# Verlässliche Echtzeitsysteme

## Übungen zur Vorlesung

Phillip Raffeck, Florian Schmaus, Simon Schuster

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

Sommersemester 2019

# Keynote: Challenges in Safety-Critical Software Chris Hobbs

- 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*

# Überblick


1 Wiederholung: Grundlagen Fehlerbäume

2 Wiederholung: Triple Modular Redundancy

3 Ausblick: Rechnerarchitektur, Replikation und Redundanz

4 Replikation von Code

# Überblick

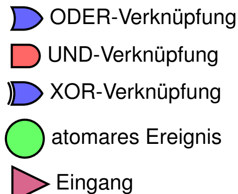
1 Wiederholung: Grundlagen Fehlerbäume

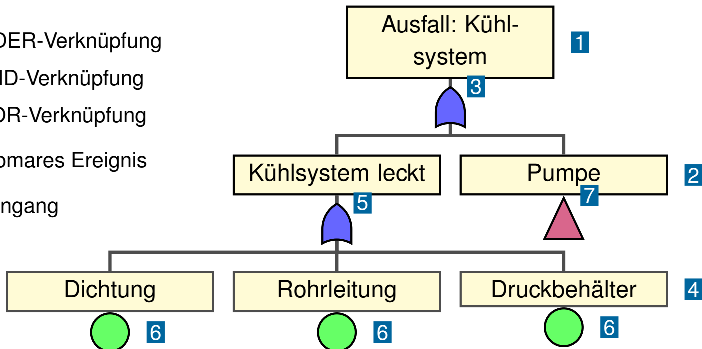
2 Wiederholung: Triple Modular Redundancy

3 Ausblick: Rechnerarchitektur, Replikation und Redundanz

4 Replikation von Code

# Fehlerbäume – Wiederholung




1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung
4. Ereignisse auf Ebene 3
5. Logische Verknüpfung
6. Atomare Ereignisse
7. Eingänge zerlegen den Fehlerbaum $\mapsto$ Neuer Teilbaum

# Überblick

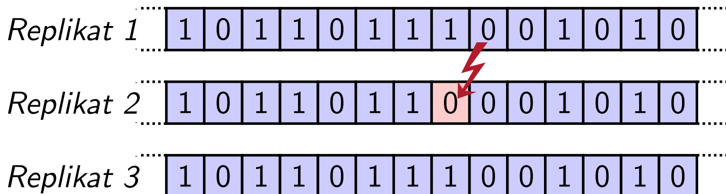
1 Wiederholung: Grundlagen Fehlerbäume

2 Wiederholung: Triple Modular Redundancy

3 Ausblick: Rechnerarchitektur, Replikation und Redundanz

4 Replikation von Code

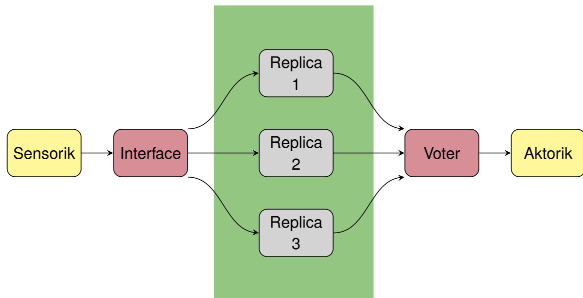
# Fehlerhypothese




- **Wie viele Replikate** benötigt man?
- Arten des Fehlverhaltens (von $n$ Replikaten sind $f$ fehlerhaft)

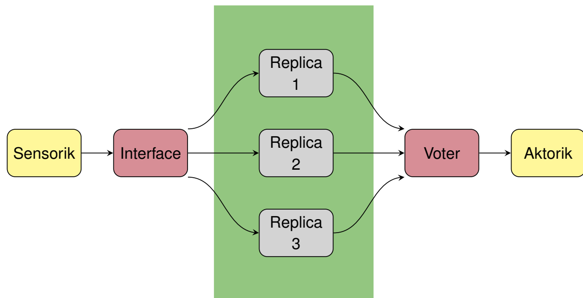
1. fail-silent $\mapsto$ Anzahl Replikate: $n = f + 1$
2. fail-consistent $\mapsto$ Anzahl Replikate: $n = 2f + 1$
3. malicious $\mapsto$ Anzahl Replikate: $n = 3f + 1$

# 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

- Erweiterung der Ausgangsseite mit Informationsredundanz
- Mehrheitsentscheid über Berechnungsergebnisse

# Replikdeterminismus

### Replikat 1

```
void repl_1(void *p){
  ticks_t time =
    ezs_get_time();

  ...
}
```

### Replikat 2

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

  ...
}
```

### Replikat 3

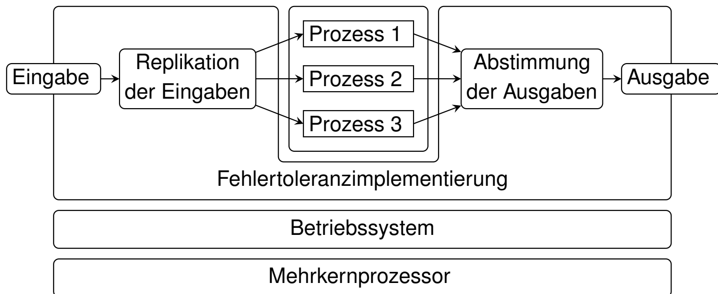
```
void repl_3(void *p){
  ticks_t time =
    ezs_get_time();

  ...
}
```

### Sicherstellung Replikdeterminismus

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

☞ Sicherstellung, dass Replikat *innerhalb Zeitspanne* Ergebnis liefert

# Process-Level Redundancy



# Überblick

1 Wiederholung: Grundlagen Fehlerbäume

2 Wiederholung: Triple Modular Redundancy

3 Ausblick: Rechnerarchitektur, Replikation und Redundanz

4 Replikation von Code

# Ebenen

| | Ebene |
|---|---|
| 5 | problemorientierte Programmiersprachenebene |
| 4 | Assembliersprachenebene |
| 3 | Machinenprogrammebene |
| 2 | Befehlssatzebene |
| 1 | Mikroarchitekturebene |
| 0 | digitale Logikebene |

Übersetzer

Assemblierer

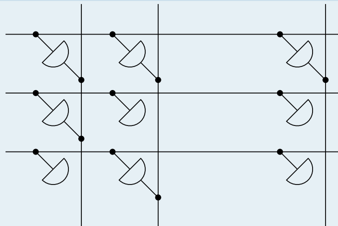
Betriebssystem

Hardwareentwurf

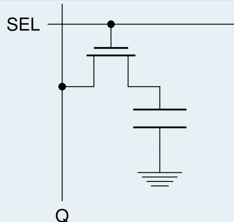
Hardwaresynthese

# Digitale Logikebene

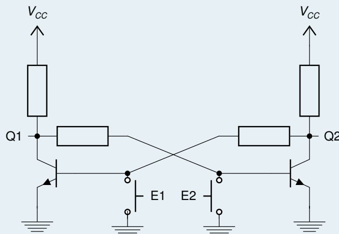
## ROM



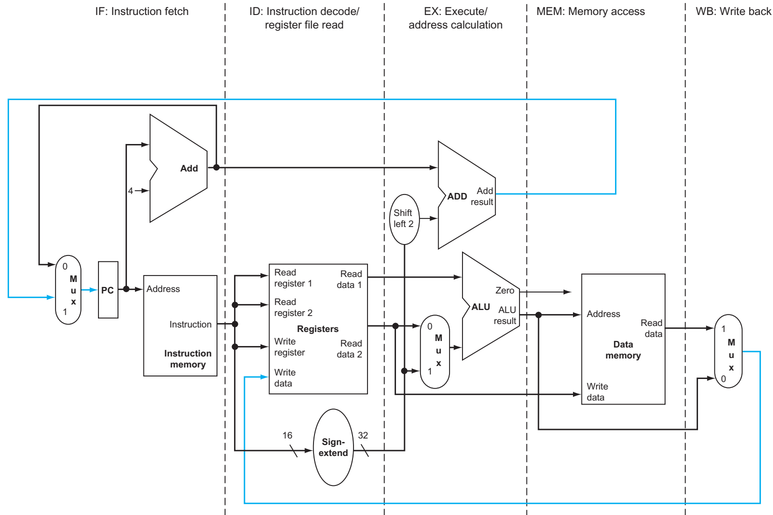
## DRAM




## RS - FlipFlop

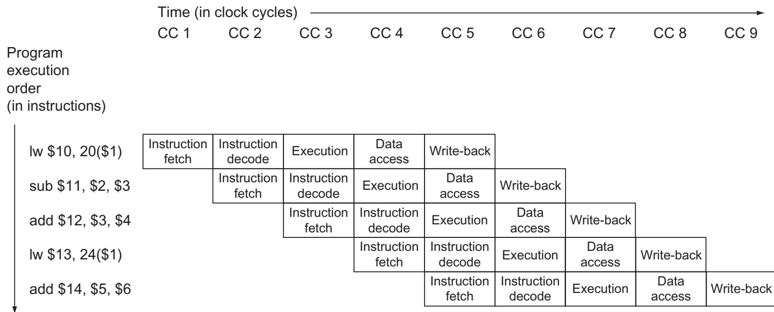


# MIPS: Single-Cycle



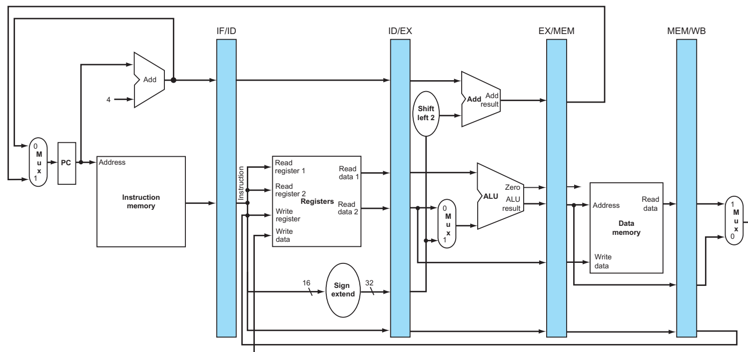

Source: D. A. Patterson und J. L. Hennessy, Computer organization and design: the hardware/software interface, 4th ed., 2012

# 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

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9 CC 10

Program execution order (in instructions)

lw $2, 20($1) — IM, Reg, DM, Reg

and becomes nop — IM, Reg, DM, Reg (bubble)

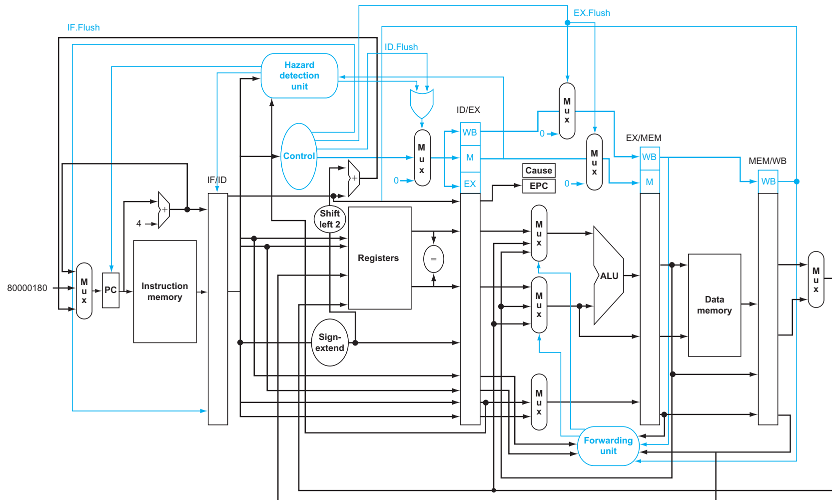
and $4, $2, $5 — IM, Reg, DM, Reg

or $8, $2, $6 — IM, Reg, DM, Reg

add $9, $4, $2 — IM, Reg, DM, Reg

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

# Eigenschaften von CPU-Architekturen

- 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

| | |
|---|---|
| 5 | problemorientierte Programmiersprachenebene |
| 4 | Assembliersprachenebene |
| 3 | Machinenprogrammebene |
| 2 | Befehlssatzebene |
| 1 | Mikroarchitekturebene |
| 0 | digitale Logikebene |

Übersetzer

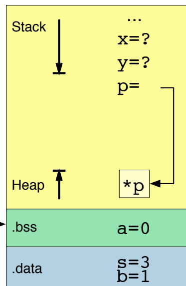
Assemblierer

Betriebssystem

Hardwareentwurf

Hardwaresynthese

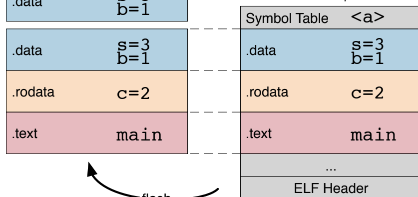
# Speicherorganisation auf einem Mikrocontroller




```c
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




$\mu$-Controller

ELF-Binary

# Betriebssystem

Benutzermodus

Betriebssystem

Datei-Handle 1 · · · Datei-Handle X · · · Mutex · · ·

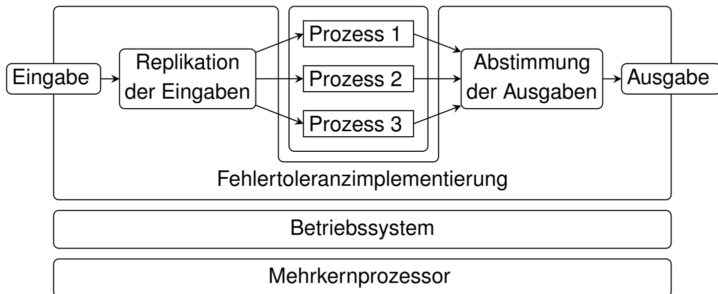
① ② ③

CPU Speicher · · · Hintergrund-Speicher · · ·

Hardware

Systemmodus

# 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
  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`

# Überblick

1 Wiederholung: Grundlagen Fehlerbäume

2 Wiederholung: Triple Modular Redundancy

3 Ausblick: Rechnerarchitektur, Replikation und Redundanz

4 Replikation von Code

# Code-Replikation: Stringification

## 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++ Templates

## 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>(23U, 42);
```

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

# C++ Templates

- Explizite Typen als Templateparameter möglich
- Nutzbar zum „Zählen” von Templates

## Indizierte Template-Spezialisierung

```
template <typename T, unsigned INDEX>
T max(T x, T y) {
  T value;
  if (x < y)
    value = y;
  else
    value = x;
  return value;
}
...
auto m0 = max<int, 0>(23, 42);
auto m1 = max<int, 1>(23, 42);
```

# Umgang mit Assembly-Code I

## 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 max<int,0u>(int,int)
9  000000000000067c t int max<int,1u>(int,int)
```

- nm: Ausgabe der Symboltabelle
- Nützliche Optionen
  - -C: Dekodieren der C++-Namensmangelung:
    `_Z3maxIiLj0EET_S0_S0_` ⇒ `int max<int, 0u>(int, int)`
    `_Z3maxIiLj1EET_S0_S0_` ⇒ `int max<int, 1u>(int, int)`
- Demangling auf der Kommandozeile: `c++filt`

# Umgang mit Assembly-Code II

## Assembly-Code gemischt mit Source-Code

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

- `objdump`: Ausgabe von Informationen von Objektdateien
- Nützliche Optionen
  - `-S`: Ausgabe von Quell-Code im Assembly-Code (Debug-Symbole notwendig)
  - `-D`: alle Sektionen disassemblieren

# C-Code in C++ Einbinden

## C Linkage

```
// C++ code
extern "C" void f(int); // one way
extern "C" {            // another way
  int g(double);
  double h();
};
void code(int i, double d)
{
  f(i);
  int ii = g(d);
  double dd = h();
  // ...
}
```

- Name mangling von C++ verhindern

⇒ C++-Code aus C-Code aufrufen