Bab 1

Code Generation untuk Target Architecture

1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami proses code generation dari intermediate representation ke target code
2. Menjelaskan konsep instruction selection dan implementasinya
3. Mengimplementasikan register allocation sederhana (local allocation)
4. Membuat code generator untuk operasi aritmatika dan assignment
5. Memahami dan mengimplementasikan calling convention untuk function calls
6. Menghasilkan assembly code yang valid untuk target architecture (x86 atau RISC-V)

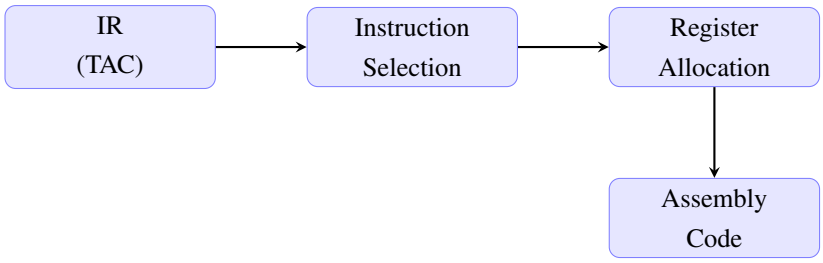
1.2 Pendahuluan

Code generation adalah fase terakhir dalam back-end kompilator yang bertanggung jawab untuk menghasilkan target code dari intermediate representation (IR) yang telah dioptimasi. Menurut sumber dari StudyLib:

> "Code generation: instruction selection, machine model. Implement code generation for a target architecture, mapping intermediate code into efficient target code, managing run-time structures."**?**

Code generator mengambil IR (biasanya dalam bentuk three-address code atau format serupa) dan menghasilkan kode assembly atau machine code yang dapat dieksekusi pada target architecture tertentu.

Gambar 1.1 menunjukkan proses code generation.



Gambar 1.1: Proses code generation

1.2.1 Tugas Code Generator

Code generator memiliki beberapa tugas utama:

1. **Instruction Selection**: Memilih instruksi machine yang tepat untuk setiap operasi IR
2. **Register Allocation**: Mengalokasikan register untuk variabel dan temporary values
3. **Instruction Scheduling**: Mengatur urutan instruksi untuk optimasi pipeline
4. **Address Assignment**: Mengalokasikan memory untuk variabel dan data structures
5. **Code Emission**: Menghasilkan assembly atau machine code dalam format yang sesuai

1.2.2 Input dan Output Code Generator

Input Code Generator:

- Optimized Intermediate Representation (TAC, quadruples, atau format IR lainnya)
- Symbol table dengan informasi tipe dan alamat variabel
- Target architecture specification (instruction set, register set, addressing modes)

Output Code Generator:

- Assembly code (untuk assembler) atau machine code langsung
- Relocation information (jika diperlukan)
- Debug information (optional)

1.3 Target Architecture

Target architecture adalah platform hardware yang menjadi tujuan kompilasi. Setiap architecture memiliki karakteristik yang berbeda yang mempengaruhi bagaimana code generator bekerja.

1.3.1 Karakteristik Target Architecture

Menurut dokumentasi LLVM[1], target architecture memiliki beberapa karakteristik penting:

1. **Instruction Set Architecture (ISA)**: Kumpulan instruksi yang didukung oleh processor
    - RISC (Reduced Instruction Set Computer): Instruksi sederhana, uniform, banyak register
    - CISC (Complex Instruction Set Computer): Instruksi kompleks, berbagai format, addressing modes yang kaya
2. **Register Set**: Jumlah dan jenis register yang tersedia
    - General-purpose registers
    - Special-purpose registers (stack pointer, frame pointer, dll.)
    - Floating-point registers
3. **Addressing Modes**: Cara mengakses operan (register, memory, immediate)
    - Register addressing: `ADD R1, R2`
    - Immediate addressing: `ADD R1, #42`
    - Memory addressing: `LOAD R1, [address]`
    - Indexed addressing: `LOAD R1, [R2 + offset]`
4. **Memory Model**: Bagaimana memory diorganisir dan diakses
    - Byte-addressable vs word-addressable
    - Alignment requirements
    - Endianness (little-endian vs big-endian)

1.3.2 Contoh Target Architecture

Dalam pembelajaran ini, kita akan fokus pada dua target architecture populer:

1. x86-64 (AMD64)

- CISC architecture dengan instruksi kompleks
- 16 general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15)
- Berbagai addressing modes

[1] https://llvm.org/docs/CodeGenerator.html

- Variable-length instructions
- Little-endian

2. RISC-V

- RISC architecture dengan instruksi sederhana
- 32 general-purpose registers (x0-x31)
- Fixed-length instructions (32-bit atau 16-bit untuk compressed)
- Simple addressing modes
- Little-endian atau big-endian (configurable)
- Open standard, populer untuk pembelajaran

Untuk pembelajaran, kita akan menggunakan subset sederhana dari RISC-V karena lebih mudah dipahami dan diimplementasikan.

1.4 Instruction Selection

Instruction selection adalah proses memilih instruksi machine yang tepat untuk mengimplementasikan setiap operasi dalam IR. Tujuannya adalah menghasilkan kode yang efisien dalam hal:

- Execution time (runtime performance)
- Code size
- Energy consumption

1.4.1 Metode Instruction Selection

Menurut Wikipedia[2], terdapat beberapa metode instruction selection:

1. Simple Translation

Metode paling sederhana: setiap operasi IR langsung dipetakan ke satu atau beberapa instruksi machine.

Contoh: TAC `t1 = t2 + t3` untuk RISC-V:

```
ADD t1, t2, t3
```

Kelebihan: Implementasi mudah, cepat Kekurangan: Tidak selalu optimal, tidak memanfaatkan instruksi kompleks

[2] https://en.wikipedia.org/wiki/Instruction_selection

2. Tree Pattern Matching

Menggunakan expression tree dan mencocokkan subtree dengan pattern instruksi machine.

Contoh: Ekspresi `a + b * c` dapat dicocokkan dengan pattern:

- Pattern 1: MUL + ADD (dua instruksi terpisah)
- Pattern 2: FMA (fused multiply-add, satu instruksi jika tersedia)

3. Dynamic Programming

Menggunakan dynamic programming untuk menemukan sequence instruksi yang optimal dengan mempertimbangkan cost.

1.4.2 Contoh Instruction Selection

Mari kita lihat contoh instruction selection untuk operasi sederhana pada RISC-V:

Contoh 1: Assignment

```
TAC: x = y
RISC-V: MV x, y    (atau ADD x, y, x0 untuk register zero)
```

Contoh 2: Arithmetic Operations

```
TAC: t1 = a + b
RISC-V: ADD t1, a, b

TAC: t2 = c * d
RISC-V: MUL t2, c, d

TAC: t3 = t1 - t2
RISC-V: SUB t3, t1, t2
```

Contoh 3: Load/Store

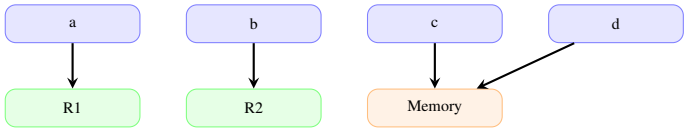
```
TAC: x = mem[addr]
RISC-V: LW x, 0(addr)     (load word)

TAC: mem[addr] = x
RISC-V: SW x, 0(addr)     (store word)
```

Contoh 4: Constant Loading

```
TAC: x = 42
RISC-V: LI x, 42          (load immediate, pseudo-instruction)
       atau
       ADDI x, x0, 42    (add immediate dengan zero register)
```

Gambar 1.2 menunjukkan konsep register allocation.



Gambar 1.2: Register allocation: variabel dialokasikan ke register atau memory

1.5 Register Allocation

Register allocation adalah proses menentukan variabel dan temporary values mana yang disimpan di register (fast storage) dan mana yang di-spill ke memory. Karena jumlah register fisik terbatas, ini adalah optimasi yang constrained.

1.5.1 Mengapa Register Allocation Penting?

Menurut Wikipedia[3]:

- Register jauh lebih cepat daripada memory access
- Jumlah register fisik terbatas (biasanya 16-32 register)
- Program mungkin memiliki lebih banyak variabel aktif daripada jumlah register
- Register allocation yang baik dapat meningkatkan performa secara signifikan

1.5.2 Dua Fase Register Allocation

1. Allocation Phase

Memutuskan *mana* nilai yang harus disimpan di register pada setiap program point. Ini melibatkan:

- Live range analysis: Kapan variabel hidup (live) dan kapan mati (dead)
- Interference graph: Grafik yang menunjukkan variabel mana yang tidak bisa menggunakan register yang sama secara bersamaan

2. Assignment Phase

Memetakan nilai yang dialokasikan ke register fisik spesifik. Jika lebih banyak nilai yang perlu register daripada register yang tersedia, beberapa nilai harus di-spill ke memory.

[3] https://en.wikipedia.org/wiki/Register_allocation

1.5.3 Local Register Allocation

Local register allocation bekerja dalam satu basic block (satu entry, satu exit, tidak ada branching). Ini lebih sederhana karena control flow linear.

Algoritma Simple Local Allocation

Algoritma sederhana untuk local allocation:

1. Scan basic block dari awal hingga akhir
2. Untuk setiap instruksi:
    - Jika operan tidak di register, load dari memory
    - Eksekusi operasi menggunakan register
    - Jika hasil perlu disimpan dan register penuh, spill register yang paling lama tidak digunakan
3. Di akhir block, store semua register yang modified ke memory

Contoh Local Allocation

Misalkan kita memiliki basic block dengan TAC berikut:

```
t1 = a + b
t2 = t1 * c
d = t2
```

Dengan 3 register tersedia (R1, R2, R3), allocation bisa seperti ini:

```
LOAD R1, a        ; Load a ke R1
LOAD R2, b        ; Load b ke R2
ADD R1, R1, R2    ; R1 = a + b (t1)
LOAD R2, c        ; Load c ke R2 (b tidak lagi diperlukan)
MUL R1, R1, R2    ; R1 = t1 * c (t2)
STORE R1, d       ; Store hasil ke d
```

1.5.4 Global Register Allocation

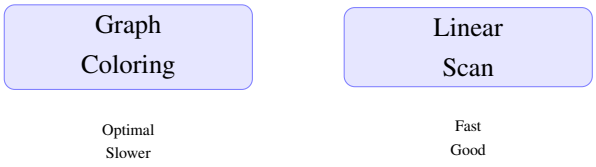
Global register allocation bekerja lintas basic blocks atau seluruh function. Ini lebih kompleks karena perlu mempertimbangkan control flow.

Metode populer:

- **Graph Coloring**: Membangun interference graph dan mewarnainya dengan k warna (k = jumlah register)

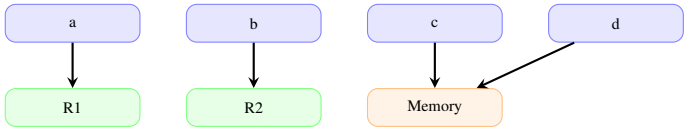
- **Linear Scan**: Lebih cepat, menghitung live intervals dan assign register dalam satu pass

Gambar 1.3 menunjukkan perbandingan metode register allocation.



Gambar 1.3: Perbandingan metode register allocation

Gambar 1.4 menunjukkan konsep register allocation.



Gambar 1.4: Register allocation: variabel dialokasikan ke register atau memory

Untuk pembelajaran, kita akan fokus pada local allocation yang lebih sederhana.

1.6 Code Generation untuk Operasi Aritmatika

Mari kita implementasikan code generator untuk operasi aritmatika dasar. Kita akan menggunakan RISC-V sebagai target architecture.

1.6.1 Struktur Code Generator Sederhana

Code generator sederhana dapat diimplementasikan sebagai visitor pattern yang traverse AST atau TAC dan menghasilkan assembly code.

Contoh Implementasi dalam C++

Berikut adalah struktur dasar code generator:

Listing 1.1: Struktur dasar Code Generator

```
class CodeGenerator {
private:
    std::vector<std::string> assemblyCode;
    int tempCounter;
    std::map<std::string, std::string> varToReg;
    std::set<std::string> availableRegs;
```

```

public:
    CodeGenerator() : tempCounter(0) {
        // Inisialisasi register yang tersedia
        for (int i = 1; i <= 8; i++) {
            availableRegs.insert("t" + std::to_string(i));
        }
    }

    std::string getRegister(const std::string& var) {
        // Alokasi register untuk variabel
        if (varToReg.find(var) != varToReg.end()) {
            return varToReg[var];
        }

        if (!availableRegs.empty()) {
            std::string reg = *availableRegs.begin();
            availableRegs.erase(reg);
            varToReg[var] = reg;
            return reg;
        }

        // Spill ke memory jika register penuh
        return spillRegister(var);
    }

    void generateAdd(const std::string& result,
                     const std::string& op1,
                     const std::string& op2) {
        std::string reg1 = getRegister(op1);
        std::string reg2 = getRegister(op2);
        std::string regResult = getRegister(result);

        assemblyCode.push_back("ADD " + regResult + ", " +
                               reg1 + ", " + reg2);
    }

    void generateMul(const std::string& result,
                     const std::string& op1,
                     const std::string& op2) {
        std::string reg1 = getRegister(op1);
        std::string reg2 = getRegister(op2);
        std::string regResult = getRegister(result);

        assemblyCode.push_back("MUL " + regResult + ", " +
                               reg1 + ", " + reg2);
    }

    void generateLoad(const std::string& reg,
                      const std::string& var) {
        assemblyCode.push_back("LW " + reg + ", " + var);
    }

    void generateStore(const std::string& var,
```

```
                        const std::string& reg) {
        assemblyCode.push_back("SW " + reg + ", " + var);
    }

    std::vector<std::string> getAssembly() {
        return assemblyCode;
    }
};
```

1.6.2 Generating Code untuk Ekspresi Kompleks

Untuk ekspresi kompleks seperti `a + b * c`, kita perlu mempertimbangkan precedence:

```
TAC untuk: result = a + b * c

t1 = b * c
t2 = a + t1
result = t2

Generated RISC-V code:
LW t1, a          ; Load a
LW t2, b          ; Load b
LW t3, c          ; Load c
MUL t4, t2, t3    ; t4 = b * c
ADD t5, t1, t4    ; t5 = a + t4
SW t5, result     ; Store result
```

1.7 Handling Function Calls dan Calling Convention

Function calls memerlukan koordinasi antara caller dan callee untuk:

- Passing arguments
- Saving/restoring registers
- Managing stack frame
- Returning values

Calling convention mendefinisikan aturan ini.

1.7.1 RISC-V Calling Convention

RISC-V memiliki calling convention yang didefinisikan dalam ABI (Application Binary Interface):

Register Usage

- **Argument Registers**: a0-a7 (x10-x17) untuk 8 argument pertama
- **Return Register**: a0 (x10) untuk return value
- **Caller-saved Registers**: t0-t6 (x5-x7, x28-x31) - caller harus save
- **Callee-saved Registers**: s0-s11 (x8-x9, x18-x27) - callee harus save
- **Stack Pointer**: sp (x2)
- **Frame Pointer**: fp/s0 (x8)

Function Call Sequence

Caller Side:

```
# Save caller-saved registers jika diperlukan
# Pass arguments ke a0-a7
# Call function
JAL ra, function_name
# Return value di a0
# Restore caller-saved registers
```

Callee Side (Function Prologue):

```
function_name:
    # Save return address dan frame pointer
    ADDI sp, sp, -frame_size
    SW ra, frame_size-4(sp)
    SW fp, frame_size-8(sp)
    ADDI fp, sp, frame_size

    # Save callee-saved registers yang digunakan
    # Allocate space untuk local variables
```

Callee Side (Function Epilogue):

```
    # Restore callee-saved registers
    # Put return value di a0
    # Restore frame pointer dan return address
    LW fp, frame_size-8(sp)
    LW ra, frame_size-4(sp)
    ADDI sp, sp, frame_size
    RET  # atau JALR x0, 0(ra)
```

1.7.2 Contoh Function Call

Mari kita lihat contoh function call untuk `int add(int a, int b)`:

Caller Code:

```
# Prepare arguments
LI a0, 10        # First argument (a = 10)
LI a1, 20        # Second argument (b = 20)

# Call function
JAL ra, add

# Result is in a0
# Use result...
```

Callee Code (add function):

```
add:
    # Function prologue
    ADDI sp, sp, -16    # Allocate stack frame
    SW ra, 12(sp)       # Save return address
    SW fp, 8(sp)        # Save frame pointer
    ADDI fp, sp, 16     # Set frame pointer

    # Function body
    ADD a0, a0, a1      # a0 = a0 + a1 (result)

    # Function epilogue
    LW fp, 8(sp)        # Restore frame pointer
    LW ra, 12(sp)       # Restore return address
    ADDI sp, sp, 16     # Deallocate stack frame
    RET                 # Return
```

1.8 Implementasi Code Generator Lengkap

Mari kita buat implementasi code generator yang lebih lengkap yang dapat menangani berbagai operasi.

1.8.1 Code Generator untuk TAC

Berikut adalah contoh code generator yang mengambil TAC dan menghasilkan RISC-V assembly:

Listing 1.2: Code Generator untuk TAC ke RISC-V

```
1 class TACCodeGenerator {
2 private:
3     std::vector<std::string> assembly;
4     int labelCounter;
```

```
    std::map<std::string, int> varOffset;  // Offset di stack frame
    int stackOffset;

public:
    TACCodeGenerator() : labelCounter(0), stackOffset(0) {}

    void generateTAC(const TACInstruction& tac) {
        switch (tac.op) {
            case TAC_OP::ADD:
                generateAdd(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::SUB:
                generateSub(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::MUL:
                generateMul(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::DIV:
                generateDiv(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::ASSIGN:
                generateAssign(tac.result, tac.arg1);
                break;
            case TAC_OP::LOAD:
                generateLoad(tac.result, tac.arg1);
                break;
            case TAC_OP::STORE:
                generateStore(tac.arg1, tac.result);
                break;
            // ... operasi lainnya
        }
    }

    void generateAdd(const std::string& result,
                     const std::string& arg1,
                     const std::string& arg2) {
        std::string reg1 = loadToRegister(arg1);
        std::string reg2 = loadToRegister(arg2);
        std::string regResult = allocateRegister(result);

        assembly.push_back("ADD " + regResult + ", " +
                           reg1 + ", " + reg2);

        releaseRegister(reg1);
        releaseRegister(reg2);
    }

    // ... implementasi operasi lainnya
};
```

1.9 Testing dan Validasi

Setelah code generator menghasilkan assembly, kita perlu:

1. **Assemble**: Mengubah assembly menjadi object file
2. **Link**: Menyatukan object files menjadi executable
3. **Run**: Mengeksekusi program dan memverifikasi hasilnya

1.9.1 Workflow Lengkap

```
Source Code (C/C++)
    ↓
[Compiler Front-end]
    ↓
TAC / IR
    ↓
[Code Generator] → Assembly Code (.s)
    ↓
[Assembler] → Object File (.o)
    ↓
[Linker] → Executable
    ↓
[Run] → Verify Output
```

1.9.2 Contoh Testing

Misalkan kita memiliki program sederhana:

```
int main() {
    int a = 10;
    int b = 20;
    int c = a + b;
    return c;
}
```

TAC yang dihasilkan:

```
t1 = 10
a = t1
t2 = 20
b = t2
t3 = a + b
c = t3
return c
```

Assembly yang dihasilkan (RISC-V):

```
main:
    ADDI sp, sp, -16
    SW ra, 12(sp)
    SW fp, 8(sp)
    ADDI fp, sp, 16

    # a = 10
    LI t0, 10
    SW t0, -4(fp)    # Store a di stack

    # b = 20
    LI t0, 20
    SW t0, -8(fp)    # Store b di stack

    # c = a + b
    LW t1, -4(fp)    # Load a
    LW t2, -8(fp)    # Load b
    ADD t0, t1, t2
    SW t0, -12(fp)   # Store c

    # return c
    LW a0, -12(fp)   # Return value

    LW fp, 8(sp)
    LW ra, 12(sp)
    ADDI sp, sp, 16
    RET
```

1.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Code generation adalah fase terakhir yang mengubah IR menjadi target code
2. Instruction selection memilih instruksi machine yang tepat untuk setiap operasi IR
3. Register allocation menentukan variabel mana yang disimpan di register vs memory
4. Local register allocation lebih sederhana dan cocok untuk pembelajaran awal
5. Calling convention mengatur bagaimana function calls dilakukan
6. Code generator harus menghasilkan assembly yang valid dan dapat di-assemble, link, dan run

Implementasi code generator yang baik memerlukan pemahaman mendalam tentang target architecture dan trade-off antara code size, execution time, dan register pressure.

1.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang code generation, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* **?** - Bab 8: Code Generation
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 7-9: Code Shape, Introduction to Optimization, Scalar Optimizations
- **RISC-V Instruction Set Manual**: https://riscv.org/technical/specifications/ - Dokumentasi lengkap instruksi RISC-V
- **LLVM Code Generator Documentation**: https://llvm.org/docs/CodeGenerator.html - Dokumentasi code generator LLVM
- **StudyLib - Outcomes-Based Education**: Materials tentang code generation dan runtime structures **?**
- **Wikipedia - Register Allocation**: https://en.wikipedia.org/wiki/Register_allocation - Artikel tentang teknik register allocation
- **Wikipedia - Instruction Selection**: https://en.wikipedia.org/wiki/Instruction_selection - Artikel tentang instruction selection