Lehrstuhl für Rechnerarchitektur und parallele Systeme
Fakultät für Informatik
Technische Universität München

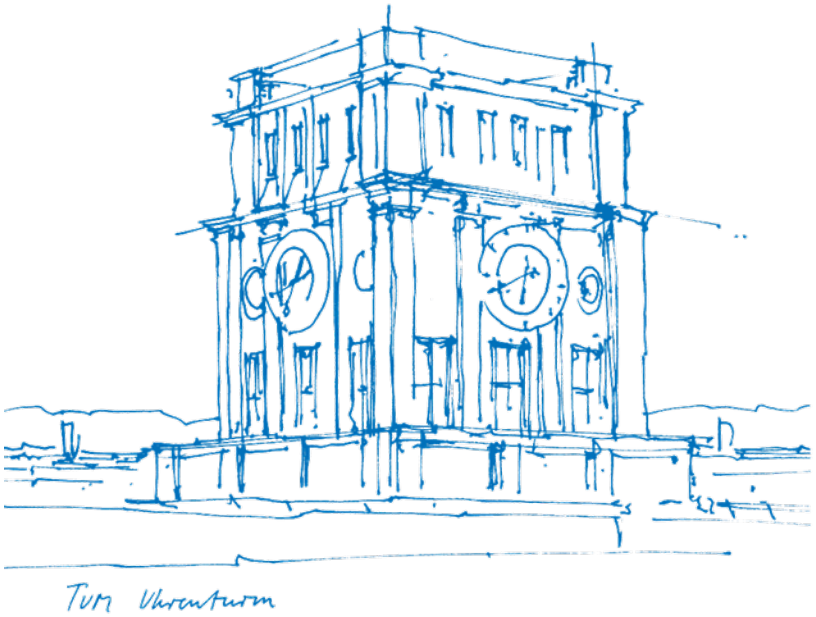
Dynamische Binärübersetzung: RISC–V → x86–64

Zwischenpräsentation

Noah Dormann[1], Simon Kammermeier[1],
Johannes Pfannschmidt[1], Florian Schmidt[1]

[1]Fakultät für Informatik, Technische Universität München (TUM)

21. Juli 2020



Gliederung

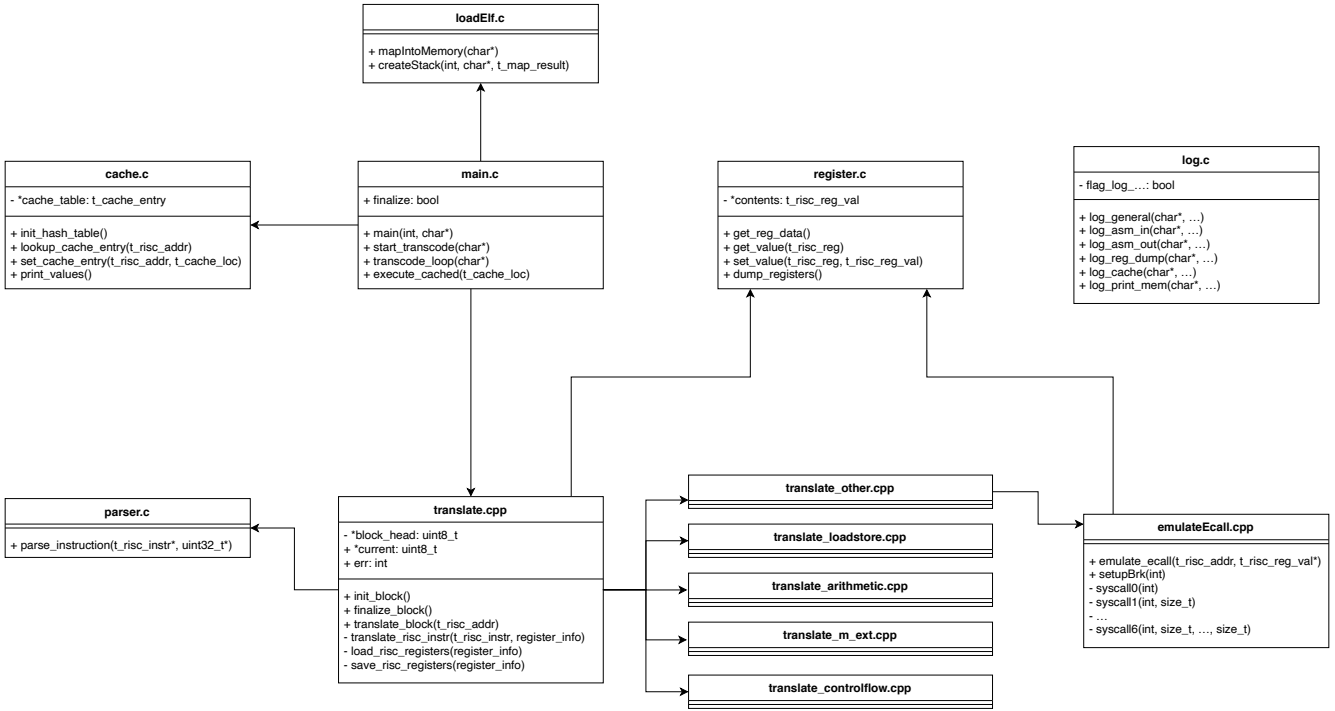
1. Einführung
1.1 Dynamische Binärübersetzung
1.2 Grobüberblick über die RISC–V ISA
1.3 Angebot

2. Systemarchitektur
2.1 ELF-Loader
2.2 Parser
2.3 Parser
2.4 Register File
2.5 Block Loader
2.6 Code Generator
2.7 Code Cache

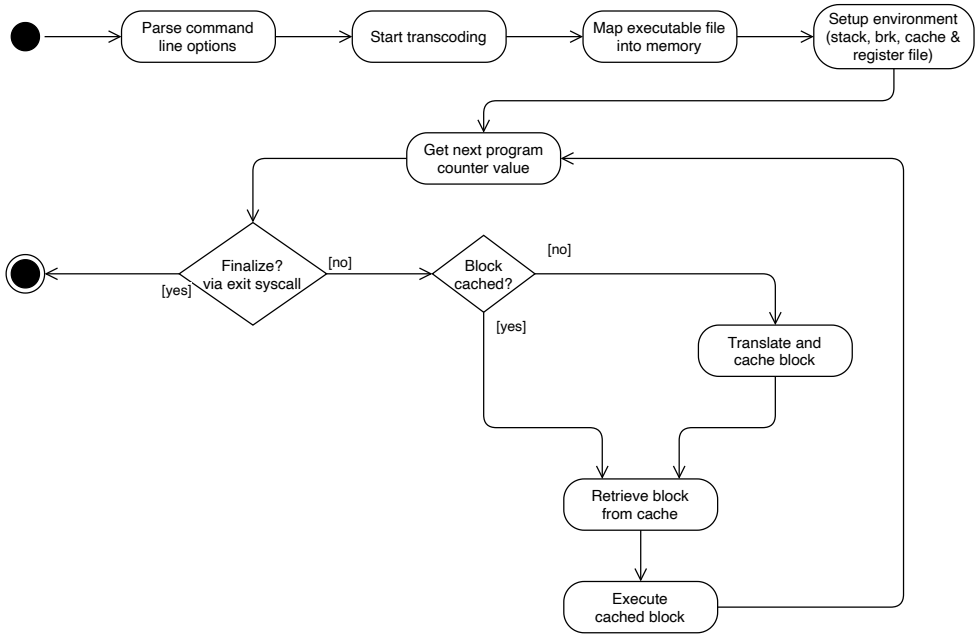
3. Anhang

Einführung

Systemarchitektur



Vorgehen


Parse command line options
Start transcoding
Map executable file into memory
Setup environment (stack, brk, cache & register file)
Get next program counter value
Finalize? via exit syscall
[yes]
[no]
Block cached?
[no]
[yes]
Translate and cache block
Retrieve block from cache
Execute cached block

ELF-Loader

Überblick

Aufgabe: Speicher für das Gastprogramm anlegen und mit Startwerten, Instruktionen, etc. befüllen

Aufgeteilt in:

- Memory mapping
- Stack Allocation

ELF-Loader

Memory mapping

Ziel: Die Binärdatei einlesen und alle Segmente, die mit „load“ gekennzeichnet sind, an die korrekten Orte im Speicher laden.

Input: String des Pfads der Binärdatei.
Output: Einsprungadresse des geladenen RISCV-Programms (plus einige Metadaten) in einem `t_risc_elf_map_result` struct.

```
typedef struct {
      bool valid;
      t_risc_addr entry;
      t_risc_addr phdr;
      Elf64_Half ph_count;
      Elf64_Half ph_entsize;
      t_risc_addr dataEnd;
} t_risc_elf_map_result;
```

ELF-Loader

Memory mapping

Ziel: Die Binärdatei einlesen und alle Segmente, die mit „load“ gekennzeichnet sind, an die korrekten Orte im Speicher laden.

Input: String des Pfads der Binärdatei.
Output: Einsprungadresse des geladenen RISCV-Programms (plus einige Metadaten) in einem `t_risc_elf_map_result` struct.

- Laden des ELF-Headers vom Anfang der Datei.
- Checken der Flags auf nicht unterstützte RISCV ABIs
- Iterieren über alle Segment-Header um den Addressbereich des Binarys zu erhalten
- Allozieren des gesamten benötigten Addressbereichs an der nativen Addresse
- Laden aller „load“ Segmente an die richtigen Speicheradressen

```
typedef struct {
    bool valid;
    t_risc_addr entry;
    t_risc_addr phdr;
    Elf64_Half ph_count;
    Elf64_Half ph_entsize;
    t_risc_addr dataEnd;
} t_risc_elf_map_result;
```

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings
Output: Die Adresse des Stackpointers nach dem Initialisieren

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings
Output: Die Adresse des Stackpointers nach dem Initialisieren

- Allozieren von Speicher für das Stack entsprechend des stack limits des Kernels (oder 8 MiB als Standardwert)

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings
Output: Die Adresse des Stackpointers nach dem Initialisieren

- Guard page am unteren Ende des Stacks um Overflow abzufangen.
- Alignment, damit der resultierende Stack Pointer die ABI erfüllt (16 Byte aligned)
- Kopieren der Werte des auxiliary vectors (aus Resultat des memory mappings bzw. Daten des Hostprogramms)
- Kopieren des enviroment vectors des Hostprogramms
- Kopieren des Argumentarrays und der Argumentanzahl

⋮ (Nicht Teil des Stacks)
Stack Alignment
Null terminierter auxiliary vector
Null terminierter enviroment vector
Null terminierter Argumentarray
Argumentanzahl (Stack Pointer am Ende auf hier)
⋮
Guard page

Parser

Überblick

Ziel: Decodieren der 4 byte großen codierten RISCV-Befehle aus der geladenen elf-Datei.
$\rightarrow$ (Zählen der Anzahl von Zugriffen auf einzelne Register für spätere Optimierungen.)

Input: Zeiger auf RISCV-Befehl im Speicher
Output: Ausgefüllte `t_risc_instr` Struktur mit allen relevanten Informationen

-
- Ausfüllen der `t_risc_instr` struct mit den Informationen
- Instruktionsmapping RISC-V $\rightarrow$ x86
- einzelne Übersetzungsfunktionen für jede Instruktion

- allokierte Speicherseite für die x86-Assembly
- Encoding der Instruktionen in den Speicherbereich

```
typedef struct {
    t_risc_addr addr;
    t_risc_mnem mnem;
    t_risc_optype optype;
    t_risc_reg reg_src_1;
    t_risc_reg reg_src_2;
    t_risc_reg reg_dest;
    t_risc_imm imm;
} t_risc_instr;
```

Parser

Ansatz

Ziel: Decodieren der 4 byte großen codierten RISCV-Befehle aus der geladenen elf-Datei.

```
void parse_instruction(t_risc_instr *p_instr_struct, uint32_t *reg_count);
```

Parser

Ansatz

Ziel: Decodieren der 4 byte großen codierten RISCV-Befehle aus der geladenen elf-Datei.

```
void parse_instruction(t_risc_instr *p_instr_struct, uint32_t *reg_count);
```

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

```
typedef struct {
    t_risc_addr addr;
    t_risc_mnem mnem;
    t_risc_reg_type op_type;
    t_risc_reg_reg_src_1;
```

Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Emulieren der Register `x0` bis `x31` sowie `pc` in

```
t_risc_reg_val contents[33];
```

Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Emulieren der Register `x0` bis `x31` sowie `pc` in

```
t_risc_reg_val contents[33];
```

und Zugriff via Startpointer und den convenience methods:

```
t_risc_reg_val *get_reg_data(void);
t_risc_reg_val get_value(t_risc_reg reg);
void set_value(t_risc_reg reg, t_risc_reg_val val);
```

z.T.: Caching der Inhalte in Hardware-x86-Registern je nach Registermapping für die Basic Blocks.

Dynamische Codegenerierung

Überblick

Input: geparste RISC-V-Instruktionen eines Basic Blocks
Output: übersetzte x86-Instruktionen für diesen Block

Dynamische Codegenerierung

Überblick

Input: geparste RISC-V-Instruktionen eines Basic Blocks
Output: übersetzte x86-Instruktionen für diesen Block

- Nutzen der Instruction-Structs des Parsers
- Instruktionsmapping RISC-V $\rightarrow$ x86
- einzelne Übersetzungsfunktionen für jede Instruktion

- allokierte Speicherseite für die x86-Assembly
- Encoding der Instruktionen in den Speicherbereich

```
typedef struct {
    t_risc_addr addr;
    t_risc_mnem mnem;
    t_risc_optype optype;
    t_risc_reg reg_src_1;
    t_risc_reg reg_src_2;
    t_risc_reg reg_dest;
    t_risc_imm imm;
} t_risc_instr;
```

Dynamische Codegenerierung

Ansatz

Übersetzung aller Instruktionen im Basic Block in einen x86-Buffer,

```
//aus translate_block(t_risc_addr), translate.cpp
init_block();

for (int i = 0; i < instructions_in_block; i++) {
   translate_risc_instr(block_cache[i], r_info);
}

return finalize_block();
```

anschließend

- Finalisieren des Blocks (`ret` anhängen, etc.)
- Rückgabe des Blocks an den Cache (später).

Dynamische Codegenerierung

Metadaten

Register-Mapping als Parameter für die Übersetzerfunktionen, basierend auf Zuteilung des Block Loaders:

```
/**
 * Register information for the translator functions.
 */
struct register_info {
   FeReg *map;
   bool *mapped;
   uint64_t base;
};
```

Dynamische Codegenerierung

Metadaten

Register-Mapping als Parameter für die Übersetzerfunktionen, basierend auf Zuteilung des Block Loaders:

```
/**
 * Register information for the translator functions.
 */
struct register_info {
   FeReg *map;
   bool *mapped;
   uint64_t base;
};
```

- Synchronisierung der zugewiesenen Register mit register file
- Lesen/Schreiben an Basic-Block-Grenzen
- Unterschiedliche Instruktionsübersetzungen je nach Mapping

Dynamische Codegenerierung

Dispatch

Verteilung der Übersetzung auf einzelne Funktionen für jede Instruktion:

```
//aus translate.cpp
void translate_risc_instr(const t_risc_instr &instr, const register_info &r_info) {
   switch (instr.mnem) {
      //...
      case OR:
         translate_OR(instr, r_info);
         break;
      case AND:
         translate_AND(instr, r_info);
         break;
      case SLLIW:
         translate_SLLIW(instr, r_info);
         break;
      //...
}
```

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Einfache Instruktionen, z.B. `ADD`:

```
//aus translate_arithmetic.cpp
void translate_ADD(const t_risc_instr &instr, const register_info &r_info) {
   if (r_info.mapped[instr.reg_dest] && r_info.mapped[instr.reg_src_1] &&
       r_info.mapped[instr.reg_src_2]) {
      //...
   } else {
      err |= fe_enc64(&current, FE_MOV64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_1));
      err |= fe_enc64(&current, FE_ADD64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_2));
      err |= fe_enc64(&current, FE_MOV64mr, FE_MEM_ADDR(r_info.base + 8 * instr.reg_dest), FE_AX);
   }
}
```

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Einfache Instruktionen, z.B. `ADD`:

```
//aus translate_arithmetic.cpp
void translate_ADD(const t_risc_instr &instr, const register_info &r_info) {
   if (r_info.mapped[instr.reg_dest] && r_info.mapped[instr.reg_src_1] &&
       r_info.mapped[instr.reg_src_2]) {
      //...
   } else {
      err |= fe_enc64(&current, FE_MOV64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_1));
      err |= fe_enc64(&current, FE_ADD64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_2));
      err |= fe_enc64(&current, FE_MOV64mr, FE_MEM_ADDR(r_info.base + 8 * instr.reg_dest), FE_AX);
   }
}
```

→ Load-Store-Architektur vs. Register-Memory-Architecture

Dynamische Codegenerierung

Übersetzerfunktionen (2)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Notwendigkeit von Fallunterscheidungen, z.B. `REM`: (Semantik nach[1], S. 44f.)

[1] A. Waterman et al. (2017). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2*.

Dynamische Codegenerierung

Übersetzerfunktionen (2)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Notwendigkeit von Fallunterscheidungen, z.B. `REM`: (Semantik nach[1], S. 44f.)

```
mov rax, [r_info.base + 8 * instr.reg_src_1]
cmp qword ptr [r_info.base + 8 * instr.reg_src_2], 0
jnz not_div_zero
mov [r_info.base + 8 * instr.reg_dest], rax
jz div_zero

not_div_zero:
xor rdx, rdx
idiv qword ptr [r_info.base + 8 * instr.reg_src_2]
mov [r_info.base + 8 * instr.reg_dest], rdx

div_zero:
```

[1] A. Waterman et al. (2017). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2.*

Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für `ECALL`:

Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für `ECALL`:

```
void translate_ECALL(const t_risc_instr &instr, const register_info &r_info) {
   save_risc_registers(r_info);
   err |= fe_enc64(&current, FE_MOV64ri, FE_DI, instr.addr);
   err |= fe_enc64(&current, FE_MOV64ri, FE_SI, r_info.base);
   typedef void emulate(t_risc_addr addr, t_risc_reg_val *registerValues);
   emulate *em = &emulate_ecall;
   err |= fe_enc64(&current, FE_CALL, reinterpret_cast<uintptr_t>(em));
}
```

Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für `ECALL`:

```
void translate_ECALL(const t_risc_instr &instr, const register_info &r_info) {
   save_risc_registers(r_info);
   err |= fe_enc64(&current, FE_MOV64ri, FE_DI, instr.addr);
   err |= fe_enc64(&current, FE_MOV64ri, FE_SI, r_info.base);
   typedef void emulate(t_risc_addr addr, t_risc_reg_val *registerValues);
   emulate *em = &emulate_ecall;
   err |= fe_enc64(&current, FE_CALL, reinterpret_cast<uintptr_t>(em));
}
```

- Behandlung von system calls zur Laufzeit
- Übersetzung, Adaptieren bzw. Emulieren der benötigten Funktionalität

Code Cache

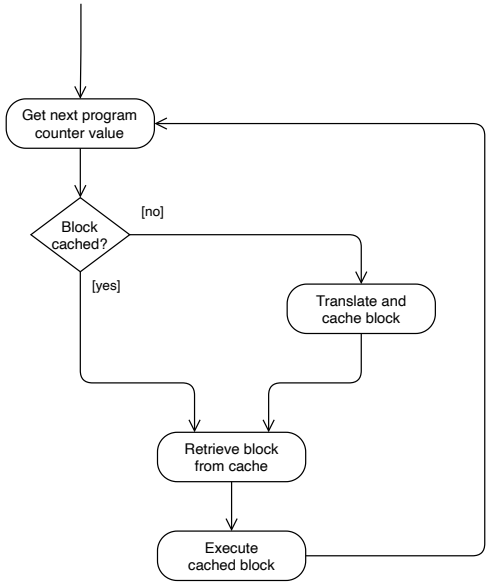
Überblick

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)

Code Cache

Überblick

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)



Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)
Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)
Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Einträge speichern RISC-V-Blockstartadresse sowie die Adresse des übersetzten Blocks:

```
typedef struct {
   t_risc_addr risc_addr;
   t_cache_loc cache_loc;
} t_cache_entry;
```

Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)
Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Einträge speichern RISC-V-Blockstartadresse sowie die Adresse des übersetzten Blocks:

```
typedef struct {
   t_risc_addr risc_addr;
   t_cache_loc cache_loc;
} t_cache_entry;
```

Lookup als Open Hashing mit linearem Sondieren, via

```
inline size_t hash(t_risc_addr risc_addr) {
   return (risc_addr & 0x0000FFF0u) >> 4u;
}
```

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei `UNSEEN_CODE` von `lookup_cache_entry(...)` einen nicht im Cache enthaltenen Block anzeigt.

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei `UNSEEN_CODE` von `lookup_cache_entry(...)` einen nicht im Cache enthaltenen Block anzeigt.

Ausführung bereits übersetzter Blöcke via

```
typedef void (*blk)(void);
((blk) loc)();
```

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei `UNSEEN_CODE` von `lookup_cache_entry(...)` einen nicht im Cache enthaltenen Block anzeigt.

Ausführung bereits übersetzter Blöcke via

```
typedef void (*blk)(void);
((blk) loc)();
```

→ Dynamische Reallokation der Größe bei Kapazitätsgrenzen

Literaturverzeichnis

 Waterman, A. et al. (2017). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2*.