig | ❍ | falsch |
| Die Durchlaufzeit einer Instruktion würde sich halbieren. | ❍ | richtig | ☒ | falsch |
| Die Taktrate des Prozessors würde sich verdoppeln. | ☒ | richtig | ❍ | falsch |

2013W

Aus den Stufen das Maximum der Instruktionsdauer herauslesen, damit kann man den theoretischen Durchsatz berechnen.

3. *(8 Punkte)* Ein Prozessor besitzt eine fünfstufige Pipeline. Der Instruktionssatz des Prozessors umfasst die drei Instruktionstypen *i1*, *i2* und *i3*. Die Dauer der Ausführung einer Verarbeitungsstufe, abhängig vom Typ der Instruktion, ist in folgender Tabelle angegeben:

| Instruktion | Fetch | Decode | Execute | Memory | Store | Summe |
|---|---|---|---|---|---|---|
| *i1* | 100ns | 50ns | 200ns | 100ns | 150ns | 600ns |
| *i2* | 100ns | 50ns | 150ns | 100ns | 150ns | 550ns |
| *i3* | 100ns | 50ns | 100ns | 200ns | 0ns | 450ns |

Um den theoretischen Durchsatz zu berechnen, muss man die Pipeline Stufe herauslesen, die am längsten braucht. Hier ist es i1 in Execute und i3 in Memory mit 200ns.

(a) Berechnen Sie den theoretischen Durchsatz bei Pipeline-Verarbeitung in MIPS!

200ns → Wird eine Instruktion fertig.

$n = 10^{-9}$

$$\frac{1}{200 \cdot 10^{-9}\,s} = \frac{1}{2 \cdot 10^{-7}\,s} = \frac{1}{2} \cdot 10^{7}\,s^{-1} = \frac{10}{2} \cdot 10^{6}\,s^{-1} = 5\,\text{MIPS}$$

(b) Angenommen, der reale Durchsatz des Prozessors liegt 20% unter dem theoretischen Durchsatz. Wie viele Instruktionen verlassen in 500ms durchschnittlich die Pipeline?

$5 \cdot 0{,}8 = 4\,\text{MIPS}$

500ms ist eine halbe Sekunde

$$4 \cdot \underbrace{0{,}5}_{500\,ms} = 2\ \text{Mio. Instr.}$$

(c) Wofür steht beim Pipelining die Abkürzung MIPS?
*(richtige Antwort: +2 Punkte, falsche Antwort: -2 Punkte, keine Antwort: 0 Punkte)*

- ❍ micro instructions per second
- ❍ micro instructions per stage
- ❍ million instructions per stage
- ☒ million instructions per second

mega

20135

10. *(9 Punkte)* Eine Teilsequenz eines Programmes beinhaltet einen bedingten Sprung. Bei einem Programmdurchlauf wird die folgende Sprungsequenz beobachtet, wobei T für Sprung ausgeführt (*taken*) und N für Sprung nicht ausgeführt (*not taken*) steht:

N N N N N N N N N N
T — T — N — T — N — T — T — T — T — N
b) N T T N T N T T T T N

Die Vorhersage-Genauigkeit einer Strategie zur Sprungvorhersage berechnet sich wie folgt:

$$\text{Genauigkeit (in \%)} = \frac{\text{Anzahl der richtig vorhergesagten Sprünge}}{\text{Anzahl aller Sprünge}} \times 100$$

(a) Berechnen Sie die Vorhersage-Genauigkeit einer statischen Sprungvorhersage, bei der ein Sprung immer als nicht ausgeführt (N) angenommen wird, für die gegebene Sprungsequenz!

$\frac{3}{10}$ = 30%

Die statische Sprungvorhersage sagt immer Nein. Die stimmt in 3 von 10 Fällen, wie man oben sehen kann. Daher 30%

(b) Berechnen Sie die Vorhersage-Genauigkeit der folgenden dynamischen Sprungvorhersage für die gegebene Sprungsequenz:

- Wird eine bedingte Sprunginstruktion erstmals ausgeführt, wird der Sprung als nicht ausgeführt (N) angenommen.
- Im Wiederholungsfall wird stets angenommen, dass der Sprung wie beim letzten Mal ausgeführt wird.

1

$\frac{4}{10}$ = 40%

Der erste Sprung wird als nicht angenommen und die nächsten werden als so wie beim letzten mal angenommen. So wie oben beschrieben. Daraus ergeben sich 4 richtige vorhersagen. 4/10 => 40%

(c) Benennen Sie die 3 Grundarten von Hazards, die beim Pipelining auftreten können!

Structural Hazards, Control Hazards, Data Hazards

2012 S

2. *(13 Punkte)* Sie arbeiten mit einem Prozessor, der eine vierstufige Pipeline besitzt: Fetch Instruction (F), Decode Instruction (D), Execute (E), Store Result (S). Bedingt durch die Pipelinestruktur kann es zu *RAW (Read After Write) Data Hazards* kommen, welche durch verzögerte Ausführung des abhängigen Befehls (*stall*) vermieden werden. Dabei wird eine Instruktion erst dann in Stufe D verarbeitet, wenn die abhängige Instruktion Stufe S abgeschlossen hat.

   Auf diesem Prozessor wird folgendes Programm ausgeführt:

```
i1    ADD  R1, R2, R2      # R1 = R2 + R2
i2    PUSH R1              # R1 auf den Stack legen
i3    MULT R2, R4, R4      # R4 quadrieren, Resultat in R2
i4    SLL R1, R4, 1        # Shift left von R4, Resultat in R1
i5    SUB R4, R4, R1       # R1 von R4 abziehen, Resultat in R4
i6    POP R2               # oberstes Stackelement in R2 lesen
i7    DIV R2, R5, R1       # R5 durch R1 dividieren, Resultat in R2
```

   (a) *(7 Punkte)* Zeichnen Sie die Belegung der Pipeline für das gegebene Programm unter der Annahme, dass die Pipeline zu Beginn und am Ende leer ist.

| Zeit ↓ | F | D | E | S |
|---|---|---|---|---|
| 1 | ADD | | | |
| 2 | Push | ADD | | |
| 3 | (MULT) | (Push) | ADD | |
| 4 | (MULT) | (Push) | | ADD |
| 5 | MULT | Push | | |
| 6 | SLL | MULT | Push | |
| 7 | SUB | SLL | MULT | Push |
| 8 | (POP) | (SUB) | SLL | MULT |
| 9 | (POP) | (SUB) | | SLL |
| 10 | POP | SUB | | |
| 11 | DIV | POP | SUB | |
| 12 | | DIV | POP | SUB |
| 13 | | | DIV | POP |
| 14 | | | | DIV |
| 15 | | | | |
| 16 | | | | |

   (b) *(2 Punkte)* Wieviele *Takte* braucht die CPU um diese Befehlsfolge abzuarbeiten?

14 Takte

   (c) *(4 Punkte)* Welchem *Durchsatz* (Instruktionen pro Sekunde) entspricht das, wenn jede Pipelinestufe 1 ns benötigt?

14 Takte ..... 7 Instruktionen
2 Takte ..... 1 Instruktionen

$\frac{1}{5} = f$

1 ns ≙ 1 GHz ≙ 100 MIPS
2 ns ≙ 500 MHz ≙ 500 MIPS
1 µs ⇒ 1 MHz ≙ 1 MIPS