Lecture 28-29

Code Generation

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

April 10, 2025

Code Generation

- output code must be correct

Code Generation

- output code must be correct
- output code must be of high quality

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader
  - assembly code requires assembler, linker, and loader

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader
  - assembly code requires assembler, linker, and loader
- Steps involved

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader
  - assembly code requires assembler, linker, and loader
- Steps involved
  - Instruction Selection

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader
  - assembly code requires assembler, linker, and loader
- Steps involved
  - Instruction Selection
  - Register Allocation and assignment

Code Generation

- output code must be correct
- output code must be of high quality
- code generator should run efficiently
- Input: Intermediate representation with symbol table assume that input has been validated by the front end
- Target programs :
  - absolute machine language fast for small programs
  - relocatable machine code requires linker and loader
  - assembly code requires assembler, linker, and loader
- Steps involved
  - Instruction Selection
  - Register Allocation and assignment
  - Evaluation order

Target Machine

- Word size

Target Machine

- Word size
- Registers

Target Machine

- Word size
- Registers
- Opcodes

Target Machine

- Word size
- Registers
- Opcodes
- Addressing mode

Sample Target code

a = b + c
d = a + e
	Mov b, R0
	Add c, R0
	Mov R0, a
	Mov a, R0
	Add e, R0
	Mov R0, d
		Can be removed

Sample Target code

a = b + c
d = a + e
	Mov b, R0
	Add c, R0
	Mov R0, a
	Mov a, R0
	Add e, R0
	Mov R0, d
		Can be removed

a = a + 1	Mov a, R0
	Add #1, R0
	Mov R0, a
		inc a

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader
  - first statement is a leader

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader
  - any statement that follows a goto statement is a leader

Basic Block

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader
  - any statement that follows a goto statement is a leader
- for each leader its basic block consists of the leader and all statements up to next leader

Flow Graphs

- Add control flow information to basic blocks

Flow Graphs

- Add control flow information to basic blocks
- Nodes are the basic blocks

Flow Graphs

- Add control flow information to basic blocks
- Nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence

Flow Graphs

- Add control flow information to basic blocks
- Nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence
  - there is a jump from the last statement of B_1 to the first statement of B_2

Flow Graphs

- Add control flow information to basic blocks
- Nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence
  - there is a jump from the last statement of B_1 to the first statement of B_2
  - B_2 follows B_1 in natural order of execution

Flow Graphs

- Add control flow information to basic blocks
- Nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence
  - there is a jump from the last statement of B_1 to the first statement of B_2
  - B_2 follows B_1 in natural order of execution
- initial node: block with first statement as leader

Next use information

- for register and temporary allocation

Next use information

- for register and temporary allocation
- remove variables from registers if not used

Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement `X = Y op Z` defines X and uses Y and Z

Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement `X = Y op Z` defines X and uses Y and Z
- scan each basic blocks backwards

Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement `X = Y op Z` defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

- Suppose we are scanning `i : X = Y op Z` in backward scan

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

- Suppose we are scanning `i : X = Y op Z` in backward scan
  - attach to i, information in symbol table about X,Y, Z

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

- Suppose we are scanning `i : X = Y op Z` in backward scan
  - attach to i, information in symbol table about X,Y, Z
  - set X to not live and no next use in symbol table

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

- Suppose we are scanning `i : X = Y op Z` in backward scan
  - attach to i, information in symbol table about X,Y, Z
  - set X to not live and no next use in symbol table
  - set Y and Z to be live and next use in i in symbol table

Algorithm to compute next use information

use of a var

- Statement `i` assign value to `x`
- Statment `j` has `x` as an operand
- Control can flow from `i` to `j` and no assignment of `x` in between
- `j` uses value of `x` computed at `i`
- `x` is live at `i`

- Suppose we are scanning `i : X = Y op Z` in backward scan
  - attach to i, information in symbol table about X,Y, Z
  - set X to not live and no next use in symbol table
  - set Y and Z to be live and next use in i in symbol table

Code Generator

- Consider each statement

Code Generator

- Consider each statement
- Remember if operand is in a register

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor
  - Keep track of what is currently in each register.

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor
  - Keep track of what is currently in each register.
  - Initially all the registers are empty

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- Address descriptor

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- Address descriptor
  - Keep track of location where current value of the name can be found at runtime

Code Generator

- Consider each statement
- Remember if operand is in a register
- Register descriptor
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- Address descriptor
  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

Code Generation Algorithm

for each $X = Y + Z$ do

Code Generation Algorithm

for each X = Y + Z do

- Use `getReg(x = y + z)` to select registers for `x`, `y` and `z`. Call these R_x, R_y and R_z
- If `y` is not in R_y issue `LD` R_y y'
- Similarly for `z`
- Issue `ADD` R_x, R_y, R_z

What about copy statement (`X=Y`)?

Code Generation Algorithm

for each X = Y + Z do

- Use `getReg(x = y + z)` to select registers for `x`, `y` and `z`. Call these R_x, R_y and R_z
- If `y` is not in R_y issue `LD` R_y y'
- Similarly for `z`
- Issue `ADD` R_x, R_y, R_z

What about copy statement (`X=Y`)?
What happen at the end of a block?

Managing Register and Address Descriptors

- For instruction `LD R,x`

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`
  - Change address descriptor of `x` so that its only location is R_x

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`
  - Change address descriptor of `x` so that its only location is R_x
  - Remove R_x from address descriptor of any other variable other than `x`

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`
  - Change address descriptor of `x` so that its only location is R_x
  - Remove R_x from address descriptor of any other variable other than `x`
- For instruction `x = y`

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`
  - Change address descriptor of `x` so that its only location is R_x
  - Remove R_x from address descriptor of any other variable other than `x`
- For instruction `x = y`
  - Add `x` to register description for R_y

Managing Register and Address Descriptors

- For instruction `LD R,x`
  - Update register descriptor for `R` so it holds only `x`
  - Update address descriptor of `x` by adding `R` as an additional location.
- `ST x, R` change address descriptor of `x` to include its own memory location.
- For `ADD` R_x, R_y, R_z
  - Change register descriptor of R_x so it holds only `x`
  - Change address descriptor of `x` so that its only location is R_x
  - Remove R_x from address descriptor of any other variable other than `x`
- For instruction `x = y`
  - Add `x` to register description for R_y
  - Change the address descriptor of `x` so that its only location is R_y

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If y is in register, pick that register as R_y. Don't issue a load instruction

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If y is in register, pick that register as R_y. Don't issue a load instruction
  - if y is not in register, but there is an empty register pick one such register as R_y

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If y is in register, pick that register as R_y. Don't issue a load instruction
  - if y is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - ⋆ If v is somewhere else also. We are OK

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If y is in register, pick that register as R_y. Don't issue a load instruction
  - if y is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If y is in register, pick that register as R_y. Don't issue a load instruction
  - if y is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK
    - Otherwise, we are not OK. Create an instruction `ST v,R`. This process called spill. R may hold many variables so we have to store all of them

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK
    - Otherwise, we are not OK. Create an instruction `ST v,R`. This process called spill. R may hold many variables so we have to store all of them
- Selection of R_x

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK
    - Otherwise, we are not OK. Create an instruction `ST v,R`. This process called spill. R may hold many variables so we have to store all of them
- Selection of R_x
  - Pick a register which is currently holding only x

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK
    - Otherwise, we are not OK. Create an instruction `ST v,R`. This process called spill. R may hold many variables so we have to store all of them
- Selection of R_x
  - Pick a register which is currently holding only x
  - If either of R_y or R_z if anyone of the is not live

Function getReg(I)

`x = y + z`

- Steps for picking R_y for y
  - If `y` is in register, pick that register as R_y. Don't issue a load instruction
  - if `y` is not in register, but there is an empty register pick one such register as R_y
  - We have to pick a register (say R) that is currently holding a variable (say v)
    - If v is somewhere else also. We are OK
    - if v is x, we are OK
    - If v is not used later, we are OK
    - Otherwise, we are not OK. Create an instruction `ST v,R`. This process called spill. R may hold many variables so we have to store all of them
- Selection of R_x
  - Pick a register which is currently holding only x
  - If either of R_y or R_z if anyone of the is not live
  - otherwise follow the steps similar to R_y

Conditional Expression

- Branch if value of `R` meets one of six conditions: negative, zero, positive, non-negative, non-zero, non-positive

Conditional Expression

- Branch if value of `R` meets one of six conditions: negative, zero, positive, non-negative, non-zero, non-positive
- `if X < Y goto Z`

```
Mov X, R0
Sub Y, R0
Jmp negative Z
```

Conditional Expression

- Branch if value of `R` meets one of six conditions: negative, zero, positive, non-negative, non-zero, non-positive
- `if X < Y goto Z`
  ```
  Mov X, R0
  Sub Y, R0
  Jmp negative Z
  ```
- Condition codes: indicate whether last quantity computed or loaded into a location is negative, zero, or positive

Conditional Expression

- Branch if value of `R` meets one of six conditions: negative, zero, positive, non-negative, non-zero, non-positive
- `if X < Y goto Z`
  ```
  Mov X, R0
  Sub Y, R0
  Jmp negative Z
  ```
- Condition codes: indicate whether last quantity computed or loaded into a location is negative, zero, or positive
- Compare instruction: sets the codes without actually computing the value

Conditional Expression

- Branch if value of `R` meets one of six conditions: negative, zero, positive, non-negative, non-zero, non-positive
- `if X < Y goto Z`
  ```
  Mov X, R0
  Sub Y, R0
  Jmp negative Z
  ```
- Condition codes: indicate whether last quantity computed or loaded into a location is negative, zero, or positive
- Compare instruction: sets the codes without actually computing the value
- `Cmp X, Y` sets condition codes to positive if `X > Y`
  ```
  Cmp X, Y
  CJL Z
  ```