

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Triple Modular Redundancy

Florian Schmaus, Simon Schuster

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
<https://www4.cs.fau.de>

30. April 2018



- Senior Developer bei QNX
 - *Ten Truths about Building Safe Embedded Software Systems*
 - IEC 61508, IEC 62304/ISO 14971, ISO 26262, EN 5012x
 - Standards sind laufend in Entwicklung
 - Bitflips vs. Rowhammering
 - Safety vs. Security
- 👉 *Need for qualified developers*

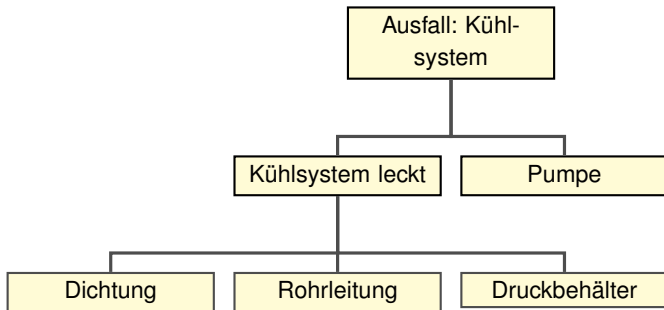


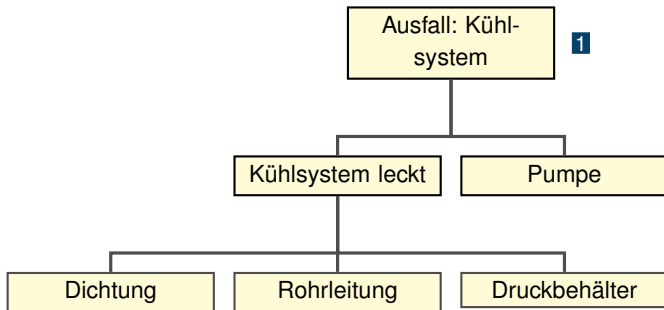
- 1 Wiederholung: Grundlagen Fehlerbäume
- 2 Wiederholung: Triple Modular Redundancy
- 3 Ausblick: Rechnerarchitektur, Replikation und Redundanz
- 4 Replikation von Code



- 1** Wiederholung: Grundlagen Fehlerbäume
- 2 Wiederholung: Triple Modular Redundancy
- 3 Ausblick: Rechnerarchitektur, Replikation und Redundanz
- 4 Replikation von Code

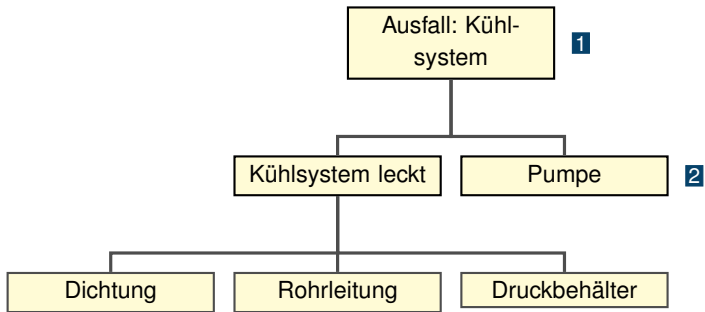






1. Schadensereignis

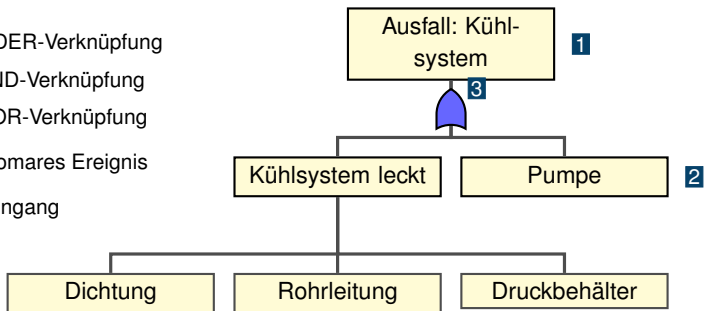
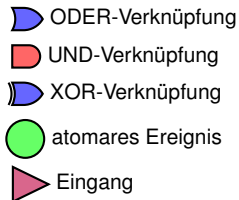




1. Schadensereignis
2. Ereignisse auf Ebene 2



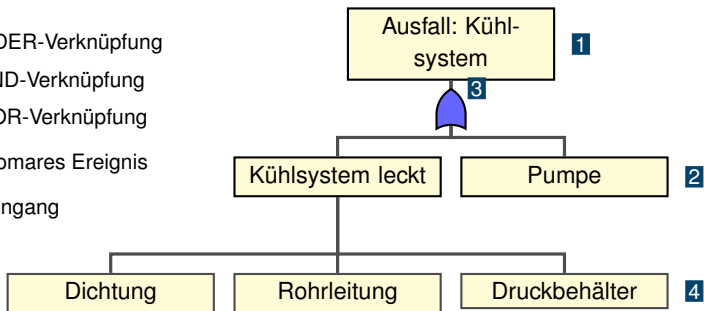
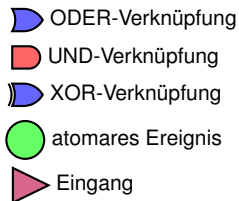
Fehlerbäume – Wiederholung



1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung



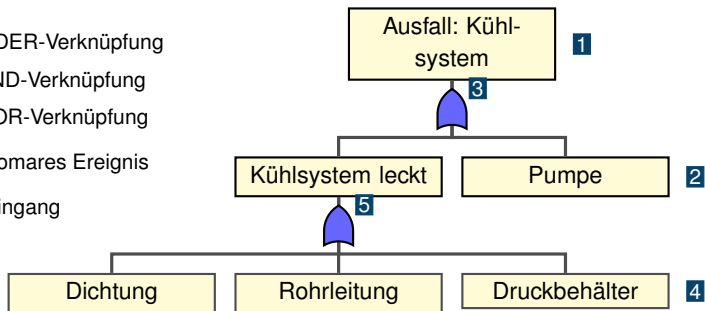
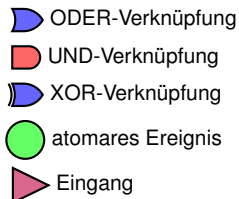
Fehlerbäume – Wiederholung



1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung
4. Ereignisse auf Ebene 3



Fehlerbäume – Wiederholung

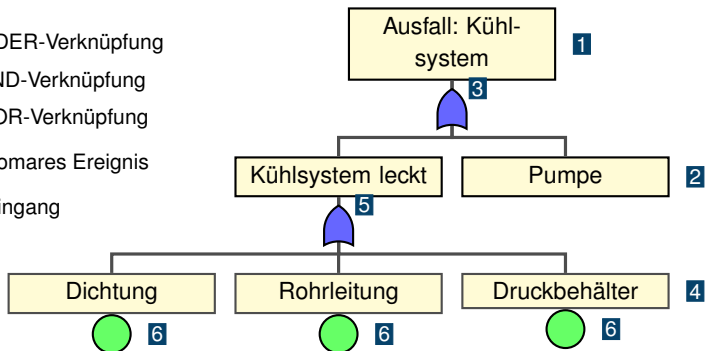
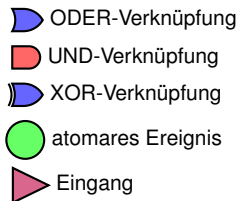


1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung
4. Ereignisse auf Ebene 3

5. Logische Verknüpfung



Fehlerbäume – Wiederholung

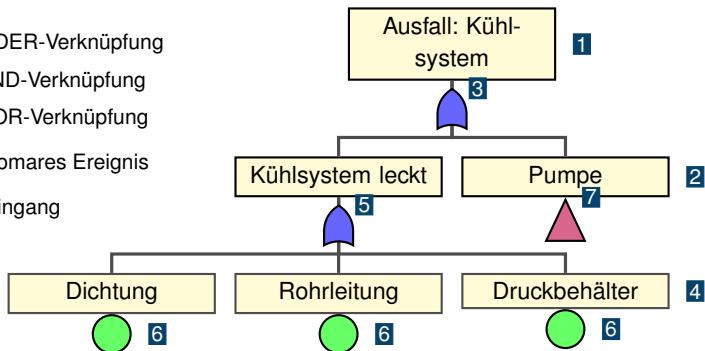
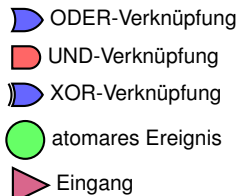


1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung
4. Ereignisse auf Ebene 3

5. Logische Verknüpfung
6. Atomare Ereignisse



Fehlerbäume – Wiederholung

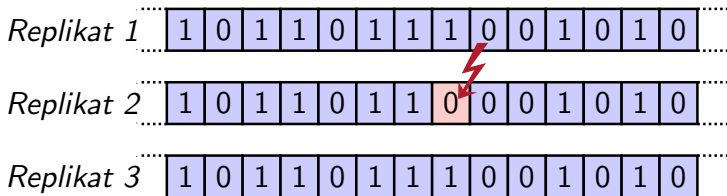


- | | | | |
|----|------------------------|----|---|
| 1. | Schadensereignis | 5. | Logische Verknüpfung |
| 2. | Ereignisse auf Ebene 2 | 6. | Atomare Ereignisse |
| 3. | Logische Verknüpfung | 7. | Eingänge zerlegen den Fehlerbaum → Neuer Teilbaum |
| 4. | Ereignisse auf Ebene 3 | | |



- 1 Wiederholung: Grundlagen Fehlerbäume
- 2 Wiederholung: Triple Modular Redundancy**
- 3 Ausblick: Rechnerarchitektur, Replikation und Redundanz
- 4 Replikation von Code

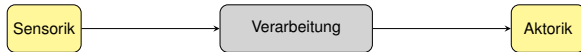


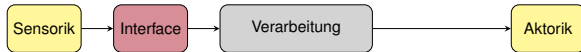


- **Wie viele Replikate** benötigt man?
- Arten des Fehlverhaltens (von n Replikaten sind f fehlerhaft)
 1. fail-silent $\rightarrow$ Anzahl Replikate: $n = f + 1$
 2. fail-consistent $\rightarrow$ Anzahl Replikate: $n = 2f + 1$
 3. malicious $\rightarrow$ Anzahl Replikate: $n = 3f + 1$



Triple Modular Redundancy

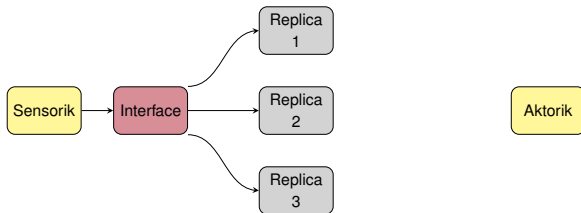




- Schnittstelle sammelt Eingangsdaten (Replikdeterminismus)



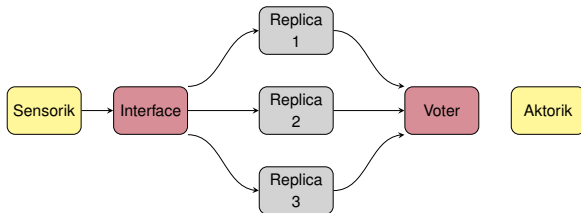
Triple Modular Redundancy



- Schnittstelle sammelt Eingangsdaten (Replikdeterminismus)
- Verteilt Daten und aktiviert Replikate



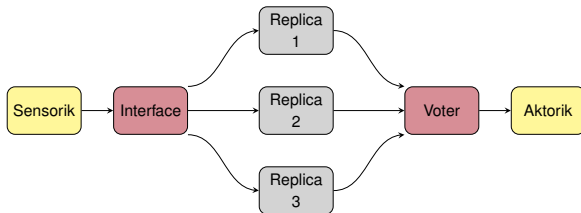
Triple Modular Redundancy



- Schnittstelle sammelt Eingangsdaten (Replikdeterminismus)
- Verteilt Daten und aktiviert Replikate
- Mehrheitsentscheider (Voter) wählt Ergebnis



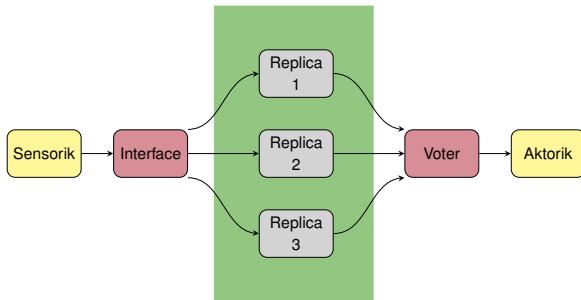
Triple Modular Redundancy



- Schnittstelle sammelt Eingangsdaten (Replikdeterminismus)
- Verteilt Daten und aktiviert Replikate
- Mehrheitsentscheider (Voter) wählt Ergebnis
- Ergebnis wird an Aktuator versendet



Triple Modular Redundancy

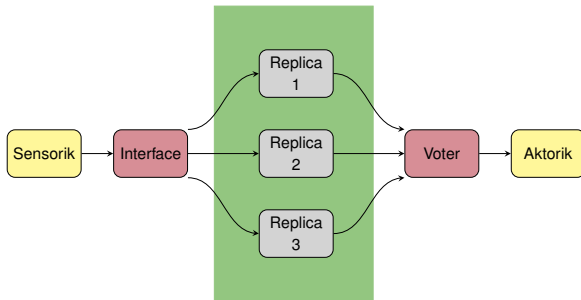


Redundanzbereich

Ausschließlich Replikatausführung



Triple Modular Redundancy



Redundanzbereich

Ausschließlich Replikatausführung

- Erweiterung der Ausgangsseite mit Informationsredundanz
- Mehrheitsentscheid über Berechnungsergebnisse



Replikat 1

```
void repl_2(void* p){  
    ticks_t time =  
        ezs_get_time();  
  
    ...  
}
```

Replikat 2

```
void repl_2(void* p){  
    ticks_t time =  
        ezs_get_time();  
  
    ...  
}
```

Replikat 3

```
void repl_3(void* p){  
    ticks_t time =  
        ezs_get_time();  
  
    ...  
}
```

Sicherstellung Replikdeterminismus

- Globale diskrete Zeitbasis
- Einigung über Eingabewerte
- Statische Kontrollstruktur der Replikate
- Deterministische Algorithmen

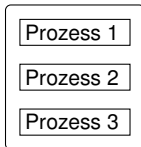
☞ Sicherstellung, dass Replikat *innerhalb Zeitspanne* Ergebnis liefert



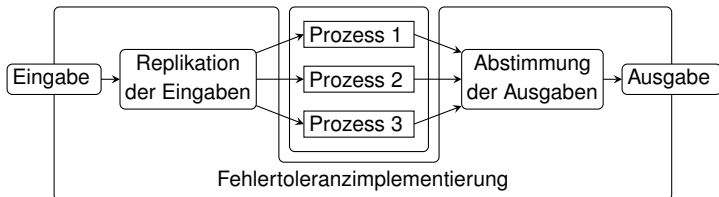
Process-Level Redundancy



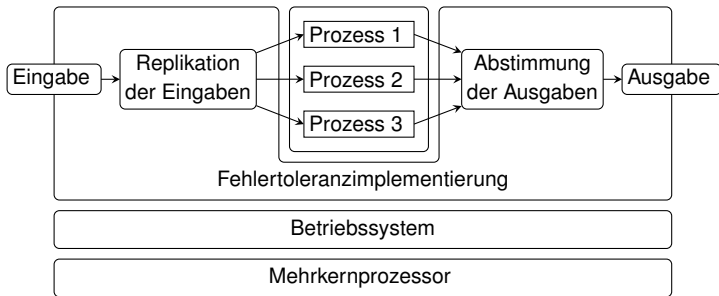
Process-Level Redundancy



Process-Level Redundancy



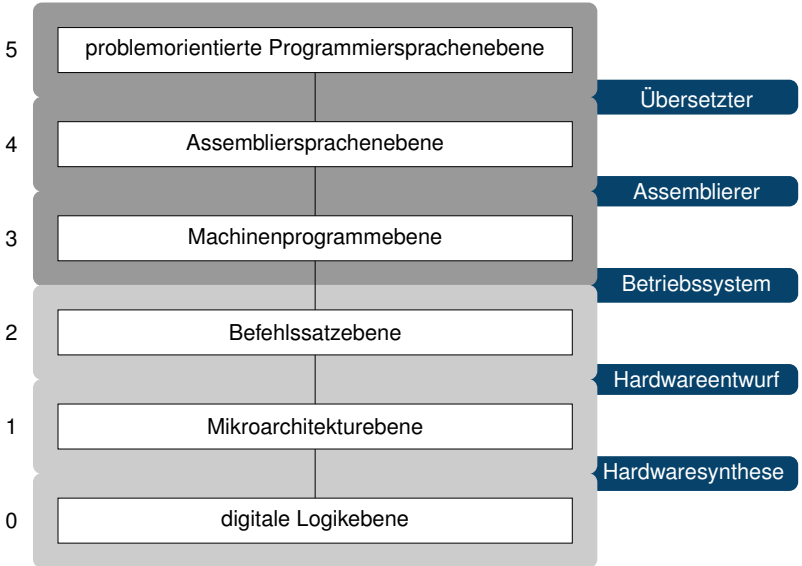
Process-Level Redundancy



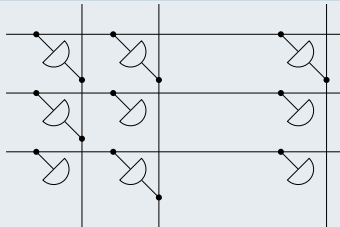
- 1 Wiederholung: Grundlagen Fehlerbäume
- 2 Wiederholung: Triple Modular Redundancy
- 3 Ausblick: Rechnerarchitektur, Replikation und Redundanz**
- 4 Replikation von Code



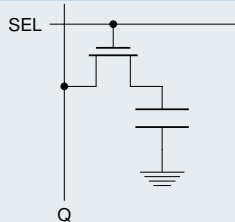
Ebenen



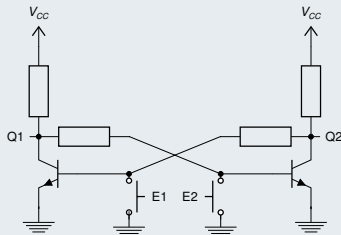
ROM



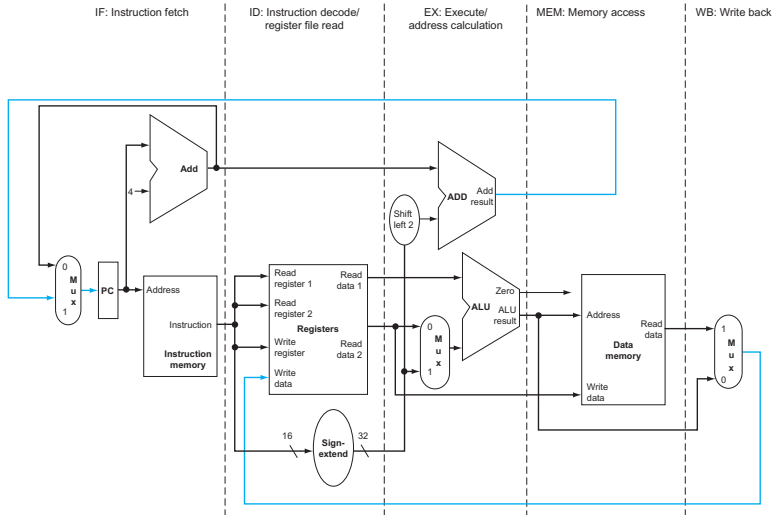
DRAM



RS - FlipFlop

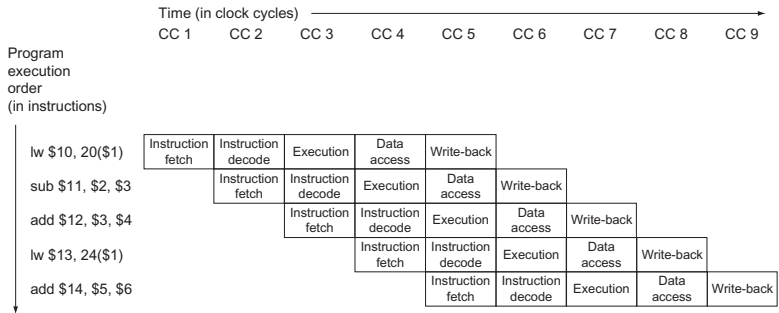


MIPS: Single-Cycle



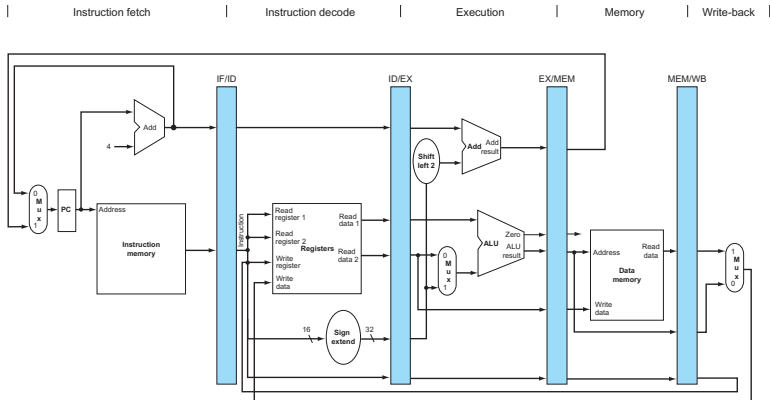
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



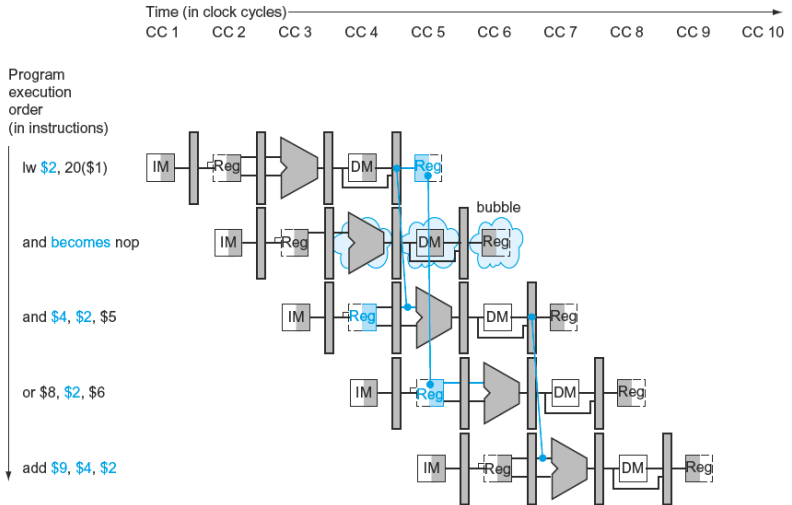
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



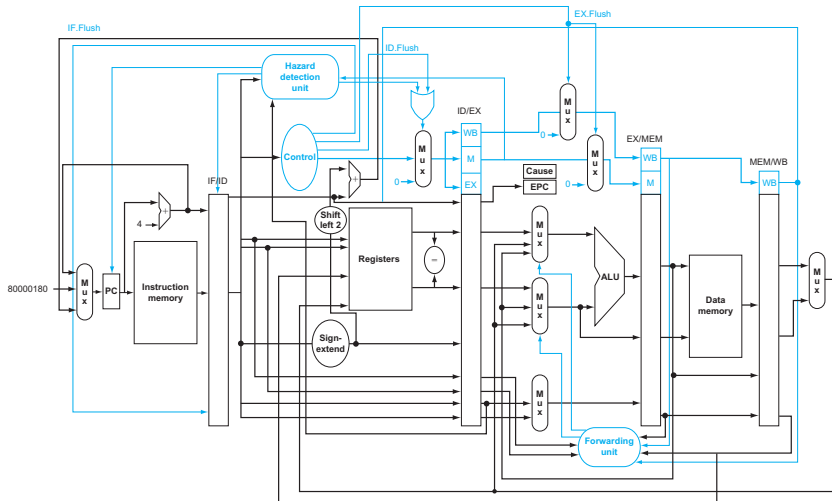
Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

MIPS: Pipelining



Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

- Mikroprogrammierbar vs. Fixed-Function
- Out-of-Order-Prozessoren
- Sprungvorhersage
- Transaktionaler Speicher
- Superskalarität
- Mehrkernarchitekturen
- Hyperthreading
- ...



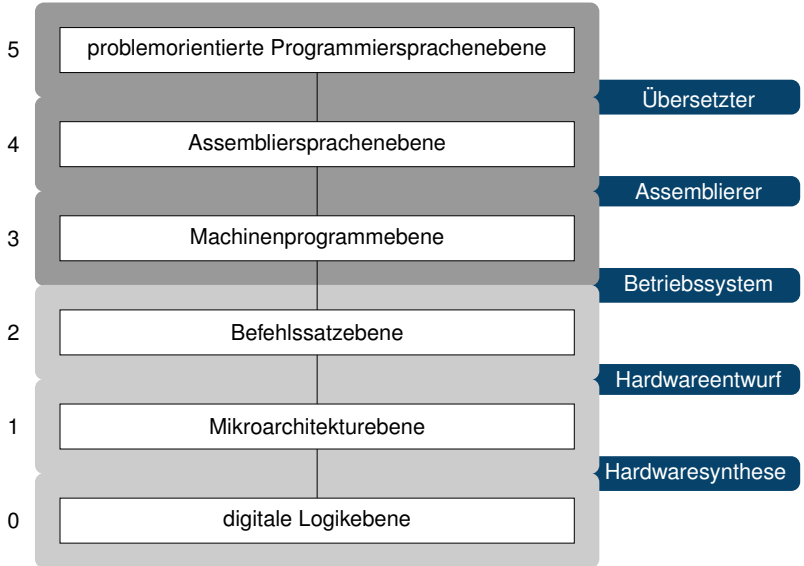
- Mikroprogrammierbar vs. Fixed-Function
- Out-of-Order-Prozessoren
- Sprungvorhersage
- Transaktionaler Speicher
- Superskalarität
- Mehrkernarchitekturen
- Hyperthreading
- ...

☞ All diese zusätzlichen Fehlerpunkte müssen im Fehlermodell berücksichtigt werden

☞ Ein Ein-Bit-Fehler in einer dieser Komponenten kann zu komplexen Mehrbitfehlern auf ISA-Ebene führen



Ebenen



Speicherorganisation auf einem Mikrocontroller

```
int a;                // a: global, uninitialized
int b = 1;            // b: global, initialized
const int c = 2;      // c: global, const

void main() {
    static int s = 3;  // s: local, static, initialized
    int x, y;          // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm

Symbol Table	<a>
.data	s=3 b=1
.rodata	c=2
.text	main
...	
ELF Header	

ELF-Binary



Speicherorganisation auf einem Mikrocontroller

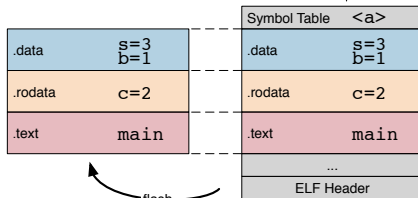
```
int a;           // a: global, uninitialized
int b = 1;       // b: global, initialized
const int c = 2; // c: global, const

void main() {
    static int s = 3; // s: local, static, initialized
    int x, y;         // x: local, auto; y: local, auto
    char* p = malloc( 100 ); // p: local, auto; *p: heap (100 byte)
}
```

compile / link

Quellprogramm

Flash / ROM

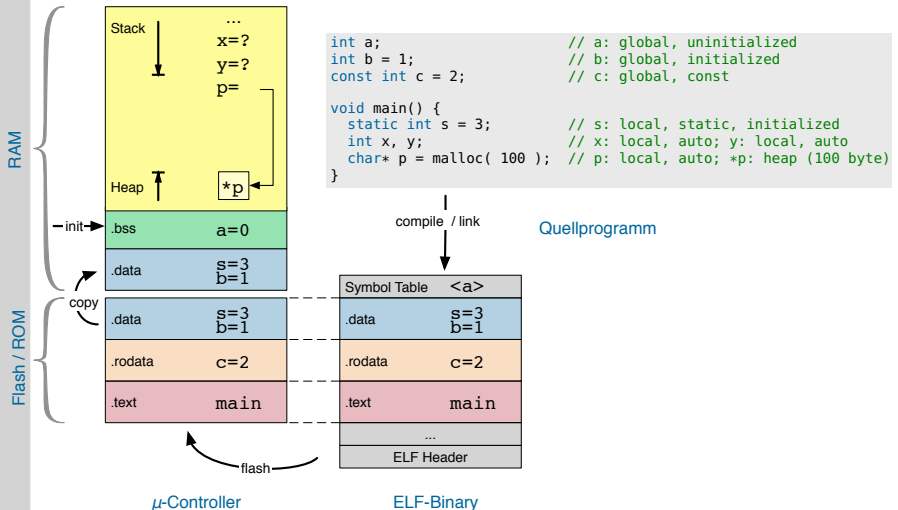


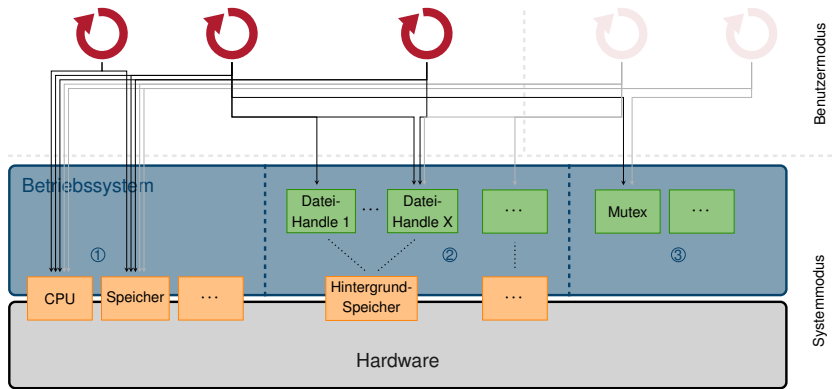
µ-Controller

ELF-Binary

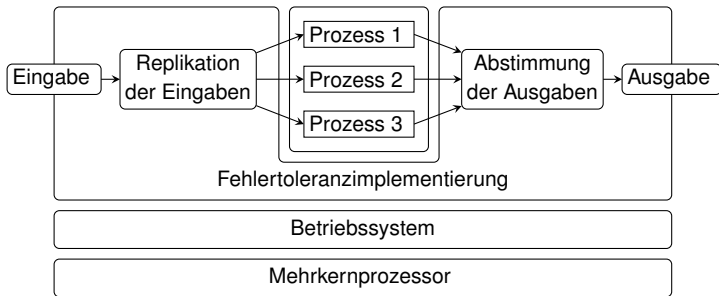


Speicherorganisation auf einem Mikrocontroller





Process-Level Redundancy



C-Code vs. Assembler-Code

C-Code

```
int a;  
int b = 1;  
const int c = 2;  
void main() {  
    static int s = 3;  
    int x, y;  
    char* p = malloc( 100 );  
}
```

Assembler-Code

```
4004f0 <main>:  
4004f0: push    %rbp  
4004f1: mov     %rsp,%rbp  
4004f4: sub     $0x10,%rsp  
4004f8: movabs  $0x64,%rdi  
4004ff: 00 00 00  
400502: callq   4003e0 <malloc@plt>  
400507: mov     %rax,-0x10(%rbp)  
40050b: add     $0x10,%rsp  
40050f: pop     %rbp  
400510: retq
```

Wo können Datenfehler auftreten?



C-Code vs. Assembler-Code

C-Code

```
int a;  
int b = 1;  
const int c = 2;  
void main() {  
    static int s = 3;  
    int x, y;  
    char* p = malloc( 100 );  
}
```

Assembler-Code

```
4004f0 <main>:  
4004f0: push    %rbp  
4004f1: mov     %rsp,%rbp  
4004f4: sub     $0x10,%rsp  
4004f8: movabs  $0x64,%rdi  
4004ff: 00 00 00  
400502: callq   4003e0 <malloc@plt>  
400507: mov     %rax,-0x10(%rbp)  
40050b: add     $0x10,%rsp  
40050f: pop     %rbp  
400510: retq
```

Wo können Datenfehler auftreten?

1. RAM: $-0x10(\%rbp)$
2. Allgemeine CPU-Register: $\%rsp$
3. Sonstige CPU-Register: $\%rip, \%rflags$



- 1 Wiederholung: Grundlagen Fehlerbäume
- 2 Wiederholung: Triple Modular Redundancy
- 3 Ausblick: Rechnerarchitektur, Replikation und Redundanz
- 4 Replikation von Code**



Stringification von CPP

```
1 #define CMP_FUNC(pre, repl, type, op) \  
2     type pre##repl(type a, type b) { \  
3         return a op b ? a : b; \  
4     } \  
5 \  
6 CMP_FUNC(max, 1, int, >); // Funktion ? \  
7 CMP_FUNC(max, 2, int, >); // Funktion ? \  
8 ... \  
9 CMP_FUNC(min, 1, int, <); // Funktion ?
```

- Verwendung des C-Präprozessors (CPP)
- ##: „Token Pasting Operator“
- Konkatenieren zweier Token zu einem
- Aufruf & Deklaration müssen erstellt werden
- ☞ Es geht eleganter ...



Stringification von CPP

```
1 #define CMP_FUNC(pre, repl, type, op) \  
2     type pre##repl(type a, type b) { \  
3         return a op b ? a : b; \  
4     } \  
5 \  
6 CMP_FUNC(max, 1, int, >); // Funktion max1 \  
7 CMP_FUNC(max, 2, int, >); // Funktion max2 \  
8 ... \  
9 CMP_FUNC(min, 1, int, <); // Funktion min1
```

- Verwendung des C-Präprozessors (CPP)
 - ##: „Token Pasting Operator“
 - Konkatenieren zweier Token zu einem
 - Aufruf & Deklaration müssen erstellt werden
- ☞ Es geht eleganter ...



C++ Template

```
1  template <typename T>
2  T max(T x, T y) {
3      T value;
4      if (x < y)
5          value = y;
6      else
7          value = x;
8      return value;
9  }
10 ...
11 double md = max<double>(2.3, 4.2);
12 auto    mi = max<int>(230, 42);
```

- Templates ermöglichen generische Programmierung
- Wiederverwendung durch **Parametrisierung**
- Unterscheidung von Funktions- & Klassen-Templates
- Expansion zur Compilezeit $\leadsto$ Quelltext muss verfügbar sein (im Header)
- „Code Bloat“ beim Compilieren $\rightarrow$ **nutzbar für Replikation von Code**



C++ Template Spezialisierung

```
1 template <float T>
2 T my_func(T x, T y) { // specialized template for T == float
3     T a = x;
4     ...
5 }
```

- Spezialisierungen von Templates möglich
 - Effizientere Implementierung für bestimmte Typen
- Nutzbar zum „Zählen“ von Templates

Mehrere Template-Parameter

```
1 template <typename A, typename B, typename C>
2 T my_other_func(A x, B y, C z) {
3     ...
4 }
5 my_other_func<Dog, Cat, Mouse>(m, n, o);
```



Symboltabellen in ELF-Dateien

```
1 0000000000201028 B __bss_start
2 0000000000201028 b completed.6983
3 ...
4 000000000000061a T main
5 0000000000000580 t register_tm_clones
6 0000000000000510 T _start
7 ...
8 000000000000066d t int inc<0>(int)
9 000000000000067c t int inc<1>(int)
10 000000000000068b t int inc<2>(int)
```

- nm: Ausgabe der Symboltabelle
- Nützliche Optionen
 - -C: Dekodieren der C++-Namensmangelung:
_Z3incILi0EEii ⇒ `int inc<0>(int)`
_Z3incILi1EEii ⇒ `int inc<1>(int)`



Assembly-Code gemischt mit Source-Code

```
1 int main(){
2     4007cd: 55                push    %rbp
3     4007ce: 48 89 e5          mov     %rsp,%rbp
4     4007d1: 48 83 ec 10       sub     $0x10,%rsp
5     int a = max<int>(23U, 42);
6     4007d5: be 2a 00 00 00    mov     $0x2a,%esi
7     4007da: bf 17 00 00 00    mov     $0x17,%edi
8     4007df: e8 04 01 00 00    callq   4008e8 <_Z3maxIiET_S0_S0_>
9     4007e4: 89 45 fc          mov     %eax,-0x4(%rbp)
10    std::cout << a << "\n";
11    4007e7: 8b 45 fc          mov     -0x4(%rbp),%eax
12    ...
```

■ objdump: Ausgabe von Informationen von Objektdateien

■ Nützliche Optionen

- -S: Ausgabe von Quell-Code im Assembly-Code (Debug-Symbole notwendig)
- -D: alle Sektionen disassemblieren



C Linkage

```
1 // C++ code
2 extern "C" void f(int); // one way
3 extern "C" {           // another way
4     int g(double);
5     double h();
6 };
7 void code(int i, double d)
8 {
9     f(i);
10    int ii = g(d);
11    double dd = h();
12    // ...
13 }
```

- Name mangling von C++ verhindern
⇒ C++-Code aus C-Code aufrufen



42

