TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

TRACE-BASIERTE VERIFIKATION DER FPGA-IMPLEMENTIERUNG EINES MIPS-PROZESSORS

Verteidigung der Studienarbeit

Valentin Gehrke
IST, Jg. 2011, Mat.Nr. 3750176

Betreuer:
Dr.-Ing. Martin Zabel

Betreuender Hochschullehrer:
Prof. Dr.-Ing. habil. Rainer G. Spallek

Dresden, 22.06.2017

Inhalt

- Einleitung
- RUBICS und Trace
- Testprogramme
- Trace-Vergleich
- Ergebnisse
- Zusammenfassung

1 Einleitung

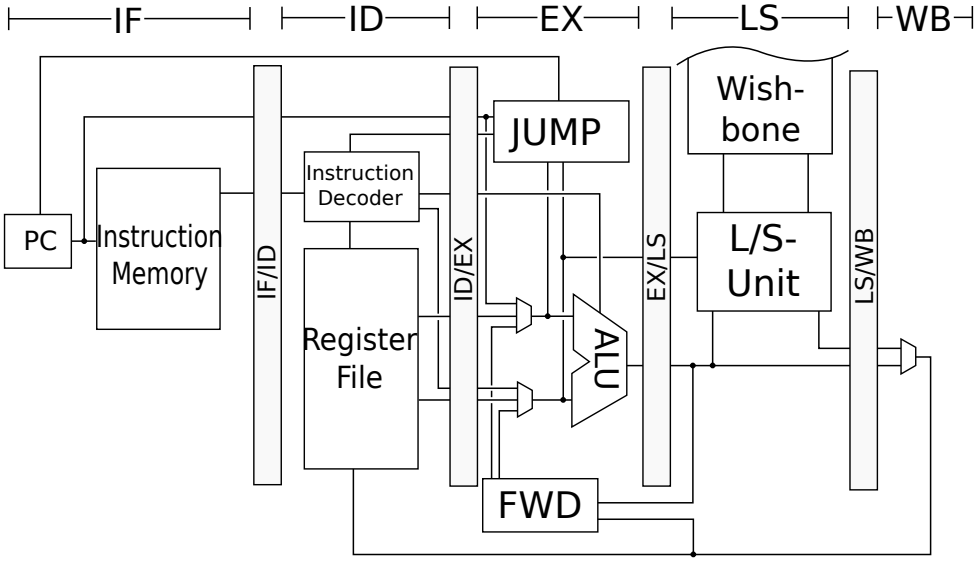
Thema

Trace-basierte Verifikation der FPGA-Implementierung eines MIPS-Prozessors

- Funktionale Verifikation eines MIPS-Prozessors
- VHDL-Implementierung auf FPGA
- Vergleich von Traces
- Befehlssatzsimulator RUBICS
- Selbes Testprogramm in beiden Prozessoren
- Beheben etwaiger Fehler im VHDL-Modell
- **Frage:** Welche Vor- und Nachteile bietet der Trace-Vergleich gegenüber der RTL-Simulation?

MIPS-Prozessor

- 32-Bit-Architektur
- 36 Befehle
- 32 Register,
  Reg. 0 konstant 0
- 5-stufige Pipeline
- Forwarding Unit reduziert Pipeline-Hazards
- 1 Branch-Delay-Slot



Literaturstudium

"Since all single-processor variants of an instruction set architecture must guarantee a unique correct execution flow of a program, their architectural traces must be identical."

- Mammo et al., „Architectural Trace-Based Functional Coverage for Multiprocessor Verification"

⇒ Unterschiede in den Traces weisen auf Fehler hin.

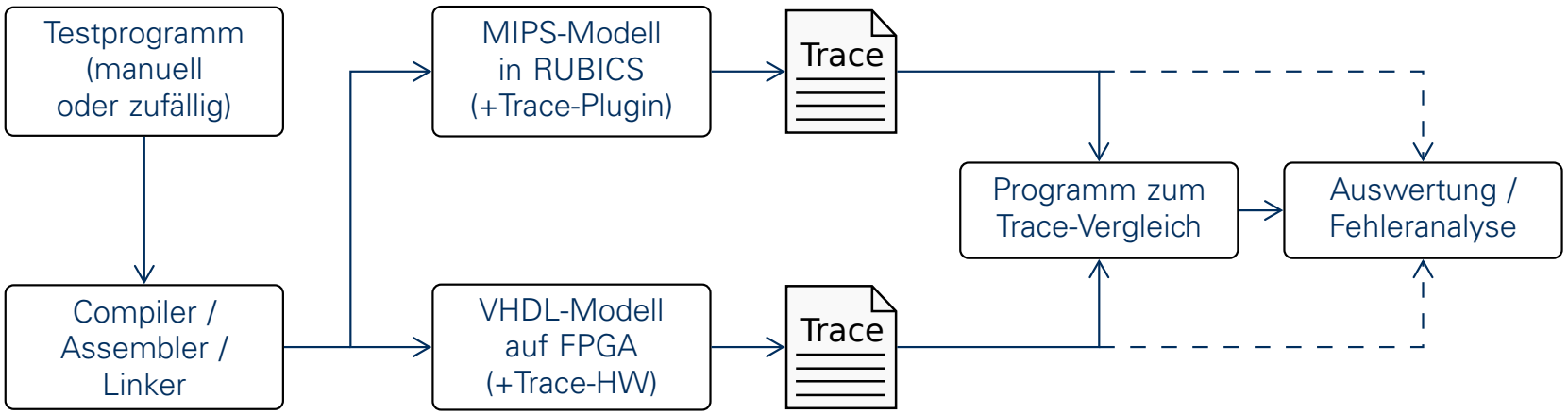
Literaturstudium

" The consept [sic] of directed random input together with high level models proved to be [a] very valuable and efficient way of finding errors from the design."

- Kasanko und Nurmi, „Functional Verification of a 32-bit RISC Processor Core"

⇒ Zufällige Eingabe/Testprogramme helfen zusätzlich

Aufbau

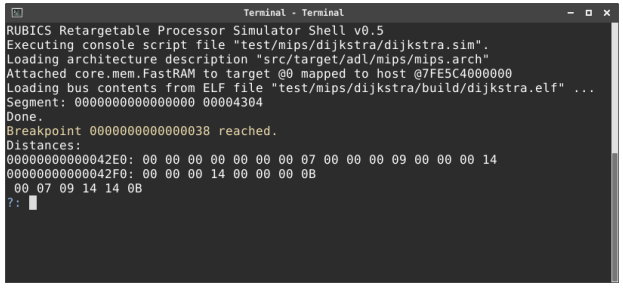

Testprogramm (manuell oder zufällig)
MIPS-Modell in RUBICS (+Trace-Plugin)
Trace
Programm zum Trace-Vergleich
Auswertung / Fehleranalyse
Compiler / Assembler / Linker
VHDL-Modell auf FPGA (+Trace-HW)
Trace

2 RUBICS und Trace

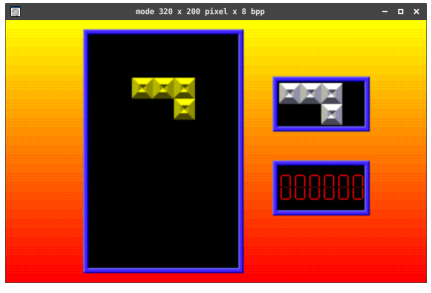
RUBICS

- **R**etargetable **U**niversal **B**inary **I**nstruction **C**onversion **S**imulator
- Befehlssatzsimulator
- Ziel-Architektur wird definiert in **A**rchitecture **D**escription **L**anguage
- konvertiert Teile der ausführbaren Datei nach .NET-Bytecode.

  → Performance-Boost
- Steuerung des Simulators über Kommandozeilen-Interface
- Erweiterbar mittels Plugins in jeder .NET-Sprache



Rubics Kommandozeile



Erweiterung: grafische Ausgabe

MIPS Befehlssatz in RUBICS

- Basierend auf Beschreibung eines DLX-Prozessors
  - 20+2 Befehle übernommen
  - 4+2 Befehle angepasst
  - 12+4 Befehle hinzugefügt
- Verifikation
  - vorgegebene Testprogramme
  - eigene Testprogramme
  - manuelle Auswertung der Ergebnisse
- Verarbeitung in ADL
  - Dekodieren der Instruktion
  - Abbilden des Befehls in mehrere Einzelschritte
  - Verarbeiten der Einzelschritte

MIPS Befehlssatz in RUBICS

```
decoder {
  default = mips;
  operation mips {
    iw = fetch.word; // fetch instruction
    decode iw[26 to 31] { // opcode
      0x08: {is; ls1r; ADDI; sdr;}
      0x23: {is; ls1r; LW; sdr;}
      0x2B: {is; ls1r; s2=d; ls2r; SW;}
      ...
    }
  }
}
behavior {
  operation ADDI {d = s1 + imm;}
  operation SW   {mem.word[s1+imm] = s2;}
  operation LW   {d = mem.word[s1+imm];}
  ...
}
```

MIPS Befehlssatz in RUBICS

22 Befehle waren vorhanden, korrekt implementiert	6 Befehle waren vorhanden, Implementierung angepasst	16 Befehle war nicht vorhanden, wurde implementiert
ADD, ADDI, ADDIU, ADDU, AND, ANDI, BEQ, BNE, J, JAL, JR, LUI, LW, OR, ORI, SUB, SUBU, SW, XOR, XORI, LBU, LHU**	*JALR, SLL, SRA, SRL, LB*, LH**	*BGEZ, BGTZ, BLEZ, BLTZ, NOR, SLLV, SLT, SLTI, SLTIU, SLTU, SRAV, SRLV, DIVU*, MFHI*, MFLO*, MULTU**

*Nicht Teil des Befehlssatzes der VHDL-Implementierung

Trace-Format

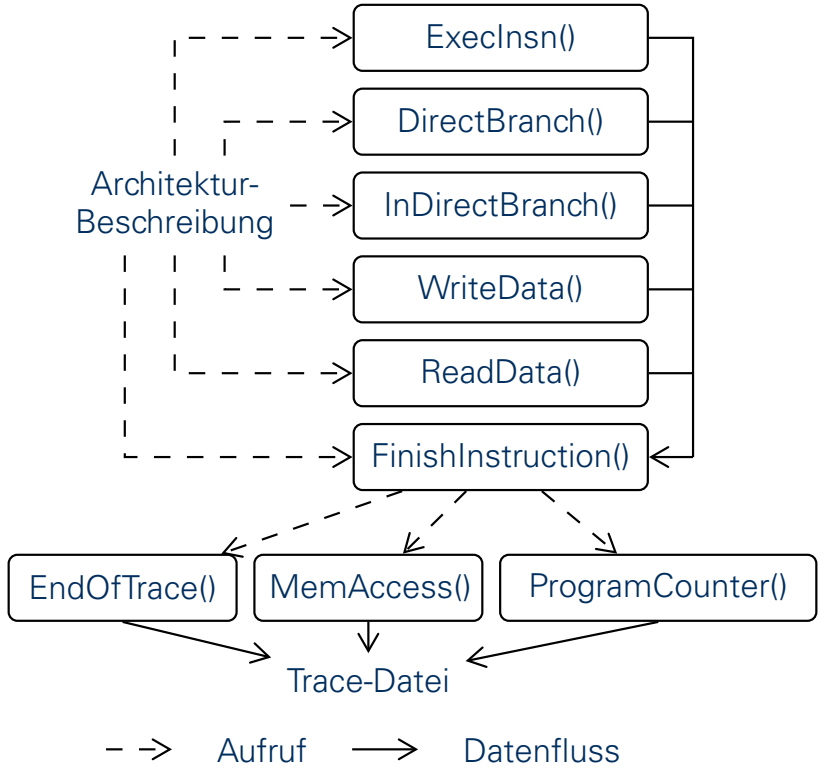
- **C**omma-**S**eparated-**V**alues-Format
- 3 Arten von Einträgen
  - Program-Counter-Eintrag
    - aktuelle Befehlsadresse
    - Branch-History
    - Anzahl ausgeführter Instruktionen
  - Memory-Access-Eintrag
    - Speicheradresse
    - Daten
  - Trace-Status-Eintrag
    - Zustandsinformationen der Trace-Hardware

Auszug einer Trace-Datei

```
...
53,00000128,0a,1d,,,,,,
54,,,,000043a8,00000017,1,0,,
55,,,,00004390,00000014,0,0,,
56,00000158,1a,3d,,,,,,
57,,,,00004398,00000018,1,0,,
58,,,,00004388,00000003,0,0,,
59,00000038,9b,04,,,,,,
60,,,,,,,,001,
```

Trace-Plugin für RUBICS

- C#-Klasse
- Methoden stehen Architektur-Beschreibung zur Verfügung
- sammelt Informationen während der Verarbeitung eines Befehls
- Einträge werden in FinishInstruction erzeugt
- Letzter Trace-Status-Eintrag mit *EndOfTrace*



Trace-Plugin in der Architektur-Beschreibung

```
$if TRACE_INSN $then
  $set TRACE_MEM_WRITE "TRC_MEM_W;"
  $set TRACE_MEM_READ "TRC_MEM_R;"
$else
  $set TRACE_MEM_WRITE ""
  $set TRACE_MEM_READ ""
$fi
...
operation TRC_MEM_W {
  trace.WriteData(toulong(pc), toulong(s1+imm), toulong(s2));
}
operation TRC_MEM_R {
  trace.ReadData(toulong(pc), toulong(s1+imm), toulong(d));
}
...
decode {
...
  0x23: {is; ls1r; LW; $TRACE_MEM_READ sdr;}
  0x2B: {is; ls1r; s2=d; ls2r; $TRACE_MEM_WRITE SW;}
...
}
```

3 Testprogramme

C-Testprogramme

- 18 Algorithmen insgesamt
- 3 Graphen- und Baumalgorithmen
  - Dijkstra
  - Floyd-Warshall
  - In-Order-Traversel in einem Binärbaum
- 5 Sortieralgorithmen
  - Insertion Sort
  - Heapsort
  - Radixsort
  - Shellsort
  - Bitonic Sort
- 5 Mathematische/Numerische Algorithmen
  - Sieb des Eratosthenes (vorgegeben)
  - Matrix-Matrix-Multiplikation
  - Primzahlfaktorisierung
  - Quadratwurzel mittels Newton-Raphson-Verfahren
- 2 Pseudozufallsgeneratoren
  - XorShift
  - LSFR basierend auf einer Veröffentlichung von Xilinx
- 2 String-Algorithmen
  - Knuth-Morris-Pratt (vorgegeben)
  - JS-Hash

Assembler-Testprogramme

- Der GCC nutzte einige Befehle nicht oder nur selten
  z.B. ADD, ADDI, SUB, J, JALR.
- Einige Befehle werden nur mit bestimmten Operanden genutzt
  z.B `jr ra`
- Alle 36 Befehle sollen abgedeckt werden
- 7 weitere Testprogramme in Assembler
  - Arithmetic
  - Branch
  - Load-Store
  - Logical
  - Registers
  - Setbit
  - Shift
- Befehle mit möglichst vielen Werten testen

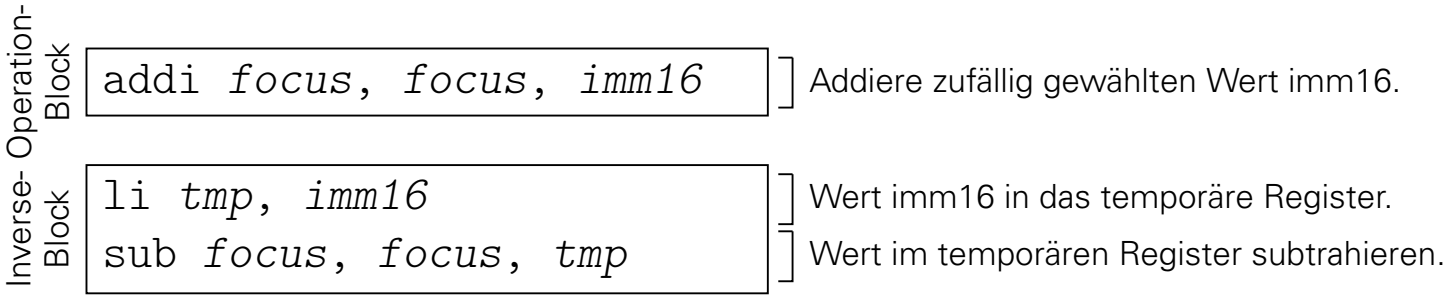
Reversi Test Generation System

- System zum Generieren zufälliger Testprogramme
- Wagner und Bertacco, „Reversi: Post-silicon validation system for modern microprocessors“
- Befehle werden mit ihren Gegenteilen getestet
- Idee: Prozessorzustand am Anfang und zum Ende soll gleich sein.
- Operation-Block $\mathrm{F_i(x)}$
- Inverse-Block $\mathrm{F_i^{-1}(x)}$
- Focus-Register $\mathrm{r_k}$

```
rk = rk << 3;
rk = rk + 5;
rk = rk * 7;
rk = rk - 9;
...
rk = rk + 9;
rk = rk / 7;
rk = rk - 5;
rk = rk >> 3;
```

$$\tilde{\mathrm{r_k}} = \mathrm{F_1^{-1}(F_2^{-1}(\ldots(F_n^{-1}(F_n(\ldots F_2(F_1(r_k)))\ldots))))}$$

Blockpaar: Addition/Subtraktion

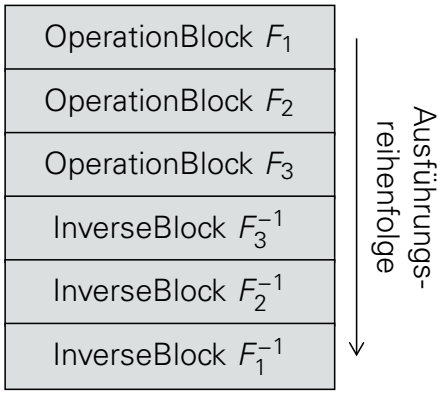


Reversi: Liste

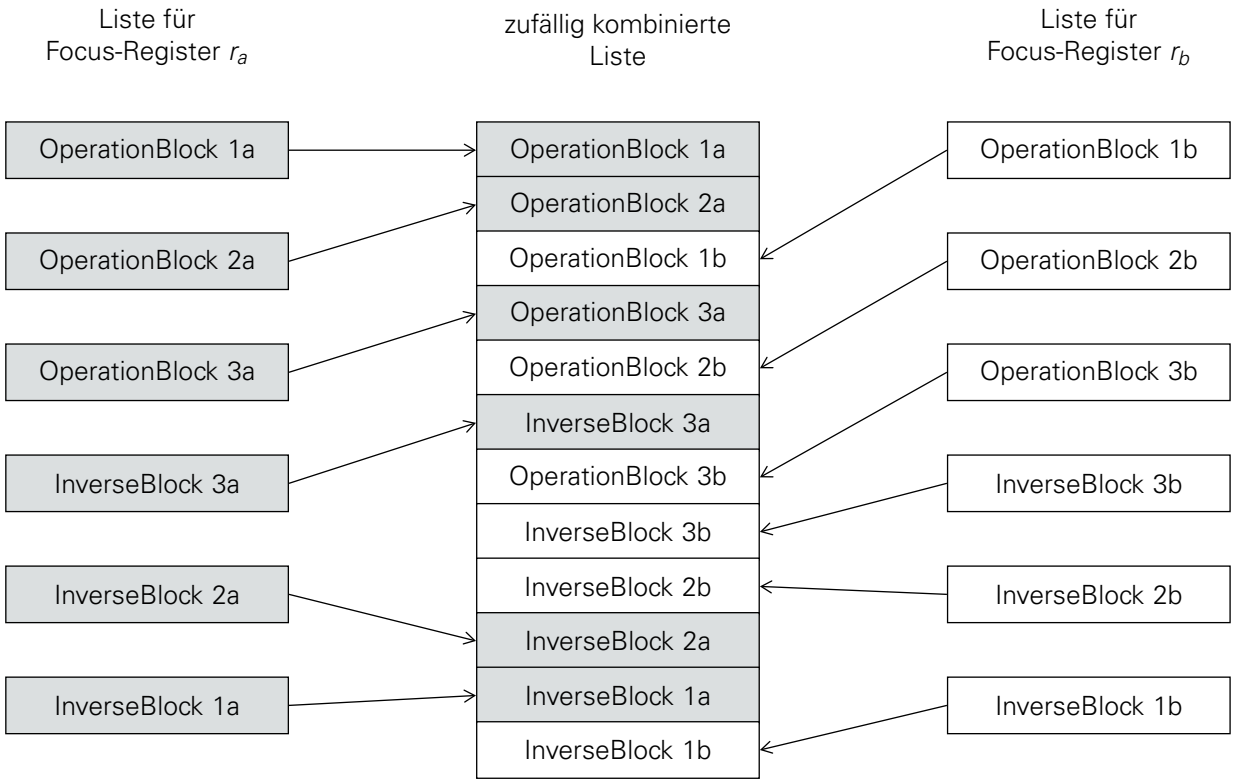
Reversi-Formel

$$r_k = F_1^{-1}(F_2^{-1}(\ldots(F_n^{-1}(F_n(\ldots F_2(F_1(r_k)))\ldots)$$

Liste für
Focus-Register r_k



Reversi: Shuffle



Reversi-Implementierung

- Python
- Reversi-Basis
  - Programm-Generator
  - Basis-Klasse für Blockpaar-Definition
- MIPS-spezifische Erweiterung
  - 25 definierte Blockpaare
  - Generator für Labels
  - Speicherverwaltung
- 24 Focus-Register
- 90 Blockpaare pro Focus-Register
- generiert 20.000 – 40.000 Zeilen Quelltext (MIPS Assembler)

4 Trace-Vergleich

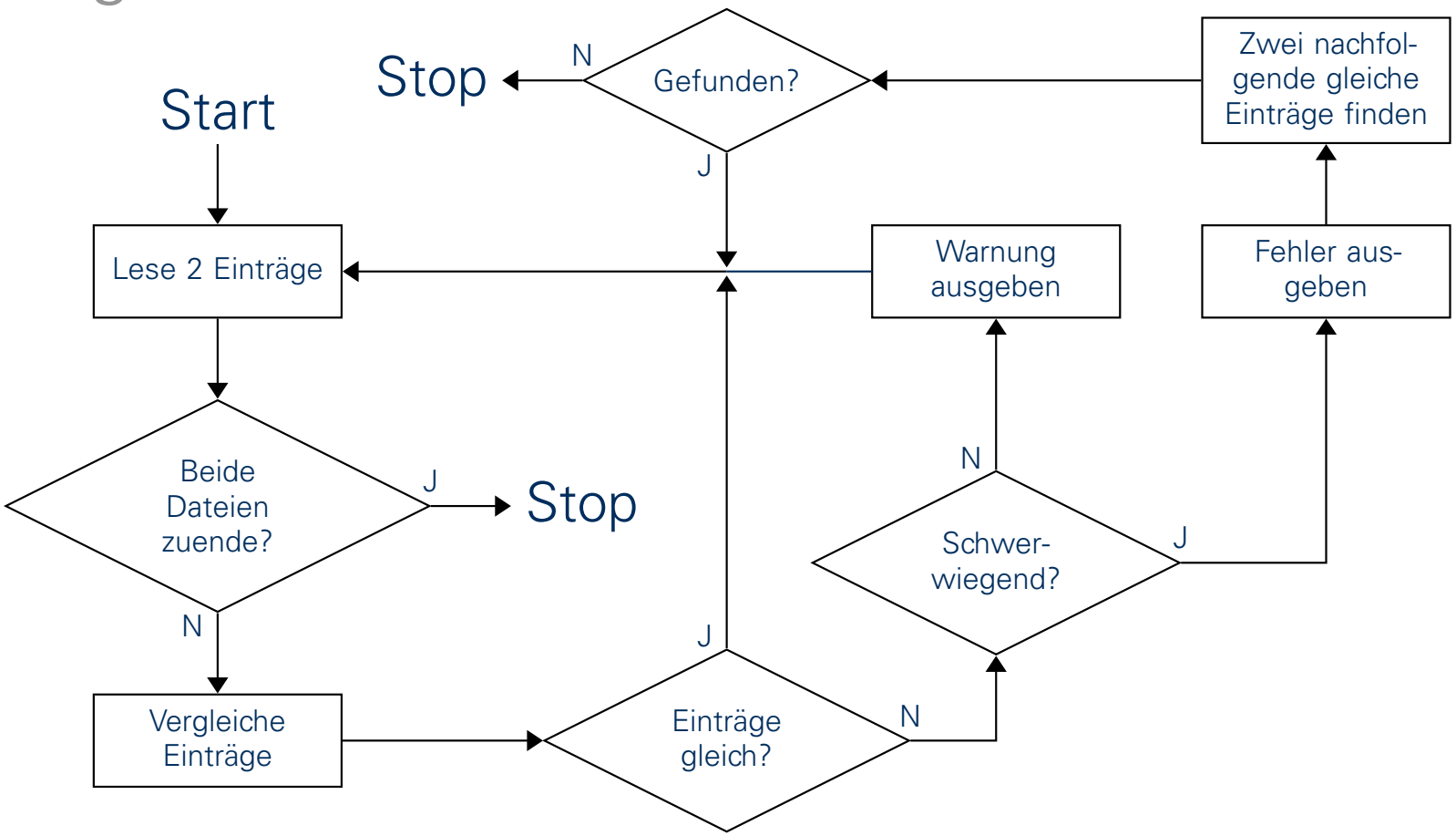
Trace-Vergleich

- Grundlegende Idee: diff-Werkzeug aus Linux/Unix-Umgebungen
- Trace-Einträge lesbar ausgeben
- Meisterwähnter Algorithmus: The Myers Diff algorithm
  - Benötigt beide Dateien komplett im RAM
  - Führt den Vergleich bis zu Ende durch

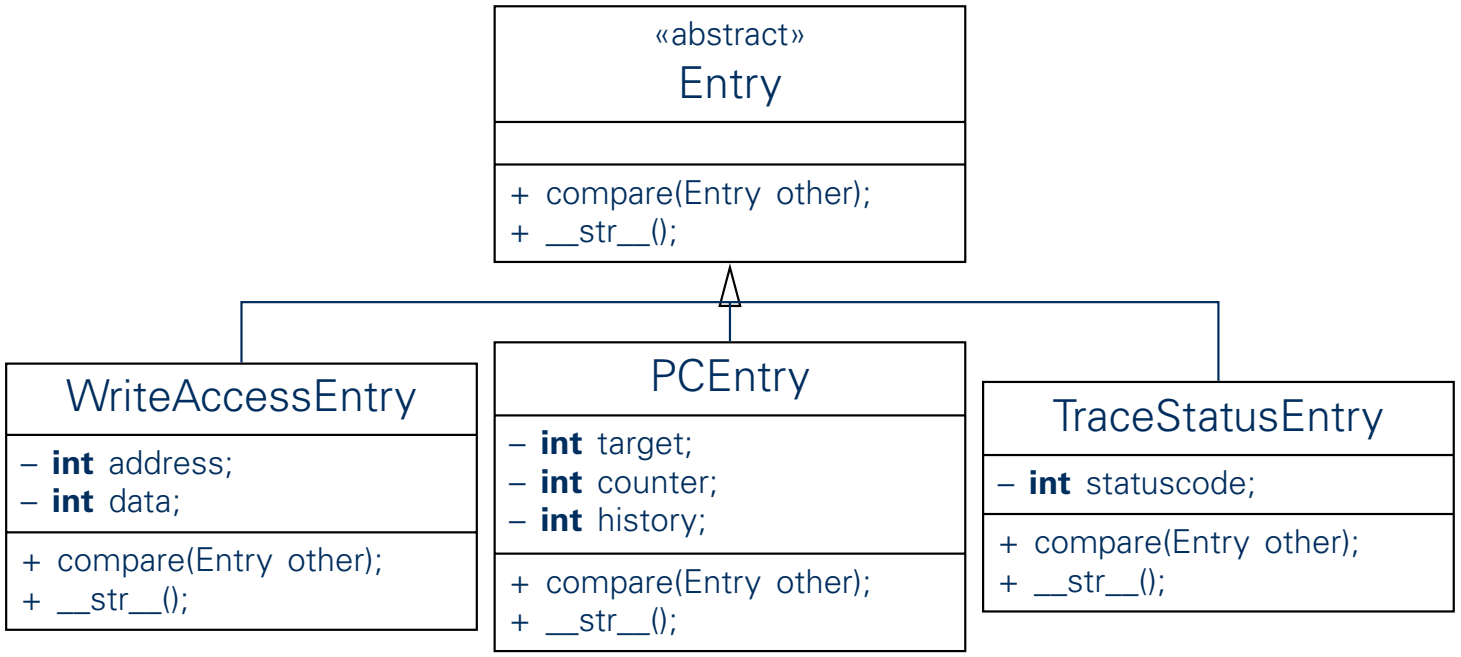
Trace-Vergleich-Programm

- Python 3
- konstanter Speicherverbrauch
- 2 Fehlerstufen
- erkennt fehlende Einträge
- leicht erweiterbar (Trace-Einträge und -Formate)

Programmablauf



Klassen für Trace-Einträge (Auswahl)

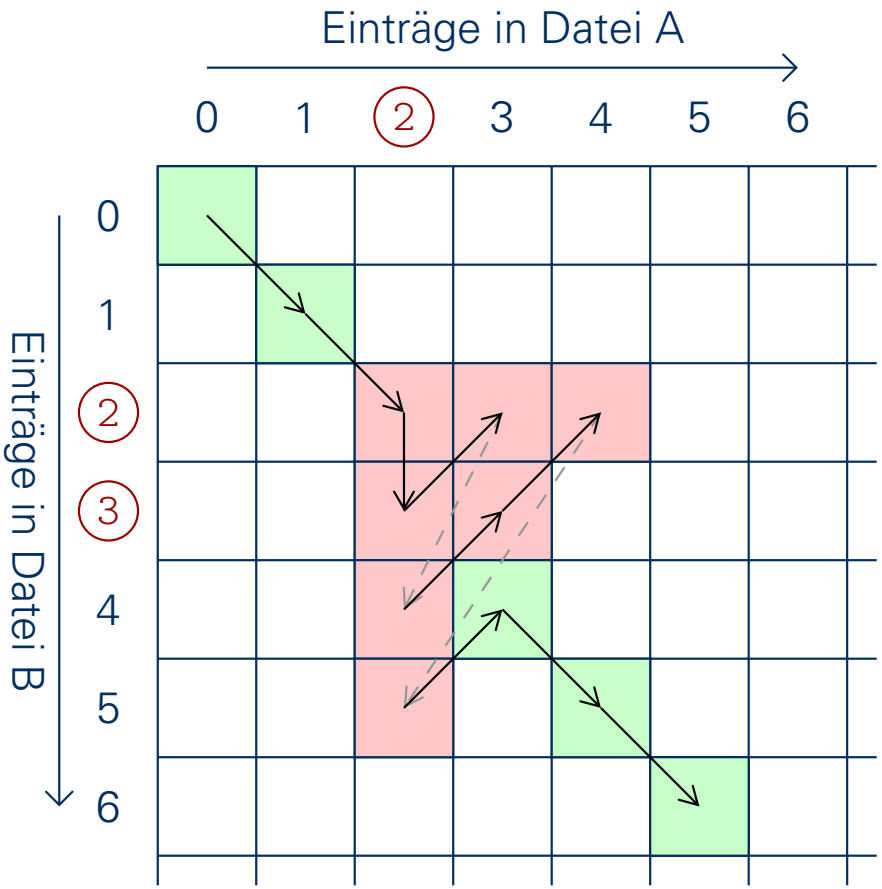


Fehlerstufen

- Warnung, nicht schwerwiegend (warning)
  - Unterschiedliche Daten beim Speicherzugriff
  - Branch-History ist unterschiedlich
  - ⇒ Prozessoren befinden sich wahrscheinlich an der gleichen Stelle im Programm
- Fehler, schwerwiegend (fatal)
  - Unterschiedliche Speicheradresse beim Speicherzugriff
  - Unterschiedliche Befehlsadresse im Program-Counter-Eintrag
  - ⇒ Prozessoren befinden sich wahrscheinlich an unterschiedlichen Stellen im Programm

Suchbeispiel

1. Normaler Vergleich
2. Suche nach zwei gleichen Einträgen
3. Meldung über unterschiedliche Zeilen
4. Normalen Vergleich fortsetzen



5 Ergebnisse

Gefundene Fehler

- 3 Fehler in der VHDL-Implementierung
  - 1 Fehler in SLLV/SRLV/SRAV

    GPR[rd] ← GPR[rt] << GPR[**rt**]
    statt GPR[rd] ← GPR[rt] << GPR[**rs**]
  - 2 Fehler in JALR

    GPR[**31**] ← **0**
    statt GPR[**rd**] ← **pc + 8**
- 1 Fehler, der beim Beheben anderer Fehler entstanden ist
  - JALR konnte r0 verändern
- 4 selbst-erzeugte Fehler
  - 2 in den Forwarding-Bedingungen
  - 1 in den Sprungbedingungen
  - 1 in der Vorzeichenerweiterung

JALR Fehler
Trace-Vergleich

```
### File A: error-jalr-ra.csv
### File B: rubics-git/trace_mips_branch.csv
----------------------------------------
Severity: Fatal
A Line   70: Program Counter at 00000000 (8 instructions executed before, 01
     branch history, last entry? = no)
B Line   70: Program Counter at 00000384 (8 instructions executed before, 01
     branch history, last entry? = no)
----------------------------------------
```

- Fehler: JALR nutzt Register 31 statt rd
- RUBICS springt nach 0x00000384
- VHDL-Implementierung springt nach 0x00000000

JALR Fehler

Fehlersuche

Trace der VHDL-Implementierung

```
67,00000364,06,01,,,,,,,
68,000002dc,08,02,,,,,,,
69,,,,000044d4,e6670a57,1,0,,,
70,00000000,08,01,,,,,,,
```

Trace von RUBICS

```
67,00000364,06,01,,,,,,,
68,000002dc,08,02,,,,,,,
69,,,,000044d4,e6670a57,1,0,,,
70,00000384,08,01,,,,,,,
```

Dissassembliertes Programm Teil 1

```
 364:   addiu    s0,s0,4
 368:   addi     s1,s1,-1
 // Hier wird nicht gesprungen
 36c:   bgtz     s1,354 <_test_jalr_1>
 370:   nop
 374:   lui      t0,0x0
 378:   addiu    t0,t0,732
 // Sprung nach 2dc
 37c:   jalr     a0,t0
 380:   nop
```

Disassembliertes Programm Teil 2

```
000002dc <test_fpoint_nora>:
    2dc:   addiu sp,sp,-4
    2e0:   lui   ra,0x70a5
    2e4:   ori   ra,ra,0x7e66
    2e8:   lui   t0,0xe667
    2ec:   ori   t0,t0,0xa57
    2f0:   sw    t0,0(sp)
    // Sprung zurueck
    2f4:   jr    a0
    2f8:   addiu sp,sp,4
```

JALR Fehler
VHDL-Quelltext

```
when "001001" =>       -- JALR
  rf_we <= '1';
  rf_wa <= "11111";    -- Register ra
  alu_src1_sel <= ALU_SRC1_SEL_PC_INC;
  alu_src2_sel <= ALU_SRC2_SEL_4;
  jmp_cond     <= JMP_COND_ALWAYS_REG;
  alu_cmd      <= ALU_CMD_ADDU;
```

- Problem:
  - VHDL-Implementierung setzt Register 31 als Zielregister
- Lösung:
  - Das von der Instruktion vorgegebene Register wählen

Vergleich zu RTL-Simulationen

Vorteile:

+ weniger Laufzeit
  - RTL-Simulation: 35 s
  - RUBICS: ~800 ms
  - FPGA: <1 s
  - Trace-Vergleich: ~70 ms
+ keine Testbenches
  - einfacheres Erstellen von Tests
  - konkreter Programmablauf unwichtig

Nachteile:

– begrenzte Sichtbarkeit
  - keine Liste von Signalen
  - keine taktgenaue Einsicht
– kleine Fehler sind schwer zu finden
  - keine oder wenig Unterschiede im Programmablauf zeigen sich nicht auf dem Trace.
– benötigt Befehlssatzsimulator
  - Nach dem Entwurf des Befehlssatzes oft schon vorhanden.

6 Zusammenfassung

Zusammenfassung

- RUBICS: Trace-Plugin und MIPS Befehlssatz (36+8 Befehle)
- Testprogramme
  - 18 Programme in C (2 davon vorgegeben)
  - 7 Programme in Assembler
  - 30 Reversi generierte Programme
- Trace-Vergleich-Programm
- 3 Fehler in der VHDL-Implementierung gefunden (5 weitere erzeugt und gefunden)
- Vorteile: weniger Laufzeit, keine Testbenches
- **Fazit:** Trace-Vergleich zum Entdecken von Fehlern, RTL-Simulation hilfreich zur Analyse von Fehlern

Fragen?

Literatur

Kasanko, Tuukka und Jari Nurmi. "Functional Verification of a 32-bit RISC Processor Core". In: (2004). URL: https://www.cs.tut.fi/~nurmi/SOC2004-Kasanko.pdf.

Mammo, Biruk et al. "Architectural Trace-Based Functional Coverage for Multiprocessor Verification". In: *Microprocessor Test and Verification (MTV), 2012 13th International Workshop on*. IEEE. 2012, pp. 1–5.

Wagner, Ilya und Valeria Bertacco. "Reversi: Post-silicon validation system for modern microprocessors". In: *Computer Design, 2008. ICCD 2008. IEEE International Conference on*. IEEE. 2008, pp. 307–314.