Aufgabe 4 – Erweiterung um Just-in-Time-Compiler

Dr.-Ing. Volkmar Sieh

Department Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität Erlangen-Nürnberg

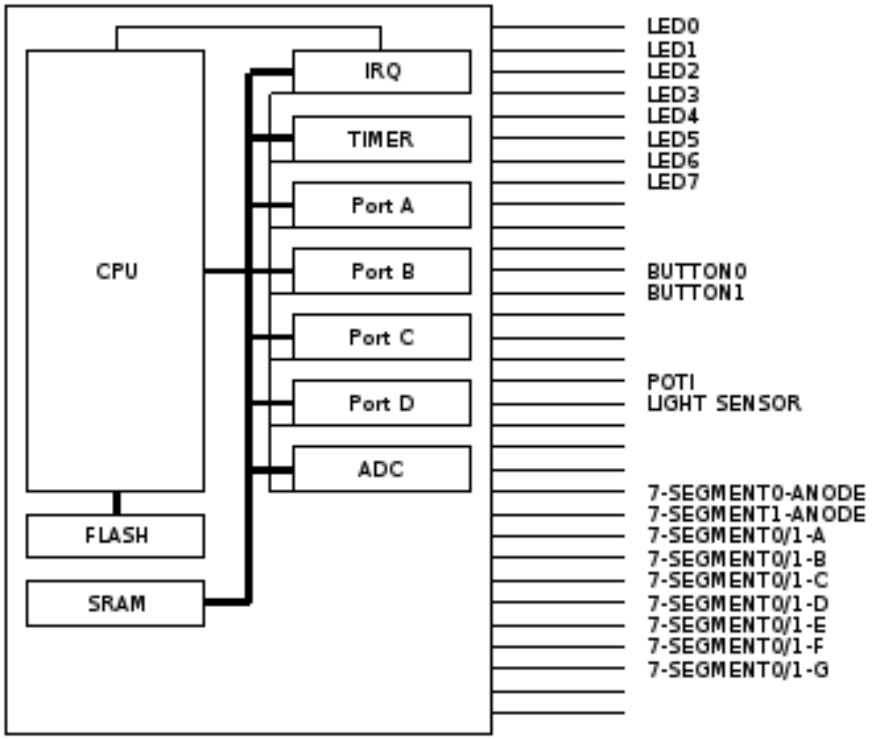
WS 2018/2019

Einführung

Entwickelt werden soll ein virtuelles (vereinfachtes) SPiC-Board mit ATmega32-Mikrokontroller mit JIT-Compiler.

SPiC-Board – Aufbau (vereinfacht)

Aufgabe 4 (unverändert gegenüber Aufgabe 3):



Geforderte Funktionalität

- Der CPU-Teil des Mikrokontrollers soll jetzt als Just-in-Time-Compiler ausgeführt werden.
- Basisblöcke sollen einmal am Stück compiliert und nachfolgend ggf. mehrfach ausgeführt werden können.
- Optimierungen (Block-Verkettungen, Lazy-Flags-Berechnungen u.ä.) sind nicht gefordert.
- Bestehende Unterprogramme (z.B. zur Berechnung eine Additions-Ergebnisses mit seinen Flags) dürfen vom JIT-Code aus verwendet werden.

Geforderte Funktionalität

Getestet werden soll die neue CPU mit den bisherigen Testprogrammen. Diese sollen lauffähig bleiben.

Bewertung

- Die Performance des alten Interpreters und die des neuen JIT-Ansatzes soll verglichen werden.
- Berechnen Sie den Speedup.
- Identifizieren Sie die noch bestehenden Flaschenhälse.

Hinweise

Im Folgenden:

(Gedankliche) Schritte, um von einem Interpreter zu einem Just-in-Time-Compiler zu kommen...

Hinweise

```
void step(struct state *s) {
    /* 1. Pipeline-Stufe: Instruktion holen */
    inst = fetch(s->pc++);

    switch (inst_format(inst) {
    case ALU_REG_REG:
        /* 2. Pipeline-Stufe: Operanden holen */
        op1 = reg_read(s, (inst >> 0) & 0xf);
        op2 = reg_read(s, (inst >> 4) & 0xf);

        /* 3. Pipeline-Stufe: Rechnen */
        res = alu(s, (inst >> 24) & 0xf, op1, op2);

        /* 4. Pipeline-Stufe: Ergebnis speichern */
        reg_write(s, (inst >> 20) & 0xf, res);
        break;

    case ALU_REG_IMM:
        /* 2. Pipeline-Stufe: Operanden holen */
        op1 = reg_read(s, (inst >> 0) & 0xf);
        op2 = (inst >> 4) & 0xffff;

        /* 3. Pipeline-Stufe: Rechnen */
        res = alu(s, (inst >> 24) & 0xf, op1, op2);

        /* 4. Pipeline-Stufe: Ergebnis speichern */
        reg_write(s, (inst >> 20) & 0xf, res);
        break;

    case ...
        ...
    }
}
```

Basisblock:

```
15: add %r0, %r1
16: sub $13, %r1
17: mov %r1, %r8
18: cmp $0, %8
19: jne 25
```

Wie sehe entsprechende C-Funktion aus?

Hinweise

```
void step(struct state *s) {
    /* 1. Pipeline-Stufe: Instruktion holen */
    inst = fetch(s->pc++);

    switch (inst_format(inst) {
    case ALU_REG_REG:
        /* 2. Pipeline-Stufe: Operanden holen */
        op1 = reg_read(s, (inst >> 0) & 0xf);
        op2 = reg_read(s, (inst >> 4) & 0xf);

        /* 3. Pipeline-Stufe: Rechnen */
        res = alu(s, (inst >> 24) & 0xf, op1, op2);

        /* 4. Pipeline-Stufe: Ergebnis speichern */
        reg_write(s, (inst >> 20) & 0xf, res);
        break;

    case ALU_REG_IMM:
        /* 2. Pipeline-Stufe: Operanden holen */
        op1 = reg_read(s, (inst >> 0) & 0xf);
        op2 = (inst >> 4) & 0xffff;

        /* 3. Pipeline-Stufe: Rechnen */
        res = alu(s, (inst >> 24) & 0xf, op1, op2);

        /* 4. Pipeline-Stufe: Ergebnis speichern */
        reg_write(s, (inst >> 20) & 0xf, res);
        break;

    case ...
        ...
    }
}
```

```
15: add %r0, %r1
16: sub $13, %r1
17: mov %r1, %r8
18: cmp $0, %r8
19: jne 25
```

```
void block(struct state *s) {
    op1 = reg_read(s, 0);
    op2 = reg_read(s, 1);
    res = alu(s, ADD, op1, op2);
    reg_write(s, 1, res);

    op1 = reg_read(s, 1);
    op2 = 13;
    res = alu(s, SUB, op1, op2);
    reg_write(s, 1, res);

    op1 = reg_read(s, 1);
    res = op1;
    reg_write(s, 8, res);

    op1 = reg_read(s, 8);
    op2 = 0;
    alu(s, SUB, op1, op2);

    if (! s->z) {
        s->pc = 25;
    } else {
        s->pc = 20;
    }
}
```

Hinweise

```
void block(struct state *s) {
    int op1;
    int op2;
    int res;

    op1 = reg_read(s, 0);
    op2 = reg_read(s, 1);
    res = alu(s, ADD, op1, op2);
    reg_write(s, 1, res);

    op1 = reg_read(s, 1);
    op2 = 13;
    res = alu(s, SUB, op1, op2);
    reg_write(s, 1, res);

    op1 = reg_read(s, 1);
    res = op1;
    reg_write(s, 8, res);

    op1 = reg_read(s, 8);
    op2 = 0;
    alu(s, SUB, op1, op2);

    if (! s->z) {
        s->pc = 25;
    } else {
        s->pc = 20;
    }
}
```

Wie sehe entsprechender Assembler-Code aus?

Hinweise

```
void block(struct state *s) {
    int op1;
    int op2;
    int res;

    op1 = reg_read(s, 0);
    op2 = reg_read(s, 1);
    res = alu(s, ADD, op1, op2);
    reg_write(s, 1, res);

    ...
}
```

```
block: // s in %rdi
    pushq %rbp
    movq %rdi, %rbp
    subq $16, %rsp

    movq %rbp, %rdi
    movl $0, %esi
    call reg_read
    movl %eax, 0(%rsp)

    movq %rbp, %rdi
    movl $1, %esi
    call reg_read
    movl %eax, 4(%rsp)

    movq %rbp, %rdi
    movl $ADD, %esi
    movl 0(%rsp), %edx
    movq 4(%rsp), %ecx
    call alu
    movl %eax, 8(%rsp)

    movq %rbp, %rdi
    movl $1, %esi
    movl 8(%rsp), %edx
    call reg_write

    ...

    addq $16, %rsp
    popq %rbp
    ret
```

Hinweise

```
void block(struct state *s) {
    int op1;
    int op2;
    int res;

    ...

    if (! s->z) {
        s->pc = 25;
    } else {
        s->pc = 20;
    }
}
```

```
block: // s in %rdi
    pushq %rbp
    movq %rdi, %rbp
    subq $16, %rsp

    ...

    cmpb $0, off_z(%rbp)
    jne l1
    movl $25, off_pc(%rbp)
    jmp l2;
l1: movl $20, off_pc(%rbp)
l2:

    addq $16, %rsp
    popq %rbp
    ret
```

Hinweise

```
block: // s in %rdi
    pushq %rbp
    movq %rdi, %rbp
    subq $16, %rsp

    movq %rbp, %rdi
    movl $0, %esi
    call reg_read
    movl %eax, 0(%rsp)

    ...

    cmpb $0, off_z(%rbp)
    jne l1
    movl $25, off_pc(%rbp)
    jmp l2;
l1: movl $20, off_pc(%rbp)
l2:

    addq $16, %rsp
    popq %rbp
    ret
```

Wie sehe Binär-Code aus?

Hinweise

```
block: // s in %rdi
    pushq %rbp
    movq %rdi, %rbp
    subq $16, %rsp

    movq %rbp, %rdi
    movl $0, %esi
    call reg_read
    movl %eax, 0(%rsp)

    ...

    cmpb $0, 0x5(%rbp)
    jne l1
    movl $25, 0x12(%rbp)
    jmp l2;
l1: movl $20, 0x12(%rbp)
l2:
    addq $16, %rsp
    popq %rbp
    ret
```

```
      55                    push   %rbp
      48 89 fd              mov    %rdi,%rbp
      48 83 ec 10           sub    $0x10,%rsp

      48 89 ef              mov    %rbp,%rdi
      be 00 00 00 00        mov    $0x0,%esi
      e8 (reg_read - lab1)  callq  reg_read
lab1: 89 04 24              mov    %eax,(%rsp)

      ...

      80 7d 05 00           cmpb   $0x0,0x5(%rbp)
      75 (lab3 - lab2)      jne    lab3
lab2: c7 45 12 19 00 00 00  movl   $0x19,0x12(%rbp)
      eb (lab4 - lab3)      jmp    lab4
lab3: c7 45 12 14 00 00 00  movl   $0x14,0x12(%rbp)

lab4: 48 83 c4 10           add    $0x10,%rsp
      5d                    pop    %rbp
      c3                    retq
```

Hinweise

Damit Bytes im Speicher ausführbar sind, muss der entsprechende Speicherbereit in der MMU als „ausführbar" markiert sein.

In der MMU werden alle Speicherbereiche als Seiten verwaltet. Seiten müssen an Adressen liegen, die durch die Seitengröße teilbar sind.

Unter Linux/gcc:

```
#include <sys/mman.h>

char jit_buf[256*4096] __attribute__((aligned(4096)));
int ret;

ret = mprotect(jit_buf, sizeof(jit_buf),
        PROT_READ | PROT_WRITE | PROT_EXEC);
assert(0 <= ret);
```

Hinweise

Hinweise x86_64-Programmierung:

https://de.wikipedia.org/wiki/AMD64

Hinweise

Bei Problemen gerne/rechtzeitig melden!