

Chapter-08

Code Generation

Code Generation



Position of code generator

Issues in the Design of a Code Generator

While the details are dependent on the specifics of

- the intermediate representation,
- the target language and
- the run-time system

tasks such as

- instruction selection,
- register allocation and assignment and
- instruction ordering

are encountered in the design of almost all code generators

Issues in the Design of a Code Generator (Cont...)

- The most important criterion for a code generator is that it produces correct code.
- Correctness takes on special significance because of the number of special cases that a code generator might face.
- Given the premium on correctness, designing a code generator so it can be easily implemented, tested and maintained is an important design goal.

Input to the Code Generator

- The input to the code generator is
 - The **intermediate representation (IR)** of the source program produced by the front end
 - Along with information in the **symbol table** that is used to determine the run-time addresses of the data objects denoted by the names in the IR
- The many choices for the IR include
 - Three-address representations such as quadruples, triples, indirect triples
 - Virtual machine representations such as byte codes and stack-machine code
 - Linear representations such as postfix notation and
 - Graphical representations such as syntax trees and DAG's

Input to the Code Generator (Cont...)

- We assume that the front end has
 - scanned,
 - parsed and
 - translated the source program into a relatively low-level IR

so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate such as integers and floating-point numbers

Input to the Code Generator (Cont...)

- We also assume that
 - all syntactic and static semantic errors have been detected
 - the necessary type checking has taken place and
 - type conversion operators have been inserted wherever necessary
- The code generator can therefore proceed on the assumption that its input is free of these kinds of errors

The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code
- The most common target-machine architectures are
 - RISC (reduced instruction set computer),
 - CISC (complex instruction set computer), and
 - stack based

The Target Program (Cont...)

- A RISC machine typically has
 - ✓ Many registers
 - ✓ Three-address instructions
 - ✓ Simple addressing modes and
 - ✓ Relatively simple instruction-set architecture

The Target Program (Cont...)

- In contrast, a CISC machine typically has
 - ✓ Few registers,
 - ✓ Two-address instructions,
 - ✓ A variety of addressing modes,
 - ✓ Several register classes,
 - ✓ Variable-length instructions and
 - ✓ Instructions with side effects

The Target Program (Cont...)

- In a stack-based machine operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack
- To achieve high performance the top of the stack is typically kept in registers
- Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many copy and swap operations.

The Target Program (Cont...)

- We shall use a very simple RISC-like computer as our target machine
- We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines
- For readability, we use assembly code as the target language
- As long as addresses can be calculated from offsets and other information stored in the symbol table the code generator can produce absolute addresses for names

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine
- The complexity of performing this mapping is determined by a factors such as
 - the level of the IR,
 - the nature of the instruction-set architecture,
 - the desired quality of the generated code

Instruction Selection (Cont...)

- If the IR is high level the code generator may translate each IR statement into a sequence of machine instructions using code templates
- Such statement-by-statement code generation, however, often produces poor code that needs further optimization
- If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences

Instruction Selection (Cont...)

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection
- For example, the uniformity and completeness of the instruction set are important factors
- If the target machine does not support each data type in a uniform manner then each exception to the general rule requires special handling
- On some machines, for example, floating-point operations are done using separate registers

Instruction Selection (Cont...)

- Instruction speed is another important factor
- If we do not care about the efficiency of the target program then instruction selection is straightforward
- For each type of three-address statement we can design a code skeleton that defines the target code to be generated for that construct

Instruction Selection (Cont...)

- For example, every three-address statement of the form $x = y + z$, where x , y and z are statically allocated, can be translated into the code sequence

```
LD R0, y           // R0 = y (load y into register R0)
```

```
LD R1, z           // R1 = z (load z into register R1)
```

```
ADD R0, R0, R1      // R0 = R0 + R1 (add R1 to R0)
```

```
ST x, R0            // x = R0 (store R0 into x)
```

Instruction Selection (Cont...)

- This strategy often produces redundant loads and stores
- For example, the sequence of three-address statements

$a = b + c$

$d = a + e$

would be translated into

Here, the fifth statement is redundant since it loads a value that has just been stored.

LD R0, b	// R0 = b
LD R1, c	// R1 = c
ADD R0, R0, R1	// R0 = R0 + R1
ST a, R0	// a = R0
LD R0, a	// R0 = a
LD R1, e	// R1 = e
ADD R0, R0, R1	// R0 = R0 + R1
ST d, R0	// d = R0

Instruction Selection (Cont...)

- The quality of the generated code is usually determined by its **speed** and **size**
- On most machines, a given IR program can be implemented by many different code sequences with significant cost differences between the different implementations
- A **naive translation** of the intermediate code may therefore lead to **correct** but **unacceptably inefficient** target code
- We need to know instruction costs in order to design good code sequences but unfortunately accurate cost information is often difficult to obtain

Instruction Selection (Cont...)

- For example, if the target machine has an “increment” instruction (INC) then the three-address statement $a = a + 1$ may be implemented more efficiently by the single instruction **INC a**
- Rather than by a more obvious sequence that loads a into a register, adds one to the register and then stores the result back into a:

LD R0, a // R0 = a

ADD R0, R0, #1 // R0 = R0 + 1

ST a, R0 // a = R0

The Target Language

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator
- We shall use as a target language assembly code for a simple computer that is representative of many register machines

A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations and conditional jumps
- The underlying computer is a byte-addressable machine with n general-purpose registers $R0, R1, \dots, R_{n-1}$
- A full-fledged assembly language would have scores of instructions
- We use a very limited set of instructions and assume that all operands are integers
- Most instructions consists of an operator followed by a target followed by a list of source operands
- A label may precede an instruction

A Simple Target Machine Model (Cont...)

We assume the following kinds of **instructions** are available:

- **Load Operations:** The instruction **LD dst, addr** loads the value in location addr into location dst
 - This instruction denotes the assignment **dst = addr**
 - The most common form of this instruction is **LD r, x** which loads the value in location x into register r
 - An instruction of the form **LD r1, r2** is a register-to-register copy in which the contents of register r2 are copied into register r1

A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Store Operations:** The instruction **ST x, r** stores the value in register r into the location x
 - This instruction denotes the assignment **x = r**

A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Computation Operations** of the form **OP dst, src1, src2** where OP is a operator like ADD or SUB and dst, src1, and src2 are locations, not necessarily distinct.
 - The effect of this machine instruction is to apply the operation represented by OP to the values in locations src1 and src2 and place the result of this operation in location dst
 - For example, **SUB r1, r2, r3** computes **$r1 = r2 - r3$**
 - Any value formerly stored in r1 is lost but if r1 is r2 or r3, the old value is read first
 - Unary operators that take only one operand do not have a src2

A Simple Target Machine Model (Cont...)

We assume the following kinds of instructions are available:

- **Unconditional jumps:** The instruction **BR L** causes control to branch to the machine instruction with label L (BR stands for branch)

A Simple Target Machine Model(Cont...)

We assume the following kinds of instructions are available:

- **Conditional jumps** of the form **Bcond r, L** where r is a register, L is a label and cond stands for any of the common tests on values in the register r
 - For example, **BLTZ r, L** causes a jump to label L if the value in register r is less than zero and allows control to pass to the next machine instruction if not.

A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of **addressing modes**:

- In instructions, a location can be a **variable name** x referring to the memory location that is reserved for x (that is l-value of x)
- A location can also be an indexed address of the form $a(r)$ where a is a variable and r is a register
- The memory location denoted by $a(r)$ is computed by taking the l-value of a and adding to it the value in register r

A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- For example, the instruction **LD R1, a(R2)** has the effect of setting **R1 = contents(a + contents(R2))** where **contents(x)** denotes the contents of the register or memory location represented by x
- This addressing mode is useful for accessing arrays, where a is the base address of the array (that is the address of the first element) and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- A memory location can be an **integer indexed by a register**
- For example, **LD R1, 100(R2)** has the effect of setting **R1 = contents(100+contents(R2))** that is of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2

A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- We also allow two **indirect addressing modes**
- ***r** means the memory location found in the location represented by the contents of register r and
- ***100(r)** means the memory location found in the location obtained by adding 100 to the contents of r
- For example, **LD R1, *100(R2)** has the effect of setting **R1 = contents(contents(100+contents(R2)))** that is of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2

A Simple Target Machine Model (Cont...)

We assume our target machine has a variety of addressing modes:

- Finally, we allow an **immediate constant addressing mode**
- The constant is prefixed by #
- The instruction **LD R1, #100** loads the integer 100 into register R1 and
- **ADD R1, R1, #100** adds the integer 100 into register R1
- Comments at the end of instructions are preceded by //

Example 8.2

- The three-address statement $x = y - z$ can be implemented by the machine instructions:

LD R1, y	// R1 = y
LD R2, z	// R2 = z
SUB R1, R1, R2	// R1 = R1 - R2
ST x, R1	// x = R1

- One of the goals of a good code-generation algorithm is to avoid using all four of these instructions whenever possible
- For example, y and/or z may have been computed in a register and if so we can avoid the LD step(s)
- Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed

Example 8.2 (Cont...)

- Suppose **a** is an array whose elements are 8-byte values
- Also assume elements of **a** are indexed starting at 0
- We may execute the three-address instruction **b = a [i]** by the machine instructions:

LD R1, i	// R1 = i
MUL R1, R1, 8	// R1 = R1 * 8
LD R2, a(R1)	// R2 = contents (a + contents (R1))
ST b, R2	// b = R2

Example 8.2(Cont...)

- Similarly, the assignment into the array `a` represented by three-address instruction `a[j] = c` is implemented by:

LD R1, c	// R1 = c
LD R2, j	// R2 = j
MUL R2, R2, 8	// R2 = R2 * 8
ST a(R2), R1	// contents (a + contents (R2)) = R1

Example 8.2 (Cont...)

- To implement a simple pointer indirection, such as the three-address statement $\mathbf{x} = *p$, we can use machine instructions like:

LD R1, p // R1 = p

LD R2, 0(R1) // R2 = contents (0 + contents (R1))

ST x, R2 // x = R2

Example 8.2 (Cont...)

- The assignment through a pointer $*p = y$ is similarly implemented in machine code by:

LD R1, p **// R1 = p**

LD R2, y **// R2 = y**

ST 0(R1), R2 **// contents (0 + contents (R1)) = R2**

Example 8.2(Cont...)

- A conditional jump three-address instruction like **if $x < y$ goto L**
- The machine-code equivalent would be something like:

LD R1, x	// R1 = x
LD R2, y	// R2 = y
SUB R1, R1, R2	// R1 = R1 - R2
BLTZ R1, M	// if R1 < 0 jump to M

- Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L. As for any three-address instruction we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored.

Program and Instruction Costs

- We often **associate a cost** with compiling and running a program
- Depending on what aspect of a program we are interested in optimizing, some common cost measures are the **length of compilation time and the size, running time and power consumption** of the target program
- Determining the actual cost of compiling and running a program is a complex problem
- Finding an optimal target program for a given source program is an undecidable problem in general and many of the sub problems involved are **NP-hard**.
- As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

Program and Instruction Costs (Cont...)

- We shall **assume** each target-language instruction has an associated cost
- For simplicity, we take the **cost of an instruction** to be **one plus** the costs associated with the **addressing modes of the operands**
- This cost corresponds to the length in words of the instruction
- **Addressing modes involving registers have zero additional cost and**
- **Modes involving** a **memory location** or **constant** in them have an additional cost of one because such operands have to be stored in the words following the instruction.

Program and Instruction Costs (Cont...)

Some examples:

▪ **LD R0, R1**

- The instruction **LD R0, R1** copies the contents of register R1 into register R0
- This instruction has a **cost of one** because no additional memory words are required

▪ **LD R0, M**

- The instruction **LD R0, M** loads the contents of memory location M into register R0
- The **cost is two** since the address of memory location M is in the word following the instruction

Program and Instruction Costs (Cont...)

Some examples:

▪ **LD R1, *100(R2)**

- The instruction **LD R1, *100(R2)** loads into register R1 the value given by **contents(contents(100+contents(R2)))**
 - The **cost is three** because the constant 100 is stored in the word following the instruction
- ## ▪ Cost of a target-language program on a given input is
- the sum of costs of the individual instructions executed when the program is run on that input
- Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs.

Addresses in the Target Code

- **Names** in the **IR** can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation.
- Each executing program runs in its own logical address space that was partitioned into **four code and data areas**:
 - ✓ A statically determined area **Code** that holds the executable target code. The size of the target code can be determined at compile time.
 - ✓ A statically determined **data area Static** for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

Addresses in the Target Code (Cont...)

- ✓ A dynamically managed area **Heap** for holding data objects that are allocated and freed during program execution. The size of the Heap cannot be determined at compile time.
- ✓ A dynamically managed area **Stack** for holding activation records as they are created and destroyed during procedure calls and returns. Like the Heap, the size of the Stack cannot be determined at compile time.

Static Allocation

- To illustrate code generation for simplified procedure calls and returns we shall focus on the following three-address statements:
 - **call callee**
 - **return**
 - **halt**
 - **action**, which is a placeholder for other three-address statements

Static Allocation (Cont...)

- The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table
- We shall first illustrate how to store the return address in an activation record on a procedure call and how to return control to it after the procedure call
- For convenience, we assume the first location in the activation holds the return address.

Static Allocation (Cont...)

- Let us consider the code needed to implement the simplest case static allocation
- Here, a **call callee** statement in the intermediate code can be implemented by a sequence of two target-machine instructions:
 ST callee.staticArea, #here + 20
 BR callee.codeArea
- The **ST** instruction saves the return address at the beginning of the activation record for callee and
- The **BR** transfers control to the target code for the called procedure callee

Static Allocation (Cont...)

ST callee.staticArea, #here + 20

BR callee.codeArea

- The attribute **callee.staticArea** is a constant that gives the address of the beginning of the activation record for callee and
- The attribute **callee.codeArea** is a constant referring to the address of the first instruction of the called procedure callee in the Code area of the run-time memory.
- The operand **#here + 20** in the ST instruction is the literal return address
- It is the address of the instruction following the BR instruction

Static Allocation (Cont...)

ST callee.staticArea, #here + 20

BR callee.codeArea

- We assume that **#here** is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of **5 words or 20 bytes**
- The code for a procedure ends with a return to the calling procedure
- Except that the first procedure has no caller, so its final instruction is **HALT** which returns control to the operating system
- A **return callee** statement can be implemented by a simple jump instruction **BR *callee.staticArea** which transfers control to the address saved at the beginning of the activation record for callee.

Example 8.3

Suppose we have the following three-address code:

```
// code for c
```

```
    action1
```

```
    call p
```

```
    action2
```

```
    halt
```

```
// code for p
```

```
    action3
```

```
    return
```

Example 8.3 (Cont...)

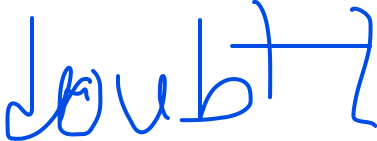
- We use the pseudo-instruction **ACTION** to represent the sequence of machine instructions to execute the statement action
- This represents three-address code that is not relevant for this discussion
- We arbitrarily start the **code** for **procedure c** at address **100** and for **procedure p** at address **200**
- We assume that each **ACTION** instruction takes **20 bytes**
- We further assume that the **activation records** for these procedures are **statically allocated** starting at locations **300** and **364** respectively

Example 8.3 (Cont...)

// code for c		
action1		
call p		
action2		
halt		
// code for p		
action3		
return		

// code area for c:		
100:	ACTION1	// code for activation
120:	ST 364, #140	// save return address 140 in location 364
132:	BR 200	// call p
140:	ACTION2	// code for activation
160:	HALT	// return to operating system

Example 8.3(Cont...)

	// code area for p:	
// code for c		// code for activation
action1	200: ACTION3	
call p	220: BR *364	// return to address saved in location 364
action2		
halt	// static area for c:	// 300-363 hold activation record for c
// code for p	300:	// return address
action3	304:	// local data for c
return		
	// static area for p:	// 364-451 hold activation record for p
	364: 140	// return address
	368:	// local data for p

Example 8.3 (Cont...)

// code for <i>c</i>	100: ACTION ₁	// code for <i>c</i>
action1	120: ST 364, #140	// code for action ₁
call p	132: BR 200	// save return address 140 in location 364
action2	140: ACTION ₂	// call <i>p</i>
halt	160: HALT	// return to operating system
	...	
	200: ACTION ₃	// code for <i>p</i>
// code for <i>p</i>	220: BR *364	// return to address saved in location 364
action3	...	
return	300:	// 300-363 hold activation record for <i>c</i>
	304:	// return address
	...	// local data for <i>c</i>
		// 364-451 hold activation record for <i>p</i>
	364:	// return address
	368:	// local data for <i>p</i>

Stack Allocation

- **Static** allocation can become **stack** allocation by using **relative addresses** for storage in activation records.
- In stack allocation the position of an activation record for a procedure is not known until run time.
- This **position** is usually stored in a **register (SP)**, so words in the activation record can be accessed as offsets from the value in this register.
- The **indexed address mode** of our target machine is convenient for this purpose.

Stack Allocation (cont..)

- When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
- After control returns to the caller, we decrement SP, thereby deallocating the activation record of the called procedure.
- **Step-1:** First procedure initializes the stack by setting SP to the start of the stack area in memory:

```
LD SP, #stackStart    // initialize the stack
// code for the first procedure
HALT                   // terminate execution
```


Stack Allocation (cont..)

- **Step-2:** A procedure call (**call callee**) sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD SP, SP, #caller.recordSize    // increment stack pointer
ST 0(SP), #here + 16              // save return address
BR callee.codeArea                // jump to the callee
```

Stack Allocation (cont..)

- **Step-3:** The return (**return**) sequence consists of two parts.
 - Part-1: The called procedure transfers control to the return address using

BR *0(SP) // return to caller

- Part-2: The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

SUB SP, SP, #caller.recordSize // decrement stack pointer

Stack Allocation (cont..) Example-8.4

- Consider the pseudo code of quicksort again.
- Suppose that the sizes of the activation records for procedures **m**, **p**, and **q** have been determined to be **msize**, **psize**, and **qsize**, respectively.
- The **first word** in each **activation** record will hold a **return address**.
- We arbitrarily assume that the code for these **procedures starts** at addresses 100, 200, and 300, respectively, and that the **stack** starts at address **600**.
- We assume that ACTION4 contains a conditional jump to the address 456 of the return sequence from **q**; otherwise, the recursive procedure **q** is condemned to call itself forever.
- Let **msize**, **psize**, and **qsize** be 20, 40, and 60, respectively

Stack Allocation (cont..) Example-8.4

```
// code for m
```

```
action1
```

```
call q
```

```
action2
```

```
halt
```

```
// code for p
```

```
action3
```

```
return
```

```
// code for q
```

```
action4
```

```
call p
```

```
action5
```

```
call q
```

```
action6
```

```
call q
```

```
return
```

Stack Allocation (cont.)

Example-8.4

```
// code for m
action1
call q
action2
halt
// code for p
action3
return
```

```
// code for q
action4
call p
action5
call q
action6
call q
return
```

```
100: LD SP, #600           // code for m
108: ACTION1              // initialize the stack
128: ADD SP, SP, #msize    // code for action1
136: ST 0(SP), #152        // call sequence begins
144: BR 300                // push return address
152: SUB SP, SP, #msize    // call q
160: ACTION2              // restore SP
180: HALT
...
...
200: ACTION3              // code for p
220: BR *0(SP)             // return
...
...
300: ACTION4              // code for q
320: ADD SP, SP, #qsize    // contains a conditional jump to 456
328: ST 0(SP), #344        // push return address
336: BR 200                // call p
344: SUB SP, SP, #qsize
352: ACTION5
372: ADD SP, SP, #qsize
380: ST 0(SP), #396        // push return address
388: BR 300                // call q
396: SUB SP, SP, #qsize
404: ACTION6
424: ADD SP, SP, #qsize
432: ST 0(SP), #440        // push return address
440: BR 300                // call q
448: SUB SP, SP, #qsize
456: BR *0(SP)             // return
...
600: ...                   // stack starts here
```

Basic Blocks and Flow Graphs

Basic Blocks and Flow Graphs

The representation is constructed as follows:

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the **properties** that
 - a) The **flow of control** can only **enter** the basic block through the **first instruction** in the block. That is, there are no jumps into the middle of the block.
 - b) **Control** will **leave** the block without **halting or branching**, except possibly at the last instruction in the block.
2. The **basic blocks** become the **nodes** of a **flow graph**, whose edges indicate which blocks can follow which other blocks.

Basic Blocks

- Our first job is to partition a sequence of three-address instructions into basic blocks.

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: *A sequence of three-address instructions.*

OUTPUT: *A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.*

METHOD:

First, we determine those instructions in the intermediate code that are leaders, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

Basic Blocks

- The rules for finding leaders are:
 1. The **first three-address instruction** in the intermediate code is a **leader**.
 2. Any **instruction** that is the **target** of a conditional or unconditional jump is a **leader**.
 3. Any instruction that **immediately follows** a conditional or unconditional jump is a leader.

Basic Blocks (Figure. 8.7)

- The **leaders** are instructions 1, 2, 3, 10, 12, and 13.
- The basic block of each leader contains all the instructions from itself until just before the next leader.
- The **basic block** of 1 is just 1, for leader 2 the block is just 2.
- Leader 3 has a basic block consisting of instructions 3 through 9, inclusive.
- Instruction 10's block is 10 and 11
- Instruction 12's block is just 12, and
- Instruction 13's block is 13 through 17.

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

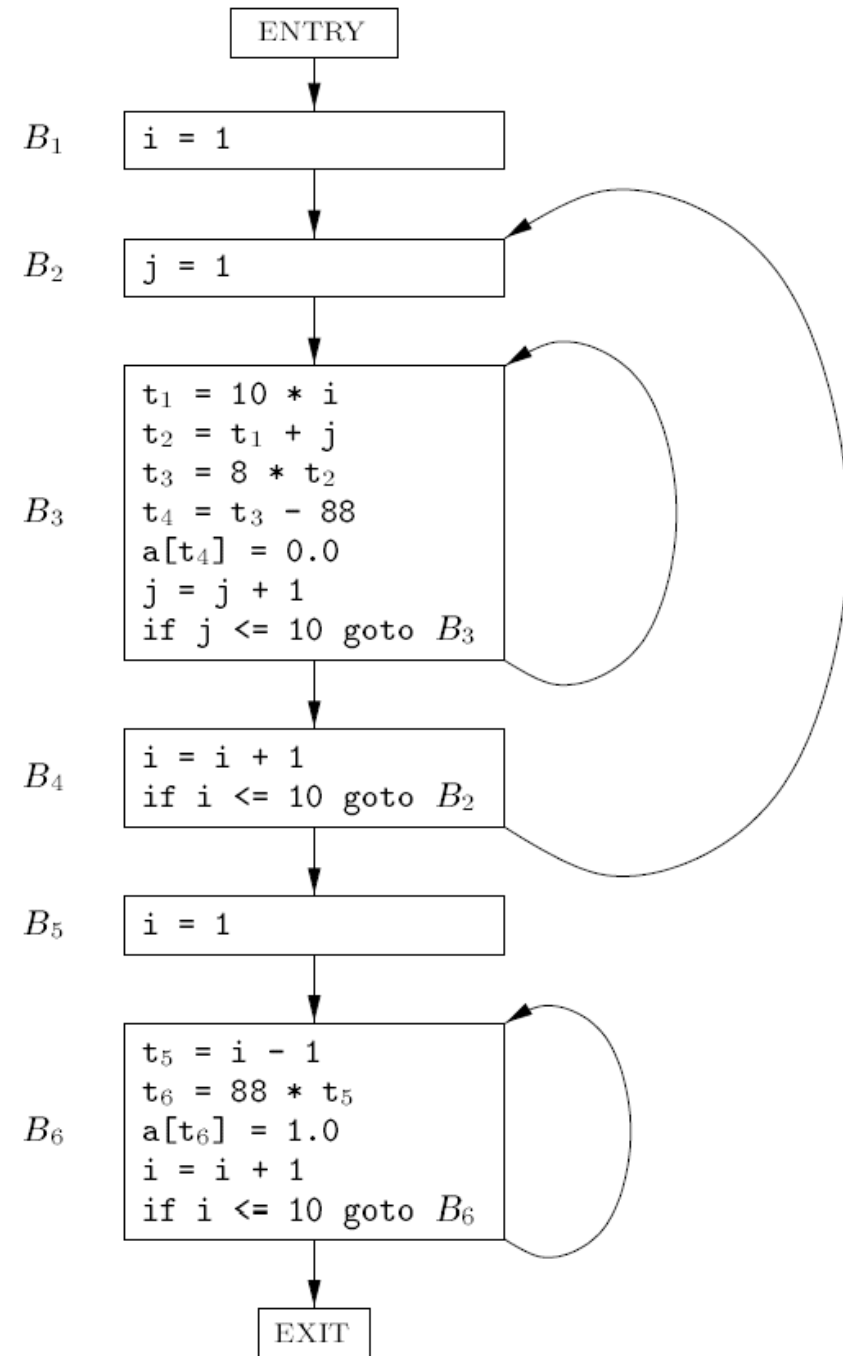
Figure. 8.7

Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The **nodes** of the flow graph are the **basic blocks**.
- There is an **edge** from block **B to block C** if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.
- There are **two ways** that such an **edge could be justified**:
 - There is a conditional or unconditional jump from the end of B to the beginning of C.
 - C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

Flow Graphs

Figure 8.9: Flow graph from Fig. 8.7



Summary

- Code generation is the final phase of a compiler.
- Instruction selection
- Register allocation is the process of deciding which IR values to keep in registers. Register assignment is the process of deciding which register should hold a given IR value.
- CISC and RISC machine intro
- Target machine model with instruction and program cost
- Address in the target code using static and stack allocation
- A basic block is a maximal sequence of consecutive three-address statements in which flow of control can only enter at the first statement of the block and leave at the last statement without halting or branching except possibly at the last statement in the basic block.
- Flow graph is a graphical representation of a program in which the nodes of the graph are basic blocks and the edges of the graph show how control can flow among the blocks.