Bab 12

Code Generation dan Target Machine

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengimplementasikan code generator untuk arsitektur target

12.1 Pengenalan Target Machine (RISC vs CISC)

12.1.1 Arsitektur RISC (Reduced Instruction Set Computer)

Contoh: RISC-V, ARM.

- Instruksi berukuran tetap (biasanya 32-bit).
- Hanya instruksi `Load` dan `Store` yang bisa mengakses memori.
- Memiliki banyak register umum (*general purpose*).

12.1.2 Arsitektur CISC (Complex Instruction Set Computer)

Contoh: x86 (Intel/AMD).

- Instruksi berukuran variabel (1-15 byte).
- Instruksi aritmatika bisa langsung mengakses operan di memori.
- Jumlah register fisik lebih terbatas dibanding RISC.

12.1.3 Dampak pada Code Generation

Code generator harus menyadari arsitektur target. Pada RISC, operasi aritmatika yang kompleks harus diurai menjadi beberapa instruksi dasar `load-load-op-store`.

12.2 Pemilihan Instruksi (Instruction Selection)

Instruction Selection adalah proses memetakan instruksi tingkat menengah (*TAC*) ke instruksi spesifik mesin target yang memberikan performa terbaik.

12.2.1 Pattern Matching

Kompilator modern sering menggunakan pencocokan pola pohon (*Tree Pattern Matching*) untuk memilih instruksi kompleks. Misal: `a = b * c + d` dapat diganti dengan satu instruksi *Multiply-Add* jika CPU mendukungnya.

12.2.2 Biaya Instruksi (Cost Estimation)

Setiap opsi instruksi memiliki bobot (siklus CPU). *Instruction selector* mencoba meminimalkan total bobot instruksi yang dihasilkan untuk sebuah blok kode.

12.3 Register Allocation

12.3.1 Register Allocation Problem

Masalah alokasi register:

- Variabel terbatas vs unlimited temporaries
- Register interference
- Spilling ke memory
- Calling convention constraints

12.3.2 Linear Scan Allocation

```
typedef struct {
    char *var_name;
    int start_point;    // First use
    int end_point;      // Last use
    int register_num;   // Assigned register (-1 if spilled)
} LiveRange;

void linear_scan_allocation(LiveRange *ranges, int count,
                            int num_registers) {
    // Sort ranges by start point
    sort_ranges_by_start(ranges, count);

    bool *registers_used = calloc(num_registers, sizeof(bool));

```

```
    for (int i = 0; i < count; i++) {
        // Free registers whose ranges have ended
        free_expired_registers(ranges[i].start_point,
                               registers_used, num_registers);

        // Find free register
        int reg = find_free_register(registers_used, num_registers);
        if (reg != -1) {
            ranges[i].register_num = reg;
            registers_used[reg] = true;
        } else {
            // Spill to memory
            ranges[i].register_num = -1;
        }
    }
}
```

12.4 Target Architecture

12.4.1 x86 Architecture

Karakteristik x86:

- CISC (Complex Instruction Set Computer)
- Variable-length instructions
- Rich addressing modes
- Backward compatibility

```
// x86 instruction examples
MOV EAX, EBX          ; Register to register
MOV EAX, [EBX+4]      ; Base + offset addressing
MOV EAX, [EBX+ECX*4] ; Base + index*scale
LEA EAX, [EBX+ECX*2] ; Load effective address
PUSH EAX              ; Stack operation
POP EBX               ; Stack operation
```

12.4.2 RISC Architecture

Karakteristik RISC (MIPS/ARM):

- Fixed-length instructions
- Load/store architecture
- Simple addressing modes

- Large register file

```
// MIPS instruction examples
ADD $t0, $t1, $t2    ; $t0 = $t1 + $t2
LW $t0, 4($t1)       ; Load word from memory
SW $t0, 8($t1)       ; Store word to memory
ADDI $t0, $t1, 10    ; $t0 = $t1 + 10 (immediate)
```

12.5 Code Generation Algorithm

12.5.1 Basic Block Code Generation

```
void generate_block_code(BasicBlock *block,
                         RegisterAllocator *alloc) {
    for (int i = 0; i < block->instruction_count; i++) {
        TACInstruction *inst = &block->instructions[i];

        // Allocate registers for operands
        int reg1 = allocate_register(inst->arg1, alloc);
        int reg2 = allocate_register(inst->arg2, alloc);
        int reg3 = allocate_register(inst->result, alloc);

        // Generate target instruction
        switch (inst->op) {
            case OP_ADD:
                emit_add(reg3, reg1, reg2);
                break;
            case OP_MUL:
                emit_mul(reg3, reg1, reg2);
                break;
            case OP_ASSIGN:
                emit_mov(reg3, reg1);
                break;
            // ... other operations
        }

        // Release temporary registers
        release_register(reg1, alloc);
        release_register(reg2, alloc);
    }
}
```

12.5.2 Function Call Generation

```
void generate_function_call(FunctionCall *call) {
    // Save caller-saved registers
    save_caller_saved_registers();

    // Push parameters (right-to-left for cdecl)
    for (int i = call->param_count - 1; i >= 0; i--) {
        push_parameter(call->parameters[i]);
```

```
    }

    // Call function
    emit_call(call->function_name);

    // Clean up stack (callee or caller depending on convention)
    cleanup_stack(call->param_count * sizeof(int));

    // Restore caller-saved registers
    restore_caller_saved_registers();

    // Move return value to target register
    if (call->has_return_value) {
        emit_mov(call->return_reg, EAX);
    }
}
```

12.6 Optimization in Code Generation

12.6.1 Peephole Optimization

```
void peephole_optimization(Instruction *instructions, int count) {
    for (int i = 0; i < count - 1; i++) {
        // MOV reg, reg -> NOP
        if (is_mov_reg_to_reg(&instructions[i])) {
            instructions[i].opcode = NOP;
        }

        // PUSH reg; POP reg -> NOP
        if (is_push_pop_same_reg(&instructions[i], &instructions[i+1])) {
            instructions[i].opcode = NOP;
            instructions[i+1].opcode = NOP;
        }

        // MOV reg, imm; ADD reg, imm -> LEA reg, [imm]
        if (can_convert_to_lea(&instructions[i], &instructions[i+1])) {
            convert_to_lea(&instructions[i], &instructions[i+1]);
        }
    }
}
```

12.6.2 Instruction Scheduling

```
void schedule_instructions(Instruction *instructions, int count) {
    // Simple list scheduling for basic blocks
    Instruction *scheduled[count];
    int scheduled_count = 0;

    while (scheduled_count < count) {
        // Find ready instructions (no dependencies)
        for (int i = 0; i < count; i++) {
```

```
            if (!is_scheduled(&instructions[i]) &&
                is_ready(&instructions[i], scheduled, scheduled_count)) {
                scheduled[scheduled_count++] = instructions[i];
                break;
            }
        }
    }

    // Copy scheduled instructions back
    memcpy(instructions, scheduled, count * sizeof(Instruction));
}
```

Aktivitas Pembelajaran

1. **Instruction Selection**: Implementasikan instruction selector untuk subset x86.
2. **Register Allocation**: Bangun linear scan register allocator.
3. **Code Generation**: Implementasikan code generator untuk simple expressions.
4. **Peephole Optimization**: Buat peephole optimizer untuk assembly code.
5. **Function Calls**: Generate code untuk function calls dengan calling conventions.

Latihan dan Refleksi

1. Generate assembly code untuk expression tree kompleks!
2. Implementasikan register allocator dengan spilling strategy!
3. Analisis instruction selection for different target architectures!
4. Optimasi generated code dengan peephole optimizations!
5. Generate code untuk recursive functions dengan proper stack management!
6. **Refleksi**: Bagaimana target architecture mempengaruhi code generation strategy?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Register allocation problem terjadi karena:
   (a) Terlalu banyak variabel
   (b) Terbatasnya jumlah register
   (c) Memory terlalu kecil
   (d) Instruksi terlalu kompleks
2. CISC architecture memiliki:
   (a) Fixed-length instructions
   (b) Variable-length instructions
   (c) Load/store only
   (d) Large register file
3. Peephole optimization bekerja pada:
   (a) Single instruction
   (b) Small window of instructions
   (c) Entire program
   (d) Basic blocks

B. Essay

1. Jelaskan complete code generation pipeline dari three-address code ke assembly!
2. Implementasikan code generator untuk bahasa sederhana dengan arithmetic expressions dan function calls!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan code generator untuk arsitektur target

☐ Saya dapat melakukan instruction selection yang efisien

☐ Saya dapat mengimplementasikan register allocation algorithms

☐ Saya dapat mengenerate code untuk function calls

☐ Saya dapat melakukan peephole optimizations

☐ Saya memahami perbedaan CISC dan RISC architectures

Rangkuman

Bab ini membahas code generation dan target machine, termasuk instruction selection, register allocation, target architectures, dan optimization techniques. Mahasiswa belajar membangun code generator yang efisien.

Poin Kunci:

- Code generation mengkonversi intermediate code ke target code
- Instruction selection memilih optimal target instructions
- Register allocation mengelola limited register resources
- Target architecture mempengaruhi generation strategy
- Peephole optimization mengoptimasi local instruction patterns
- Function calls memerlukan proper calling convention handling

Kata Kunci: ***Code Generation***, ***Instruction Selection***, ***Register Allocation***, ***Target Architecture***, ***x86***, ***RISC***, ***Peephole Optimization***, ***Calling Convention***