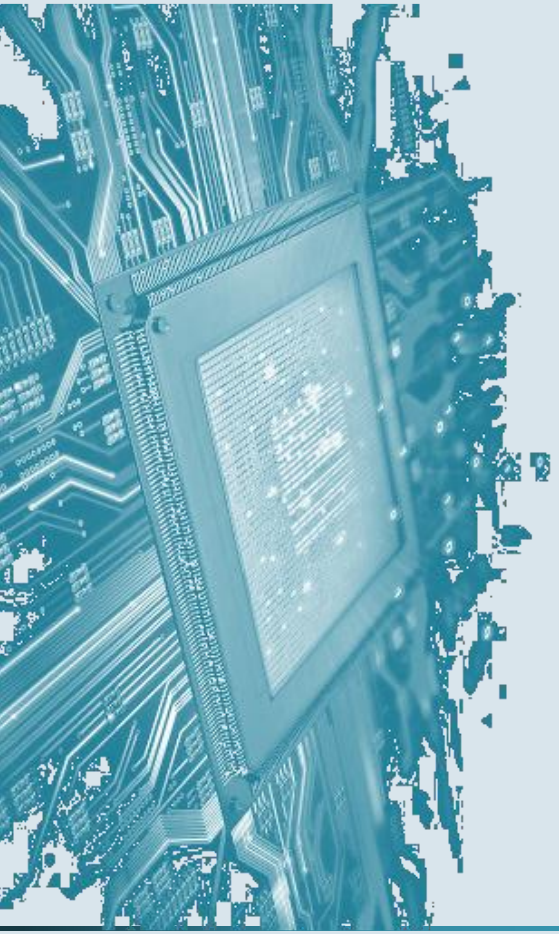


Rechnerarchitektur (AIN 2)

SoSe 2021

Kapitel 4

Multi-Cycle CPU – Pipeline-Architekturen

Prof. Dr.-Ing. Michael Blaich
mblaich@htwg-konstanz.de

Kapitel 4 - Gliederung

Kapitel 4: Multi-Cycle CPU – Pipeline-Architekturen

4.1 Prinzip einer Pipeline

4.1.1 Leistungsmerkmale eines Rechners

4.1.2 Aufbau einer Pipeline

4.1.3 Hazards

4.2 Datapath der MIPS Pipeline

4.3 Control der MIPS Pipeline

4.4 Hazards

4.5 Exceptions

Definition von Leistung/Performance

Hauptsächliche Kenngröße der Leistung: Antwortzeit

- Definition von Leistung

$$Leistung_x = \frac{1}{Antwortzeit_x}$$

- Computer A ist „n-mal so schnell wie“ Computer B, wenn

$$Leistung_A = n \times Leistung_B$$

$$Antwortzeit_A = \frac{Antwortzeit_B}{n}$$

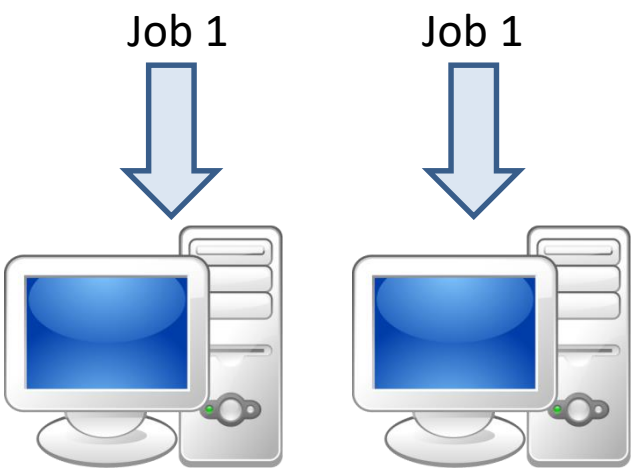
Antwortzeit/CPU-Zeit (Wiederholung)

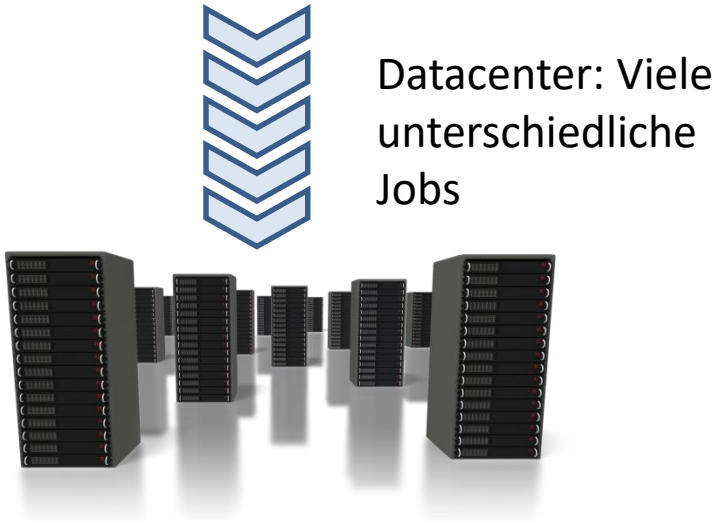
$$\text{Antwortzeit} = \frac{\text{Anzahl Taktzyklen}}{\text{Taktgeschwindigkeit}} = \frac{\text{Instruktionsanzahl } x \text{ CPI}}{\text{Taktgeschwindigkeit}}$$

- Prozessortakt:
  - bestimmt, wann anliegende Werte in Speicherelemente übernommen werden
  - in idealisierter Single-Cycle-CPU benötigen alle Instruktionen einen Takt
  - in einem Takt wird (1) die Instruktion aus dem Speicher geladen, (2) die Werte aus den Registern gelesen, (3) die Berechnungen der ALU durchgeführt, (4) Werte aus dem (Haupt-) Speicher gelesen oder in den (Haupt-)Speicher geschrieben, (5) Werte in das Zielregister geschrieben, (6) der nächste Programmzähler berechnet und in das Register geschrieben
- Taktgeschwindigkeit z.B. 4GHz bei 250ps Taktzyklus
  - Taktzyklus=Dauer eines Taktes, Taktgeschwindigkeit=1/Taktzyklus
- CPI: Clocks per Instruction (Takte pro Instruktion)
  - mittlere Anzahl Takten pro Instruktion
  - in der idealisierten Single-Cycle-MIPs CPU: CPI=1

Leistungskennzahlen für einen Computer

- Antwortzeit: Zeit, um eine Aufgabe zu erledigen
- Durchsatz: Anzahl Aufgaben pro Zeiteinheit





Kenngröße **Antwortzeit**: Computer, der den gleichen Job schneller abarbeitet, ist besser

Kenngröße **Durchsatz**: Computer, der die meisten Jobs abarbeitet, ist am besten

Kapitel 4 - Gliederung

Kapitel 4: Multi-Cycle CPU – Pipeline-Architekturen

4.1 Prinzip einer Pipeline

- 4.1.1 Leistungsmerkmale eines Rechners
- **4.1.2 Aufbau einer Pipeline**
- 4.1.3 Hazards

4.2 Datapath der MIPS Pipeline

4.3 Control der MIPS Pipeline

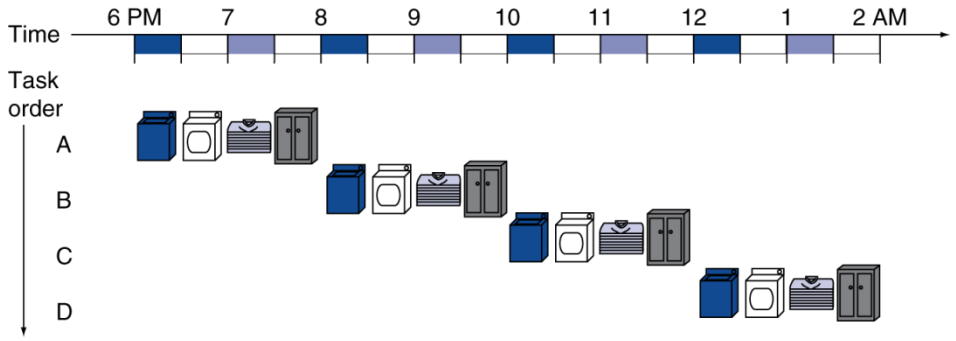
4.4 Hazards

4.5 Exceptions

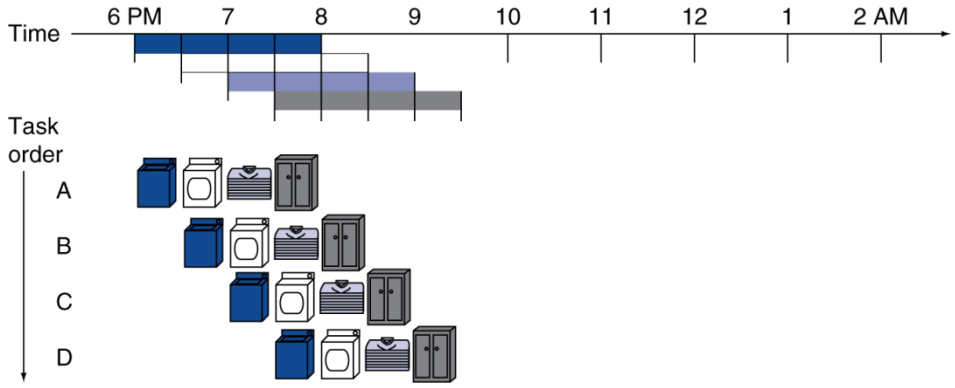
Pipelining: Analogie Wäsche-Waschen

- Wäsche machen umfasst die Tasks Waschen, Trocknen, Bügeln, Einräumen, die die unterschiedlichen Geräte Waschmaschine, Trockner, Bügelbrett und Schrank benötigen
  - 30 Minuten pro Task

Sequentiell:



Pipelining:



4 Wäscheladungen:
- sequentiell: 8h
- pipelining: 3.5 h
- Speed-Up: 8/3.5=2.3

8 Wäscheladungen:
- sequentiell: 16h
- pipelining: 5.5 h
- Speed-Up: 16/5.5=2.9

80 Wäscheladungen:
- sequentiell: 160h
- pipelining: 41.5 h
- Speed-Up: 160/41.5=3.85

Non-stop (Durchsatz):
- sequentiell: 1/2h
- pipelining: 4/2h
- Speed-Up: 4

MIPS Pipeline

- In einer **Multi-Cycle** CPU wird die Ausführung einer Instruktion auf mehrere Takte verteilt. Die Berechnung pro Takt werden so definiert, dass jede Hardware-Komponente jeweils nur in einem Takt benötigt wird. Die Berechnung in einem Takt wird als **Stage** bezeichnet. In einer **Pipeline** werden parallel mehrere Stages für aufeinanderfolgende Instruktionen in einem Takt durchgeführt. Instruktionen werden also teilweise **parallel** ausgeführt.

- Fünf Pipeline Stages, ein Takt pro Stage
  1. IF: Instruktion aus dem Speicher holen (Instruction fetch)
  2. ID: Instruktion decodieren und Register laden (Instruction decode)
  3. EX: Berechnung ausführen, Speicheradresse bestimmen
  4. MEM: Zugriff auf Speicher
  5. WB: Ergebnis in Register speichern

Pipelining Speed-Up

- Durch eine Vereinfachung der Berechnungen (der logischen Schaltungen), die pro Takt durchgeführt werden, kann die Taktdauer verringert und damit die Taktrate der CPU erhöht werden.
  - in einer Multi-Cycle (Mehrtakt) CPU sind die Takte daher kürzer
  - die Berechnung einer einzigen Instruktion benötigt aber mehrere dieser „kürzeren“ Takte, so dass die Berechnung einer Instruktion tendenziell länger dauert als bei einer Single-Cycle (Eintakt) CPU
- Durch die parallele Ausführung mehrere Instruktionen in unterschiedlichen Stages wird ein Speed-Up im Sinne eines höheren Durchsatzes erreicht
  - Durchsatz=Anzahl abgearbeiteter Instruktionen pro Zeit

Beispiel: Taktzyklus Single Cycle CPU

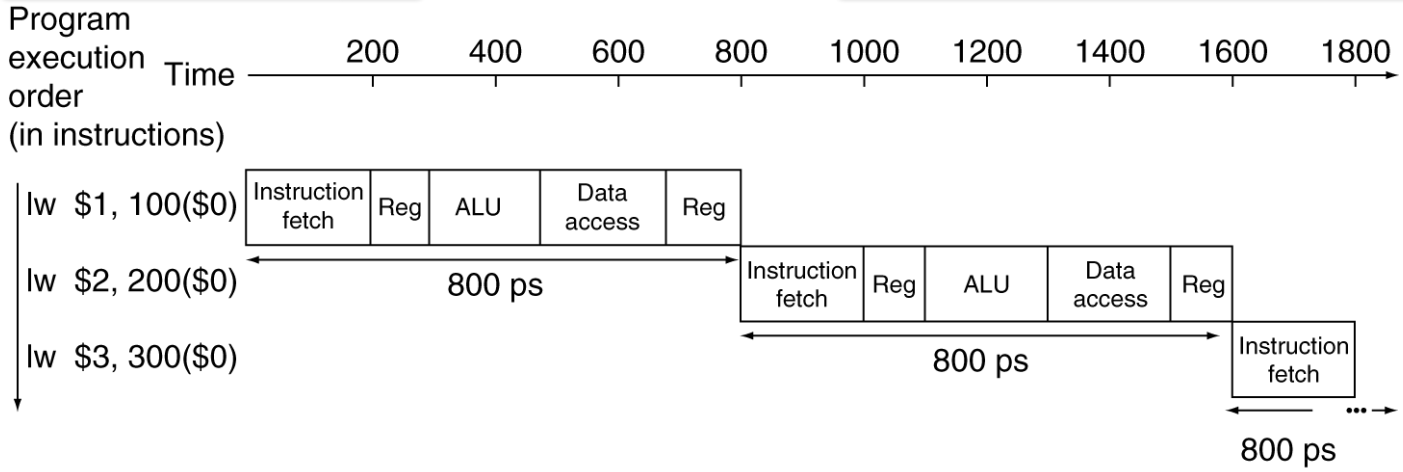
- verschiedene Instruktionen durchlaufen unterschiedliche Berechnungsschritte in verschiedenen Hardware-Komponenten des DataPath
- unterschiedliche Hardware-Komponenten haben unterschiedliche Verarbeitungszeiten (siehe Tabelle)
- Taktzyklus der Single-Cycle CPU wird bestimmt durch komplexeste (langsamste) Instruktion:
  - 800ps für load word
  - CPU Speed: 1,25 GHz

Instruction	Instruction 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

Beispiel: Taktzyklus Multi-Cycle CPU

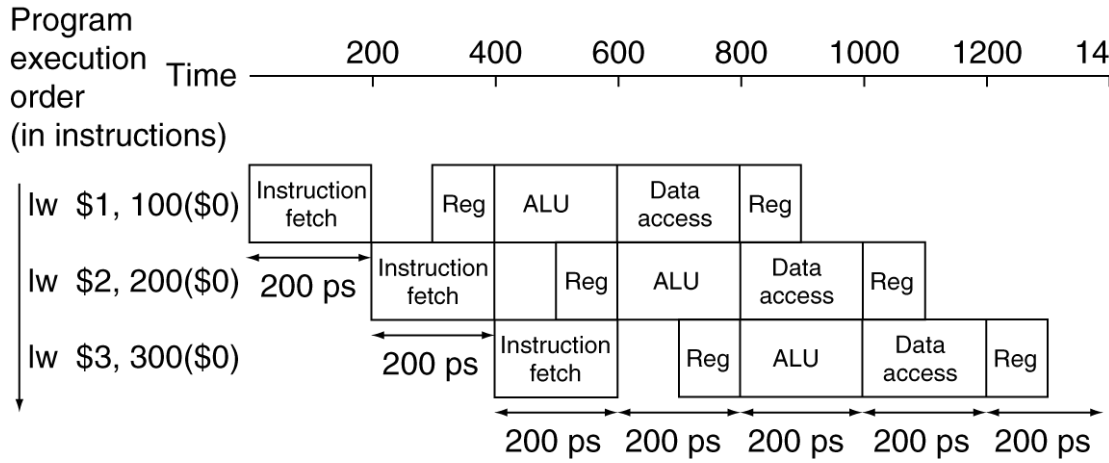
Single-cycle CPU

Taktzyklus 800ps ➔ 1,25 GHZ CPU



Pipelined Multi-Cycle CPU

Taktzyklus 200ps ➔ 5 GHz CPU



Pipelining Speed-Up

- Idealfall: ausgeglichene Stages
  - Stages fangen immer gleichzeitig an
  - wir bestimmen den Durchsatz als die Zeit zwischen dem Starten zweier Instruktionen bzw. dem Fertigstellen zweier Instruktionen

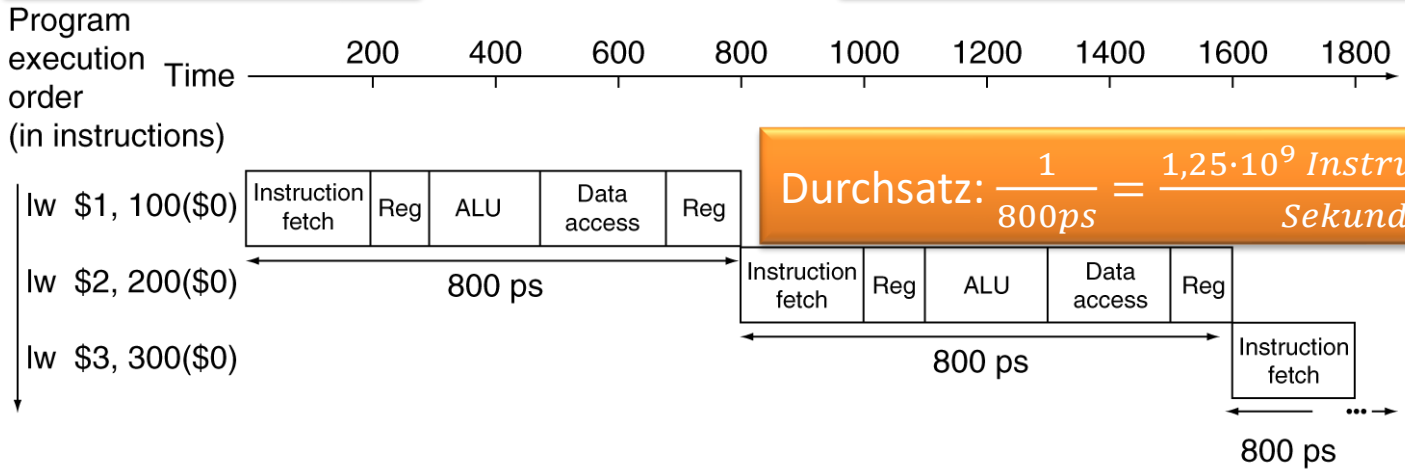
$$\text{Zeit zw. Instruktionen}_{\text{pipelined}} = \frac{\text{Zeit zw. Instruktionen}_{\text{non-pipelined}}}{\text{Anzahl Stages}}$$

- bei unausgeglichenen Stages geringerer Speed-Up
  - im Beispiel nur Speed-Up von 4, da die Stages ID und WB nur 100ps benötigen, während die Stages IF, EX und MEM jeweils 200ps benötigen
- Speed-Up
  - bezieht sich nur auf den Durchsatz (Anzahl Instruktionen pro Zeit)
  - die Latenz (Abarbeitungszeit) einer Instruktion bleibt gleich oder wird größer
    - im Beispiel steigt die Latenz einer Instruktion von 800ps auf 1000ps

Beispiel: Speed-Up Vergleich

Single-cycle CPU

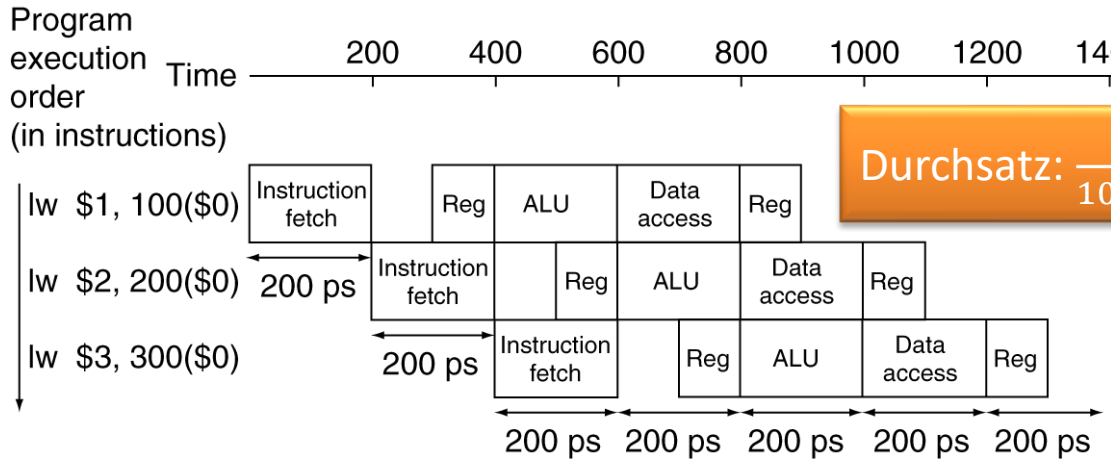
Taktzyklus 800ps ➔ 1,25 GHZ CPU



Durchsatz: $\frac{1}{800ps} = \frac{1{,}25 \cdot 10^9\ Instruktionen}{Sekunde}$

Pipelined Multi-Cycle CPU

Taktzyklus 200ps ➔ 5 GHz CPU



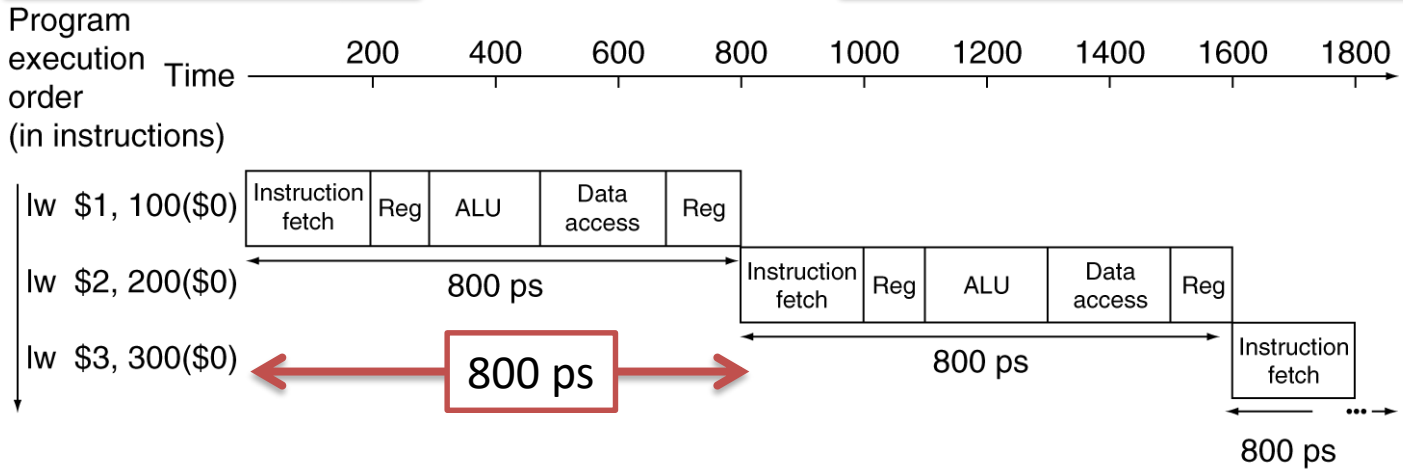
Durchsatz: $\frac{5}{1000ps} = \frac{5 \cdot 10^9\ Instruktionen}{Sekunde}$

Speed-Up: 4

Beispiel: Latenz Vergleich

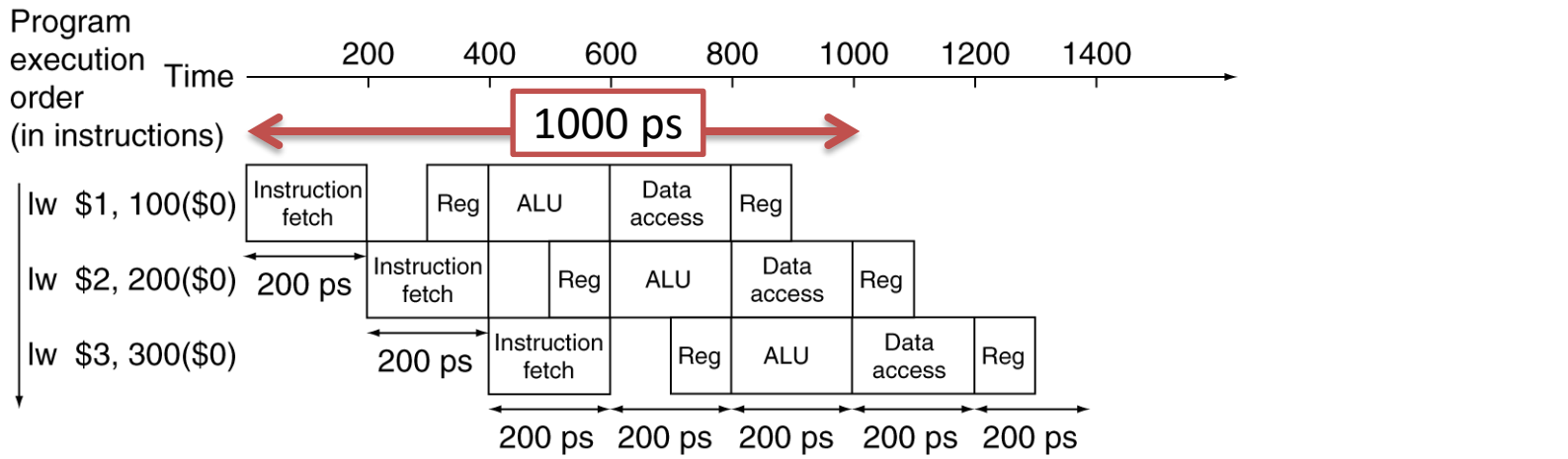
Single-cycle CPU

Taktzyklus 800ps ➔ 1,25 GHZ CPU



Pipelined Multi-Cycle CPU

Taktzyklus 200ps ➔ 5 GHz CPU



ISA Design für Pipelining

- Pipelining funktioniert am effizientesten bzw. ist am einfachsten zu implementierten, wenn der Befehlssatz möglichst einfache und einheitliche Instruktionen enthält
  - Moderne RISC Befehlssätze (z.B. MIPS ISA) werden speziell für Pipelining-Support entwickelt
  - Ältere CISC Befehlssätze (z.B. x86) wurden ohne Gedanken an Pipelining entwickelt, so dass komplexere Hardware notwendig ist, um effizientes Pipelining zu ermöglichen
    - x86 Hardware teilt Instruktionen zunächst in „homogenere“ Mikro-Instruktionen auf, die dann von der Pipeline verarbeitet werden

Was macht die MIPS ISA gut für Pipelining?

- alle Instruktionen in 32 Bit codiert
  - können in einem Taktzyklus geladen und dekodiert werden
  - x86 Instruktionen: Instruktionen mit 1 bis 17 Bytes
- wenige und regelmäßige Instruktionen
  - Register können in einem Taktzyklus dekodiert und gelesen werden
- Adressierung von Speicherzugriffen
  - Trennung von Adressberechnung und Zugriff
  - Berechnung in der 3. Stage, Zugriff in der 4. Stage
- Regelmäßige Anordnung von Speicheroperanden
  - Zugriff in einem Taktzyklus

Kapitel 4 - Gliederung

Kapitel 4: Multi-Cycle CPU – Pipeline-Architekturen

4.1 Prinzip einer Pipeline

- 4.1.1 Leistungsmerkmale eines Rechners
- 4.1.2 Aufbau einer Pipeline
- **4.1.3 Hazards**

4.2 Datapath der MIPS Pipeline

4.3 Control der MIPS Pipeline

4.4 Hazards

4.5 Exceptions

Hazards (Konflikte)

- Hazards sind Situationen, die die Ausführung der nächsten Instruktion im nächsten Takt verhindern

- Arten von Hazards
  - Strukturelle Hazards:
    - Eine benötige Ressource ist belegt und verhindert die Ausführung der Instruktion
  - Daten-Hazards:
    - Eine Instruktion benötigt einen Registeroperanden, der von einer vorigen Instruktion bestimmt wird und noch nicht im Register gespeichert ist.
    - Die Instruktion muss warten, bis der Registeroperand bestimmt und in das entsprechende Register geschrieben ist.
  - Control-Hazards:
    - Die Adresse der nächsten auszuführenden Instruktion steht noch nicht fest, wenn die Instruktion aus dem Instruktionsspeicher geladen wird
      - Branches und Sprünge

Strukturelle Hazards (Structure Hazard)

Als **Stalling** bezeichnet man die Situation, in der die nächste Instruktion nicht direkt von der Pipeline bearbeitet werden kann. Es entsteht eine Pipeline **Bubble.**

- Konflikt um den Zugriff auf eine Ressource

- Beispiel: MIPS Pipeline mit nur einem Speicher für Daten- und Instruktionen
  - IF und MEM können bei Datentransferinstruktionen nicht gleichzeitig ausgeführt werden
    - Laden/Speichern von Daten bei Datentransferinstruktion benötigt Speicher
    - Laden von Instruktionen benötigt Speicher
    - beides geht nicht in der gleichen Stage
  - Laden der nächsten Instruktion in IF Instruktion muss einen Takt warten (**Stalling**)
    - eine Pipeline-Blase (Pipeline Bubble) entsteht

- Konsequenz:
  - CPU mit Pipeline benötigt getrennten Speicherzugriff für Programmcode und Daten
  - wird beispielsweise über zwei separate Caches realisiert
    - siehe Kapitel 5: Speicherhierarchie

Kapitel 4 - Gliederung

Kapitel 4: Multi-Cycle CPU – Pipeline-Architekturen

4.1 Prinzip einer Pipeline

4.1.1 Leistungsmerkmale eines Rechners

4.1.2 Aufbau einer Pipeline

4.1.3 Hazards

4.1.3.1 Data Hazards

4.1.3.2 Control Hazards

4.2 Datapath der MIPS Pipeline

4.3 Control der MIPS Pipeline

4.4 Hazards

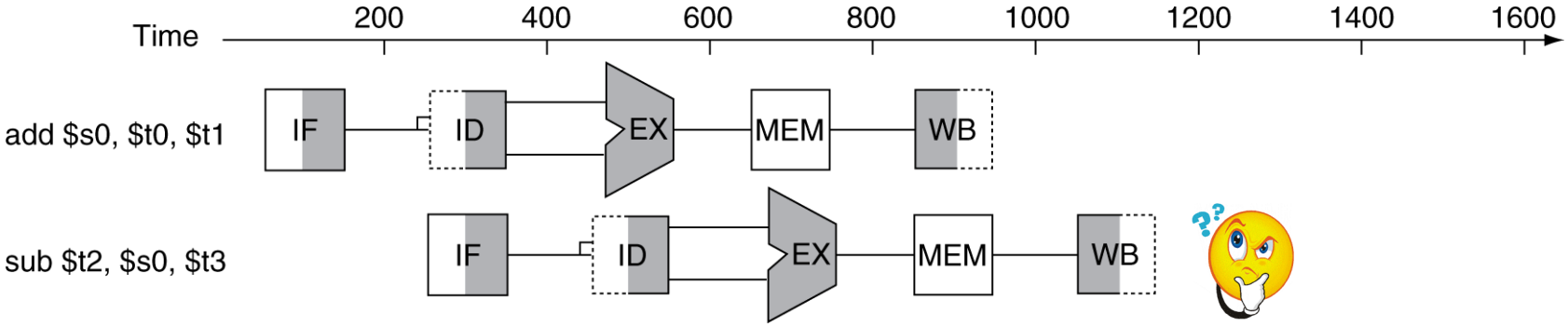
4.5 Exceptions

Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist

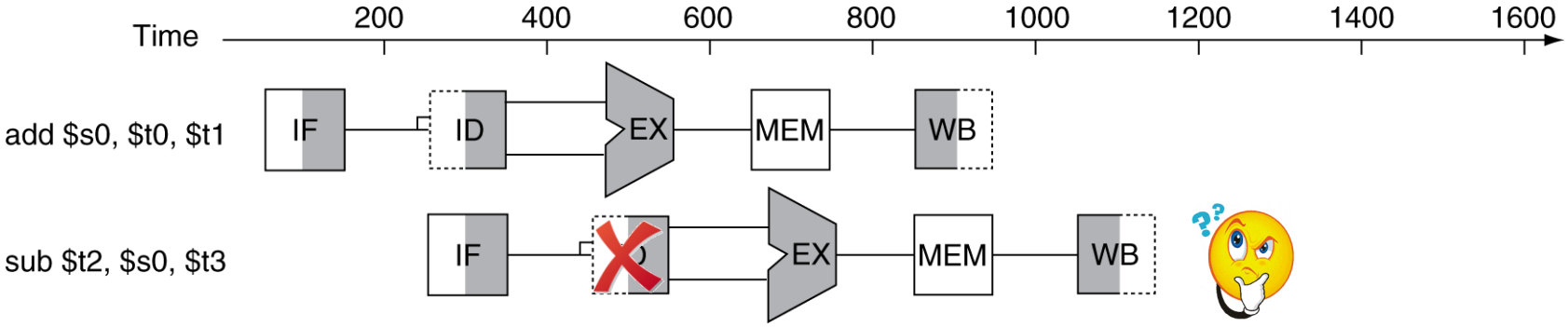


Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



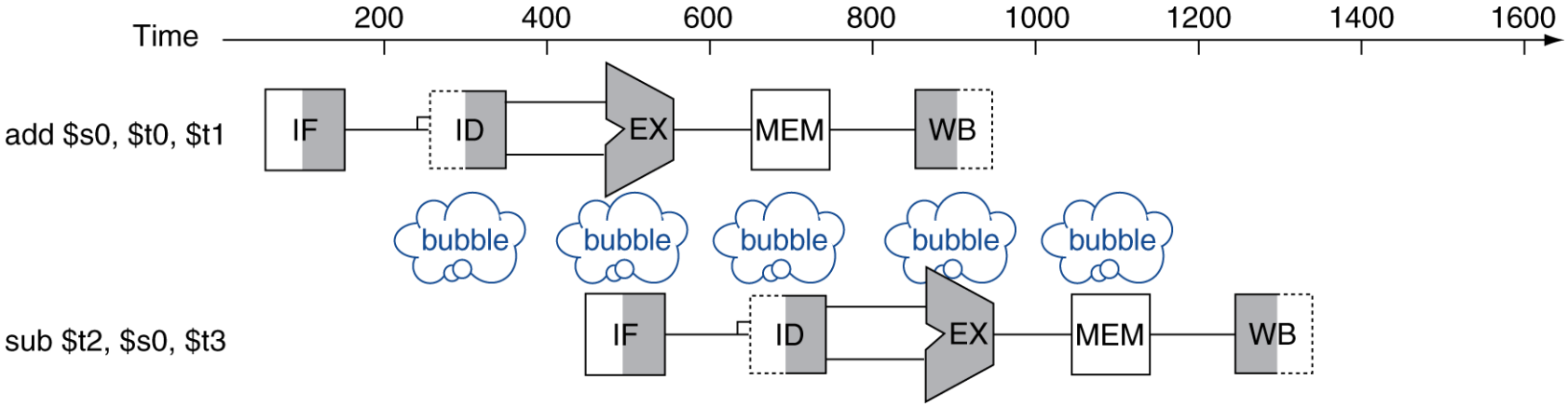
ID Stage kann noch nicht ausgeführt werden, da das Ergebnis der ADD-Instruktion noch nicht in Register $s0 steht.

Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



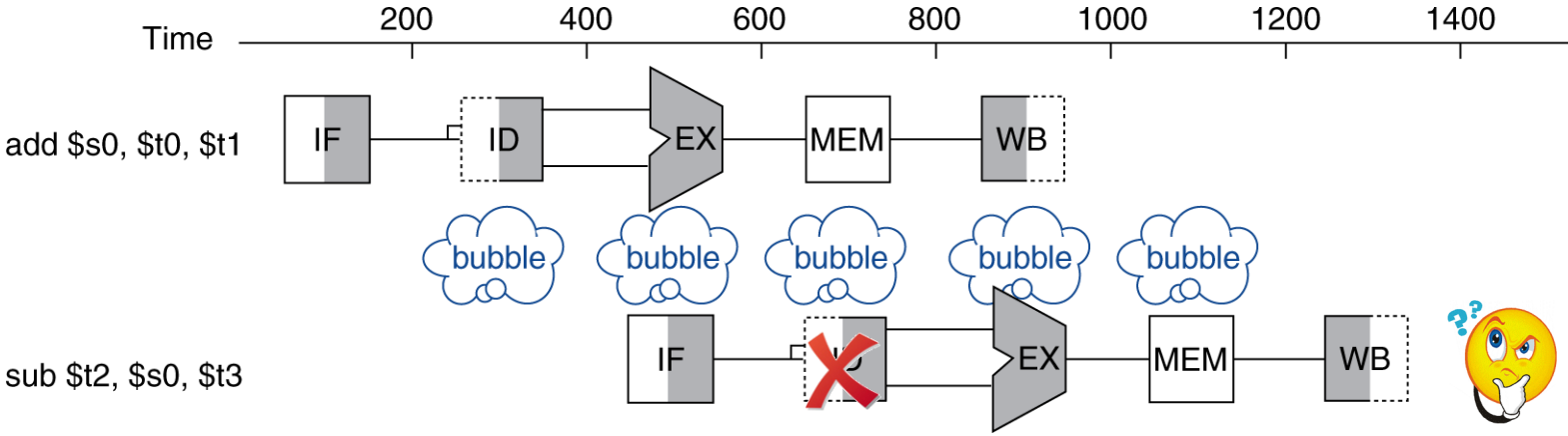
Das Steuerwerk
- fügt eine Bubble ein, d.h. eine Instruktion, in der nichts berechnet wird
- sorgt dafür, dass sich der Program Counter nicht ändert und in der IF-Stage die gleiche Instruktion (im Beispiel SUB) noch einmal geladen wird

Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



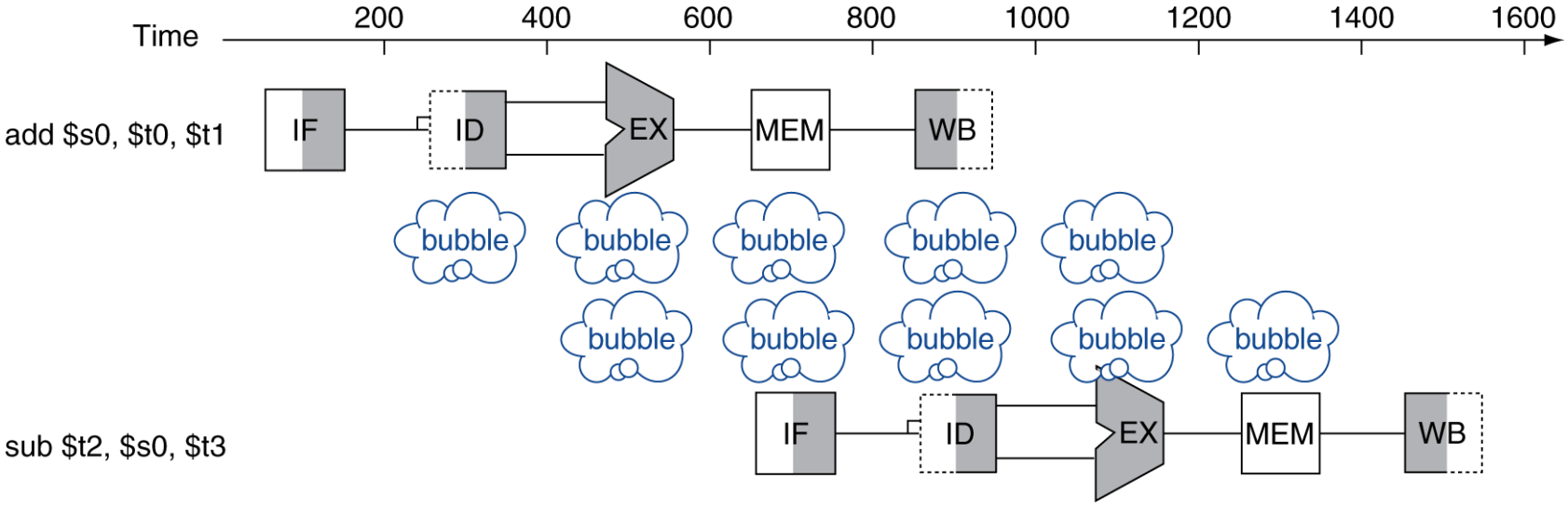
ID Stage kann immer noch nicht ausgeführt werden, da das Ergebnis der ADD-Instruktion nach wie vor nicht in Register $s0 steht.

Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



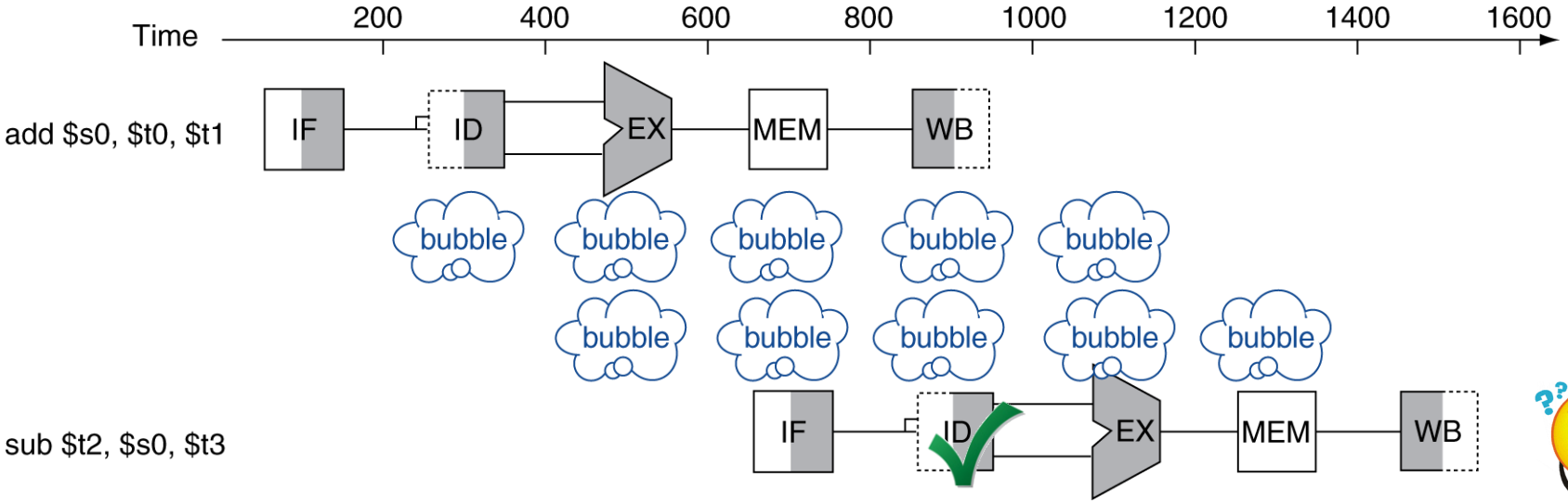
Das Steuerwerk fügt wiederum eine Bubble ein und in der IF-Stage wird noch einmal die gleiche Instruktion geladen.

Daten-Hazard (Data Hazard) - Stalling

- Beispiel:

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



- ID und WB benötigen nur 100ps also einen halben Takt
- Speichern in (WB) und Laden (ID) aus einem Register ist im gleichen Takt möglich
  - WB Stage wird in der ersten Takthälfte ausgeführt
  - ID Stage wird in der zweiten Takthälfte ausgeführt

Forwarding (auch Bypassing)

Forwarding ist eine Technik, um Daten-Hazards zu vermeiden.

- Idee:
  - eigentlich ist es nicht nötig zu warten, bis ein benötigter Wert im Register steht
  - der Wert kann schon genutzt werden, sobald er in der CPU bekannt ist, d.h.
    - in der EX Stage berechnet wurde (wie im Beispiel)
    - in der MEM Stage aus dem Speicher geladen wurde
- Forwarding heißt, dass ein Wert, der in der EX oder MEM Stage bestimmt wird, direkt in der EX Stage als Operand verwendet werden kann
  - es wird ein **Bypass** vom Datenpfad einer früheren Instruktion in den Datenpfad einer darauffolgenden Instruktion gelegt
- Der berechnete Wert wird aber in WB in das vorgesehen Register geschrieben, auch wenn eine andere Instruktion ihn zwischenzeitlich durch Forwarding benutzt hat.

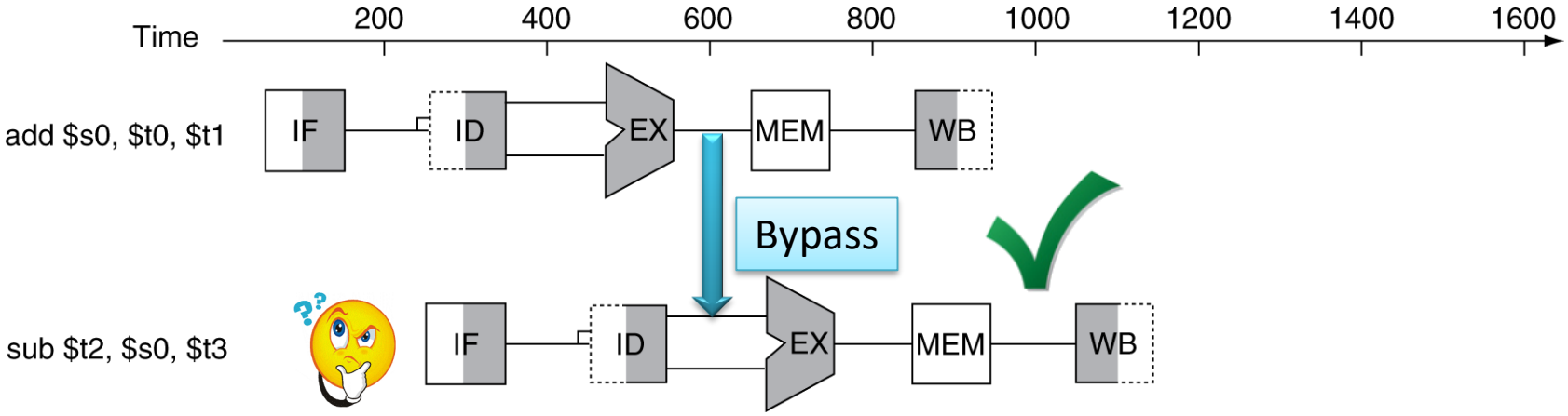
Forwarding (auch Bypassing)

Forwarding ist eine Technik, um Daten-Hazards zu vermeiden

- Beispiel:

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

- Subtraktion kann erst stattfinden, wenn Ergebnis der Addition bekannt ist



Das Steuerwerk erkennt, dass der Wert, den ADD in das Register $s0 speichern wird, im letzten Takt (Takt n) in der EX-Stage berechnet wurde. Er kann also im nächsten Takt (Takt n+1) in der EX Stage der SUB-Instruktion verwendet werden. Die SUB-Instruktion kann im nächsten Takt ausgeführt werden und Stalling wird vermieden.

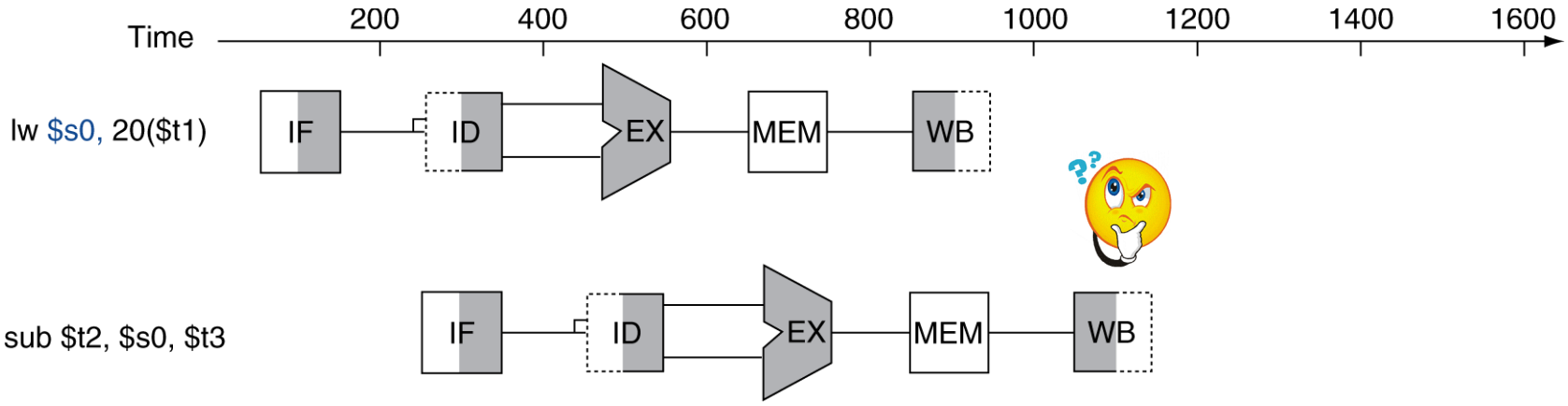
Load-Use Data Hazard

Load-Use Data Hazard: Auf eine Load-Instruktion folgt eine Instruktion, die den geladenen Wert als Operand benötigt.

- Beispiel:

```
lw      $s0, 20($t1)
sub     $t2, $s0, $t3
```

- Subtraktion kann erst nach Schreiben in Register stattfinden



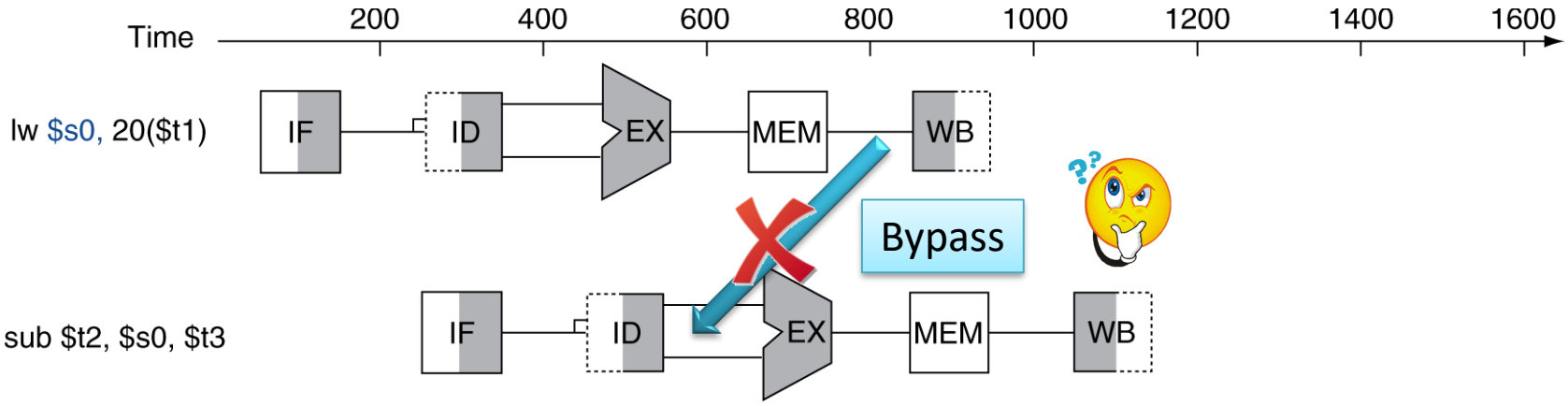
Load-Use Data Hazard

Ein Load-Use-Hazard kann durch Forwarding nicht vermieden werden.

- Beispiel:

```
lw      $s0, 20($t1)
sub     $t2, $s0, $t3
```

- Subtraktion kann erst nach Schreiben in Register stattfinden



Forwarding geht nicht rückwärts in der Zeit.

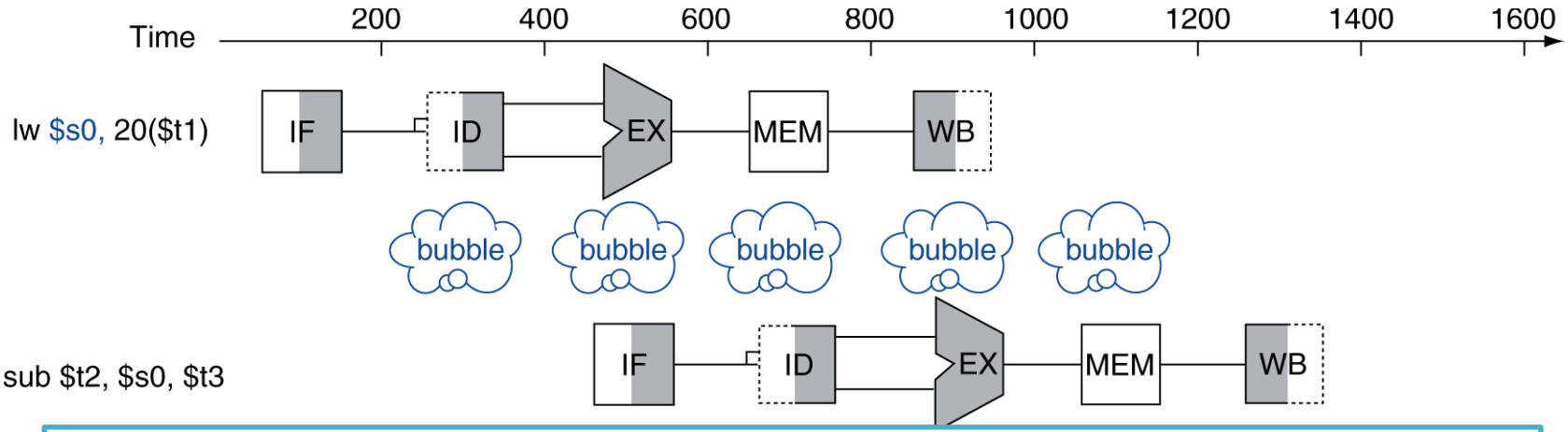
Load-Use Data Hazard

- Load-Use Data Hazard: Register, in das ein Ladebefehl schreibt, wird in der darauffolgenden Instruktion für Berechnung verwendet
- Keine Vermeidung einer Pipeline-Bubble durch Forwarding bei einem Load-Use Hazard

- Beispiel:

```
lw    $s0, 20($t1)
sub   $t2, $s0, $t3
```

- Subtraktion kann erst nach Schreiben in Register stattfinden



Das Steuerwerk erkennt, dass auch mit Forwarding der Operand nicht zur Verfügung steht und fügt wiederum eine Bubble ein. In der IF-Stage wird noch einmal die gleiche Instruktion geladen.

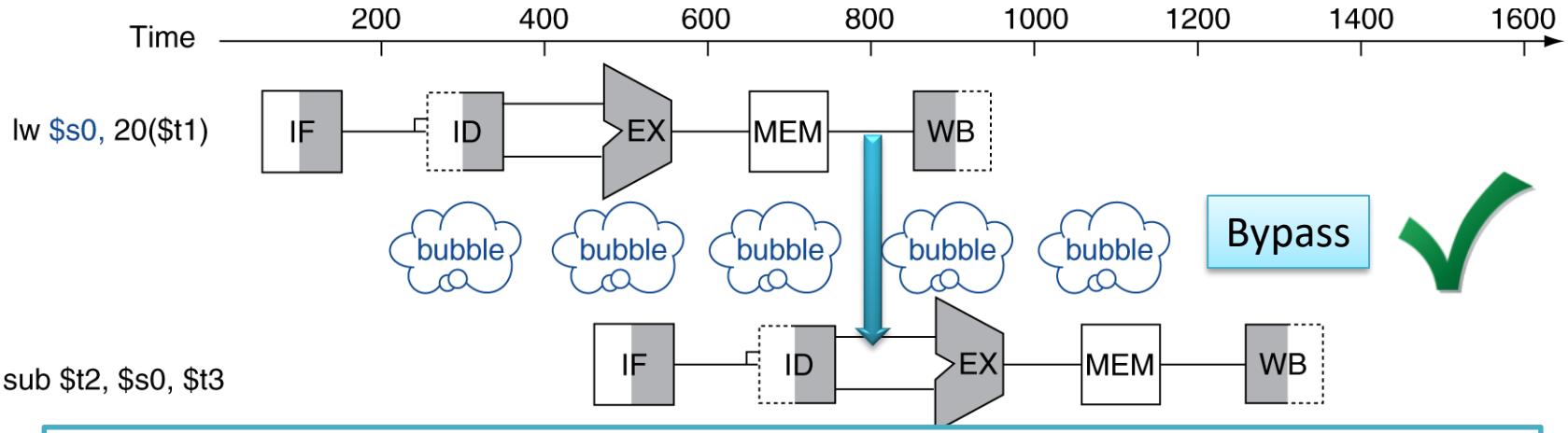
Load-Use Data Hazard

Forwarding reduziert bei einem Load-Use-Hazards die Anzahl der Bubbles von zwei auf eine.

- Beispiel:

```
lw      $s0, 20($t1)
sub     $t2, $s0, $t3
```

- Subtraktion kann erst nach Schreiben in Register stattfinden



Der Operand wird in der MEM-Stage geladen und kann im nächsten Takt von der SUB-Instruktion in der EX-Stage verwendet werden.

Code Scheduling

- Nicht alle Data-Hazards lassen sich durch Forwarding vermeiden
  - der Load-Use-Hazard ist nur ein Beispiel
- Instruktionen müssen nicht zwingend in der Reihenfolge ausgeführt werden, in der sie programmiert werden
  - ein Programmierer schreibt ein Programm, indem er logische Aufgaben nacheinander löst, so dass das Programm gut strukturiert und lesbar erscheint
  - manche Teile eines Programms sind auch weitgehend unabhängig voneinander
  - dies hat zur Folge, dass nicht jede Instruktion exakt an der Stelle ausgeführt werden muss, an der sie im Code steht
- Code Scheduling bezeichnet das Ändern der Reihenfolge von Instruktionen, um Unterbrechungen der Pipeline zu vermeiden oder zu reduzieren
  - Code Scheduling bewirkt keine Änderung in der Programmlogik
  - Code Scheduling verändert keine Instruktionen (oder höchstens sehr einfache Instruktionen)
- Code Scheduling wird normalerweise vom Compiler übernommen

Beispiel: Code Scheduling

- Logische Programmstruktur: erst Gleichung 1, dann Gleichung 2
- Aber: Berechnung der ersten und zweiten Gleichung unabhängig voneinander

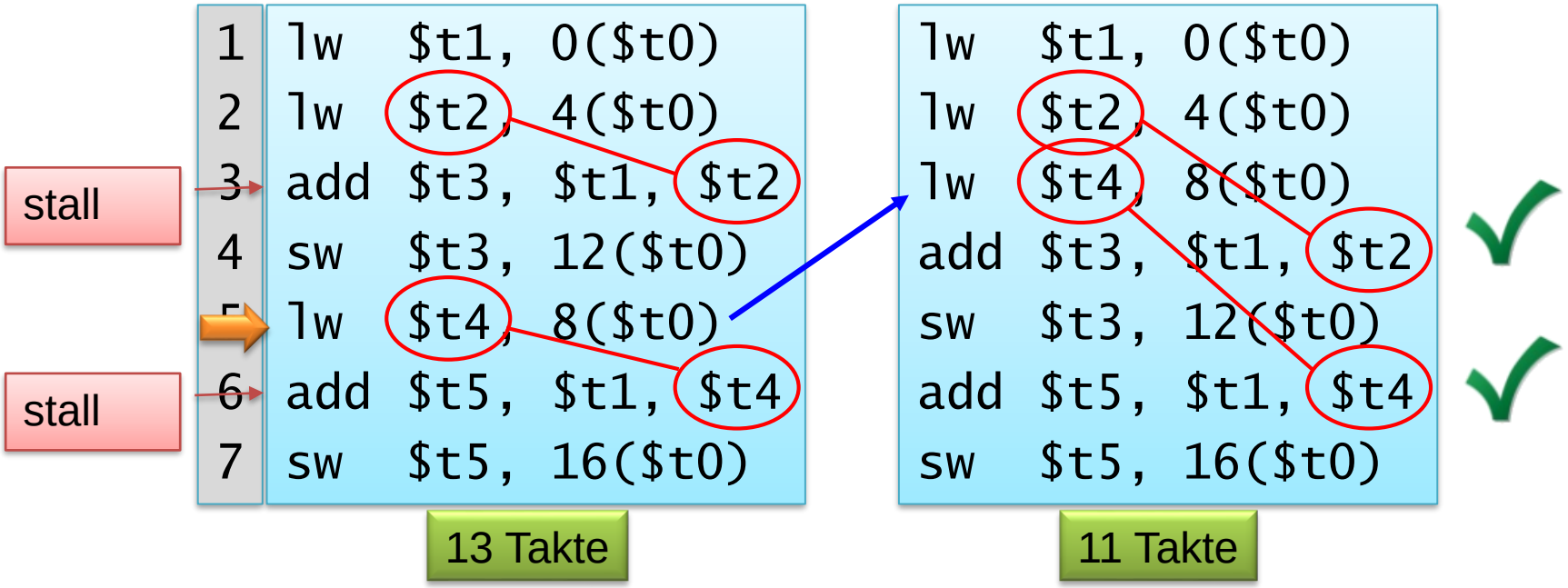
- Beispiel: Variablen müssen erst aus dem Speicher geladen werden

```
A = B + E;
C = B + F;
```

```
1   lw  $t1, 0($t0)
2   lw  $t2, 4($t0)
3   add $t3, $t1, $t2    nach Zeilen 1 und 2
4   sw  $t3, 12($t0)     nach Zeile 3
5   lw  $t4, 8($t0)
6   add $t5, $t1, $t4    nach Zeilen 1 und 5
7   sw  $t5, 16($t0)     nach Zeile 6
```

Beispiel: Code Scheduling

- Es treten zwei Load-Use-Hazards auf
- Code Scheduling: Laden von F (Zeile 5) wird vor Zeile 3 gezogen
- Lösung:
  - die beiden Load-Use-Hazards wurden beseitigt
  - Stalling wird vermieden
  - die Laufzeit wird um 2 Takte von 13 Takten auf 11 Takte verringert



Kapitel 4 - Gliederung

Kapitel 4: Multi-Cycle CPU – Pipeline-Architekturen

4.1 Prinzip einer Pipeline

4.1.1 Leistungsmerkmale eines Rechners

4.1.2 Aufbau einer Pipeline

4.1.3 Hazards

4.1.3.1 Data Hazards

4.1.3.2 Control Hazards

4.2 Datapath der MIPS Pipeline

4.3 Control der MIPS Pipeline

4.4 Hazards

4.5 Exceptions

Control-Hazards

Control Hazards treten auf, wenn die Adresse der auszuführenden Instruktion zu Beginn der IF-Stage noch nicht bekannt ist.

- Bedingte Sprünge bestimmen den Ablauf des Programms
  - welche Instruktion als nächstes aus dem Speicher geholt wird, hängt vom Ergebnis des Vergleichs der Branch-Instruktion ab
  - um ohne Stalling fortzufahren, müsste die nächste Instruktion bereits in der IF-stage bestimmt und in das PC-Register geschrieben werden
- Aber wenn die Pipeline die nächste Instruktion in der IF-Stage lädt, befindet sich die Branch-Instruktion gerade in der ID Stage
  - die Werte der Register werden geladen
  - danach müssen Sie noch verglichen und die Sprungadresse berechnet werden
    - entsprechend unserer bisherigen CPU wird die **Sprungadresse in der MEM Stage** geschrieben
    - mit zusätzlicher Hardware kann die **Sprungadresse in der ID Stage** entschieden und in das Register PC geschrieben werden

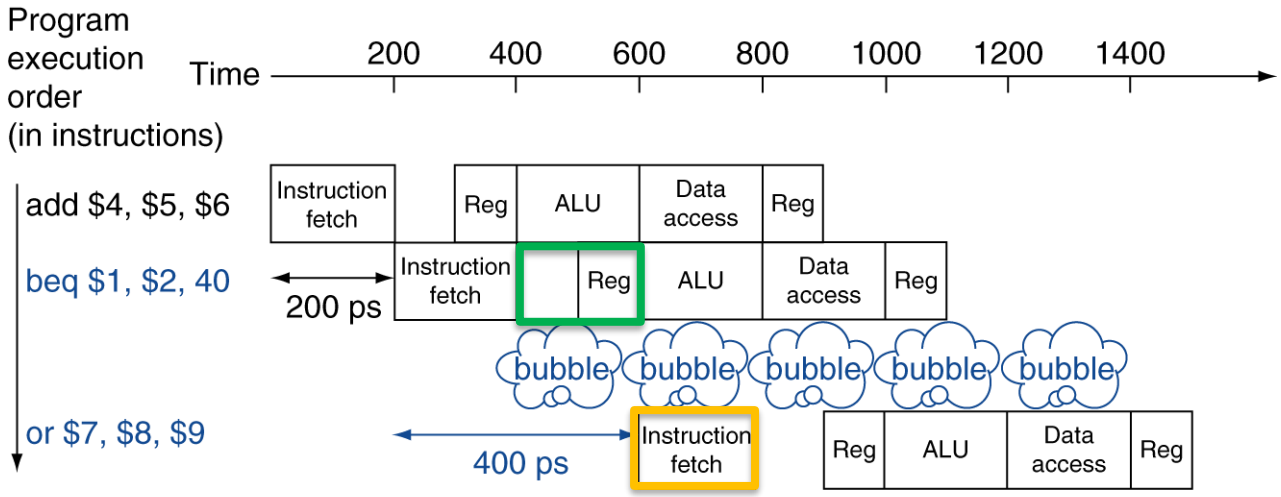
Control-Hazards

- Optionen der Pipeline sind
  - Stop-and-Wait oder Stall-on-Branch:
    - keine Instruktion ausführen und **Bubbles generieren**, bis die richtige Instruktion bekannt ist
    - Performance: jede Branch-Instruktion führt zu Stalling
  - Continue:
    - immer die nächste(n) Instruktion (PC+4) ausführen
    - Auswirkungen der Instruktionen (PC+4) löschen und mit Instruktion an Sprungadresse fortfahren, falls ein Sprung durchgeführt werden muss
    - Performance: Stalling wird vermieden, falls kein Sprung ausgeführt wird
  - Branch Prediction
    - intelligent vorhersagen, ob ein Sprung durchgeführt wird oder nicht
    - Auswirkungen der Instruktionen (PC+4) löschen und mit Instruktion an Sprungadresse fortfahren, falls ein Sprung durchgeführt werden muss
    - Performance: Stalling wird vermieden, wenn die Vorhersage zutrifft
      - deutlich besser als „Continue“

Stall on Branch

Stall on Branch: Pipeline wird nach einem bedingten Sprung unterbrochen und eine Pipeline-Bubble entsteht

- Operationsmodus: Stop-and-Wait
  - Warte, bis die nächste Instruktion entschieden ist
  - Hole nächste Instruktion



In diesem Beispiel wird die Adresse der nächsten Instruktion in der ID-Stage der Branch-Operation bestimmt. Im darauffolgenden Takt kann diese Instruktion in der IF-Stage geladen werden.

Abhilfe: Branch Prediction

- Branch Prediciton: bei manchem bedingten Sprüngen ist es möglich, die wahrscheinliche Sprungadresse vorherzusagen und auszuführen
  - bei falscher Vorhersage muss korrigiert werden

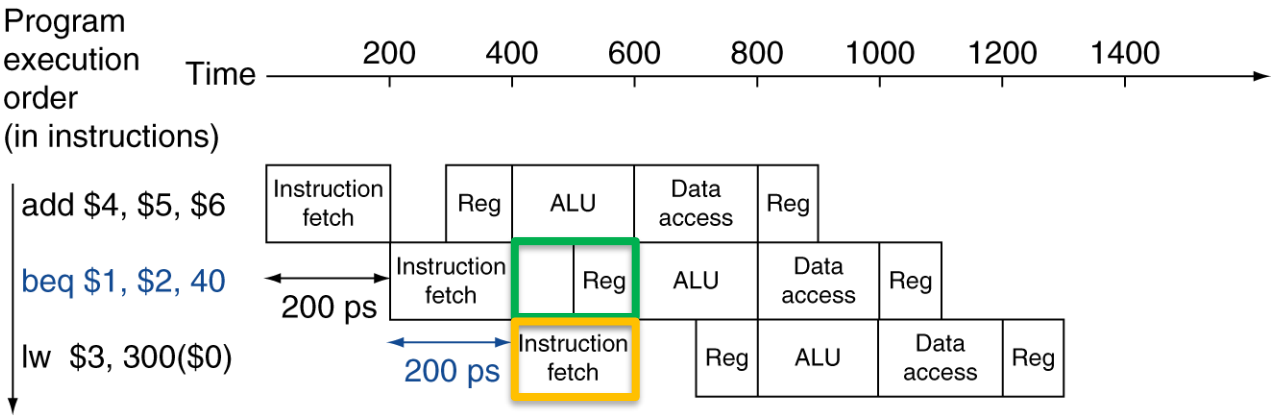
- Lange Pipelines können die nächste Instruktion bei bedingten Sprüngen nicht frühzeitig berechnen
  - Performance-Einbußen durch Stalling können eklatant und unakzeptabel sein
- Abhilfe: Vorhersage des bedingten Sprungs
  - Stalling nur bei falsch-vorhergesagten Sprungentscheidungen

- MIPS Pipeline
  - kann nicht-entschiedene Branches vorhersagen
  - kann vorhergesagte Instruktion ohne Stalling laden
    - IF-Stage der vorhergesagten Instruktion folgt der IF-Stage der Branch-Instruktion

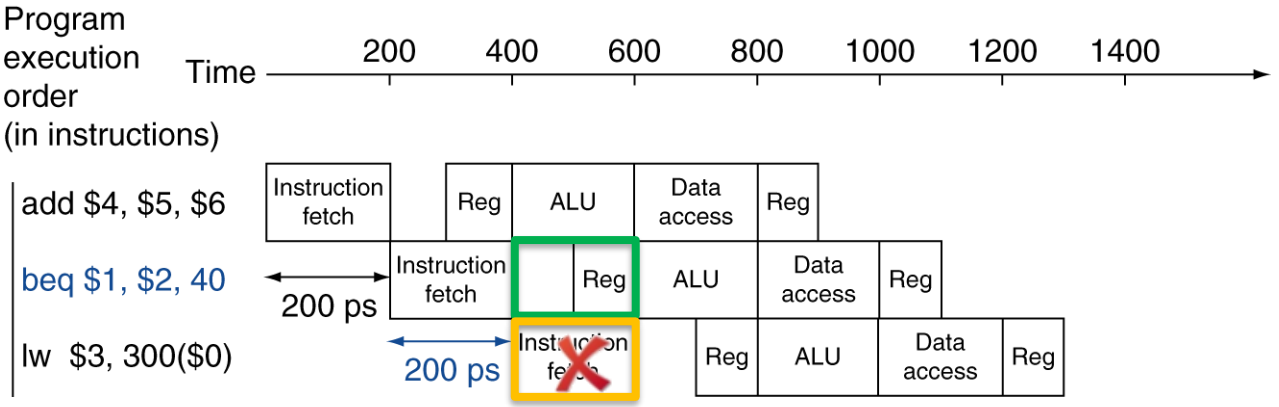
Branch Prediction – Falsche Vorhersage

Im Falle einer falschen Vorhersage wird bei MIPS "nur" eine falsche Instruktion geladen. Die Ausführung der "falschen" Instruktion muss gestoppt werden und stattdessen die richtige Instruktion geladen. Der Verlust ist eine Pipeline-Instruktion. In anderen Pipelines kann der Vorlauf der falschen Instruktion länger sein und dementsprechend ist die Korrektur schwieriger und der Schaden größer.

Richtige Vorhersage



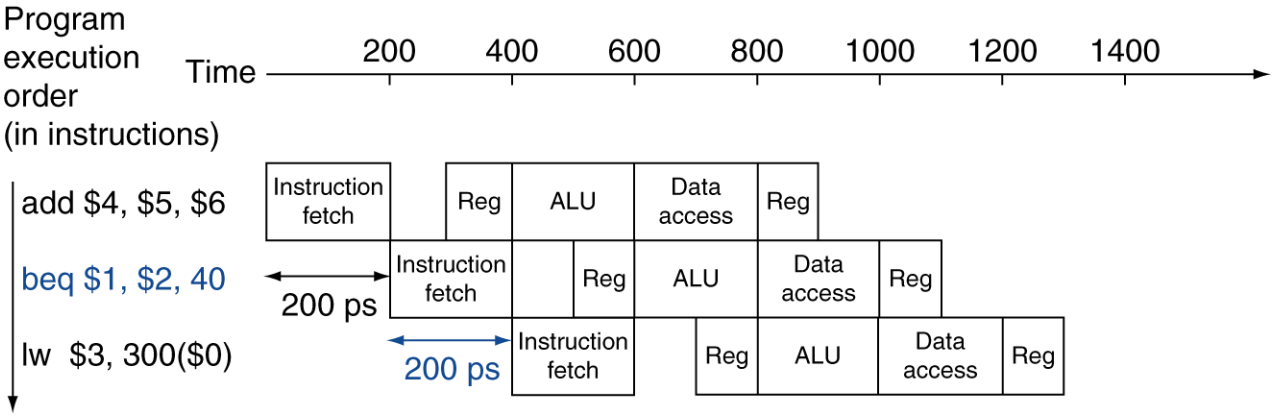
Falsche Vorhersage



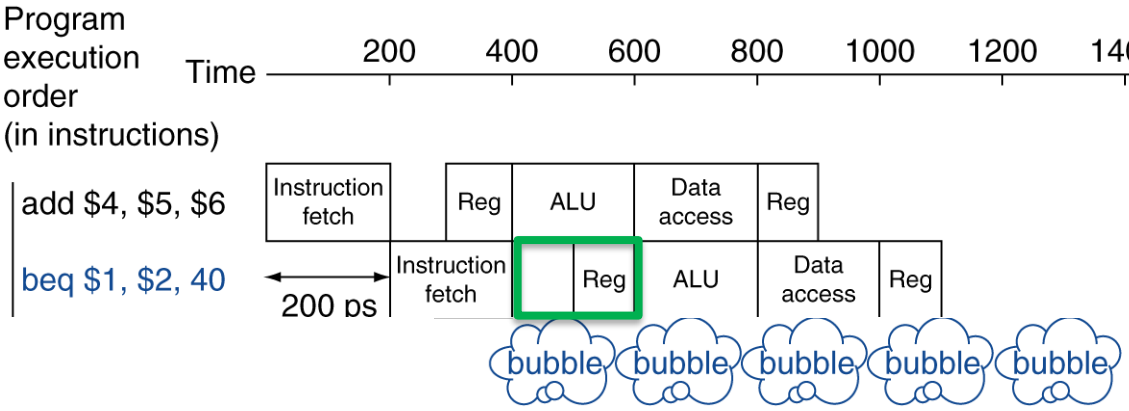
Branch Prediction – Falsche Vorhersage

Im Falle einer falschen Vorhersage wird bei MIPS "nur" eine falsche Instruktion geladen. Die Ausführung der "falschen" Instruktion muss gestoppt werden und stattdessen die richtige Instruktion geladen. Der Verlust ist eine Pipeline-Instruktion. In anderen Pipelines kann der Vorlauf der falschen Instruktion länger sein und dementsprechend ist die Korrektur schwieriger und der Schaden größer.

Richtige Vorhersage



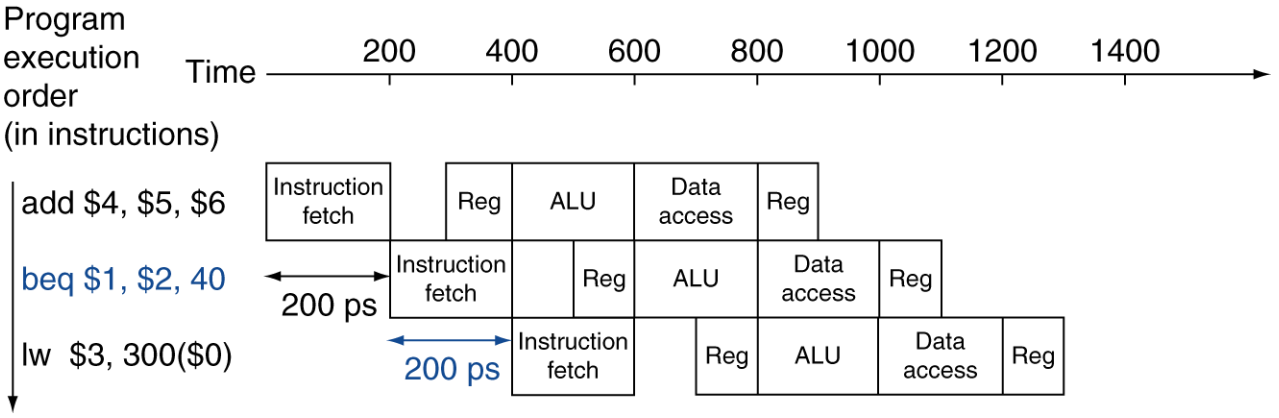
Falsche Vorhersage



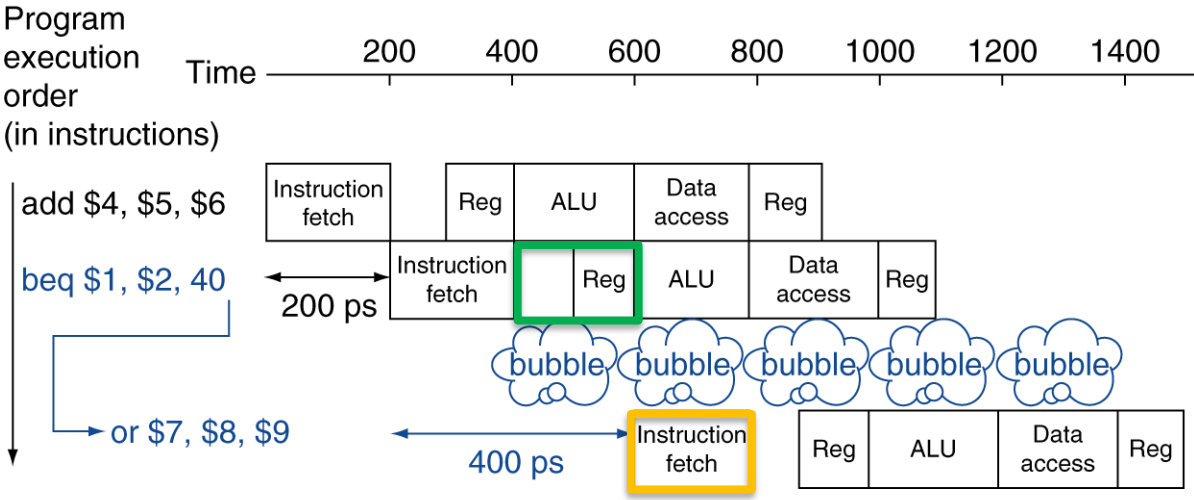
Branch Prediction – Falsche Vorhersage

Im Falle einer falschen Vorhersage wird bei MIPS "nur" eine falsche Instruktion geladen. Die Ausführung der "falschen" Instruktion muss gestoppt werden und stattdessen die richtige Instruktion geladen. Der Verlust ist eine Pipeline-Instruktion. In anderen Pipelines kann der Vorlauf der falschen Instruktion länger sein und dementsprechend ist die Korrektur schwieriger und der Schaden größer.

Richtige Vorhersage



Falsche Vorhersage



Branch Prediction – Wie geht das?

- Statische Vorhersage
  - Grundlage ist übliches Programmverhalten
  - Beispiel: Welche Sprünge sind bei einer Schleife wahrscheinlich?
    - Rückwärtssprünge sind wahrscheinlich
    - Vorwärtssprünge sind unwahrscheinlich
- Dynamische Vorhersage
  - Hardware misst das tatsächliche Verhalten
    - für jeden bedingten Sprung im Programm kann die Hardware eine Statistik erfassen, wie oft gesprungen wird
    - Vorhersage unter der Annahme, dass das zukünftige Verhalten dem Trend entspricht
  - im Fehlerfall:
    - Stalling, um richtige Instruktion zu holen
    - Statistik anpassen

Zusammenfassung: Pipelining

- Pipelining verbessert die Performance im Sinne eines erhöhten Durchsatzes
  - mehrere Instruktionen können parallel ausgeführt werden
  - die Latenz einer einzelnen Instruktion ist mindestens so groß wie ohne Pipelining
- Hazards bereiten Schwierigkeiten
  - Strukturell, Data, Control
  - intelligente Lösungen gefordert, um Unterbrechungen der Pipeline zu minimieren (und Durchsatz zu optimieren)
- Design des Befehlssatzes wirkt sich auf die Komplexität einer Pipeline-Implementierung aus
  - einfacher regelmäßiger Befehlssatz ➔ einfache Pipeline-Implementierung