Chapter 11 Target Code Generation

Zhen Gao
gaozhen@tongji.edu.cn

Outline

- Basic Issues
- Target Machine Model
- A Simple Code Generator

Input & Output

- Input to Code Generator
  - Intermediate code and information from the symbol table
  - Use symbol table information
    - determine runtime addresses of data objects
    - Type checking
- Output of Code Generator
  - Three forms:
    - Executable machine code
    - Relocatable machine modules
    - Assembly code

Basic Issues

- Generate target code as **short** as possible, avoiding redundant instructions
- Make full use of registers to reduce memory access in target code - **fast**
- Exploit the features of the instruction set for **efficiency**

Example: a:=a+1

```
INC a
```

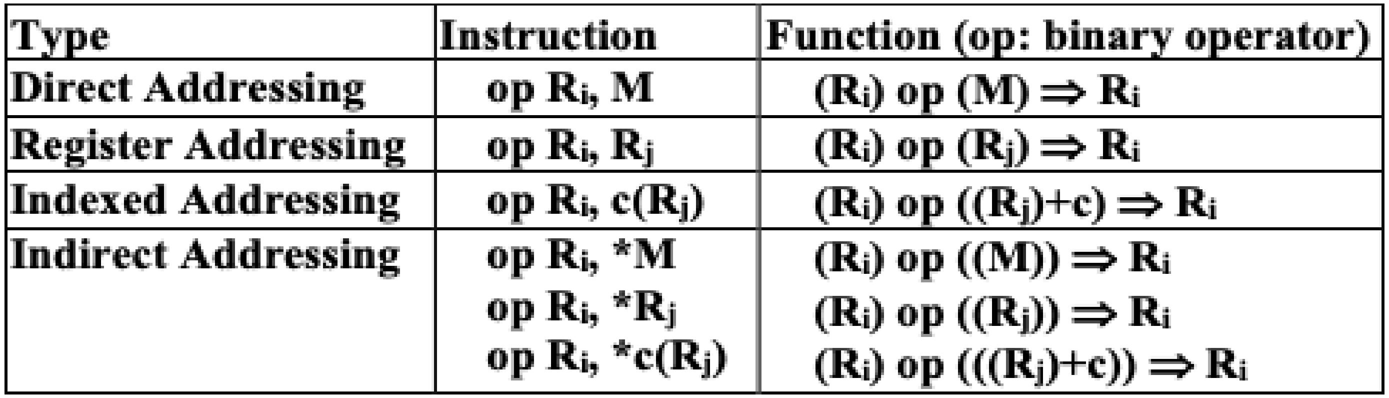
LD R_0, a
ADD R_0, #1
ST R_0, a

Outline

- Basic Issues
- Target Machine Model
- A Simple Code Generator

Target Machine Model

- An abstract computer model
  - Has multiple general-purpose registers, usable as **accumulators** or **index registers**
  - Operations must be performed in a **register**
  - Contains four types of **instruction formats**



Type	Instruction	Function (op: binary operator)
Direct Addressing	op R_i, M	(R_i) op $(M) \Rightarrow R_i$
Register Addressing	op R_i, R_j	(R_i) op $(R_j) \Rightarrow R_i$
Indexed Addressing	op R_i, $c(R_j)$	(R_i) op $((R_j)+c) \Rightarrow R_i$
Indirect Addressing	op R_i, *M op R_i, *R_j op R_i, *$c(R_j)$	(R_i) op $((M)) \Rightarrow R_i$ (R_i) op $((R_j)) \Rightarrow R_i$ (R_i) op $(((R_j)+c)) \Rightarrow R_i$

Instruction	Function
LD R_i, B	Load the contents of memory cell B into register Ri，（B）$\Rightarrow R_i$
ST R_i, B	Store the contents of register Ri into memory cell B，（Ri）$\Rightarrow$ B
J X	Unconditional jump to memory cell X
CMP A, B	Compare the values of cell A and cell B, and set the machine's internal condition register CT accordingly. CT occupies two bits. Set CT to 0, 1, or 2 depending on whether $A < B$, $A = B$, or $A > B$.
J< X	If CT=0 Transfer to cell X
J≤ X	If CT=0 or CT=1 Transfer to cell X
J= X	If CT=1 Transfer to cell X
J≠ X	If CT≠1 Transfer to cell X
J> X	If CT=2 Transfer to cell X
J≥ X	If CT=2 or CT=1 Transfer to cell X

Outline

- Basic Issues
- Target Machine Model
- A Simple Code Generator

A simple code generator

Not considering code execution efficiency, target code generation is straightforward. For example:

A:=(B+C)*D+E

Translated into quadruples:

T_1:=B+C

T_2:=T_1*D

T_3:=T_2+E

A:=T_3

Assuming only **one** register is available

Quadruples	Target Code		
T_1:=B+C	LD	R_0,	B
	ADD	R_0 ,	C
	ST	R_0 ,	T_1
T_2:=T_1*D	LD	R_0 ,	T_1
	MUL	R_0,	D
	ST	R_0 ,	T_2
T_3:=T_2+E	LD	R_0 ,	T_2
	ADD	R_0 ,	E
	ST	R_0 ,	T_3
A:=T_3	LD	R_0,	T_3
	ST	R_0 ,	A

- T1, T2, T3 are no longer referenced after the basic block
- Consider efficiency and full use of the register

LD	R_0,	B
ADD	R_0,	C
MUL	R_0,	D
ADD	R_0,	E
ST	R_0,	A

Register descriptor and address descriptor

- Register descriptor array RVALUE
  - Dynamically records the usage information of each register
  - **RVALUE[R]={A,B}**
- Variable address descriptor array AVALUE
  - Dynamically records the current storage locations of each variable
  - **AVALUE[A]={R_1, R_2, A}**

NEXT-USE & ACTIVE INFORMATION

Next-use Information

- Next-use information: If in a basic block, quadruple i defines A and quadruple j uses A with no other definition of A between them, then j is the **next-use** of A for i.
  - Example:

```
i:  A := B op C
…
j:  D := A op E
```

- Symbol table: Each variable entry contains fields for next-use and active information.

Next-use & Active Information

- (x, x) represents a variable’s next-use and active information
  - i = next-use, y = active, ^ = neither.
- In the symbol table, (x, x) → (x, x) means the latter pair replaces the former.
- Unless specified, all variables are considered inactive after the basic block exit.

Example

(1)

```
100: A:=B+C
101: D:=B-C
```

(2)

```
100: A:=B+C
101: B:=1
```

(3)

```
100: A:=B+C
101: B:=B+1
```

案例	A	B	C
(1)	(^, y)	(101, y)	(101, y)
(2)	(^, y)	(^, ^)	(^, ^)
(3)	(^, y)	(101, y)	(^, ^)

- **Solution:**

Scan the basic block **backward**.

- **Algorithm for computing next-use and active information:**
  - Initially, set each variable’s **next-use field** in the symbol table to “^” and set the **active field** according to whether the variable is active after the basic block exit.

- Process each quadruple in the basic block **backward** from exit to entry.
- For quadruple i: A := B op C, perform:
    - Attach A’s **next-use** and **active** information from the symbol table to quadruple i.
    - Set A’s next-use and active fields in the symbol table to “^,^”
    - Attach B and C’s next-use and active information from the symbol table to quadruple i.
    - Set B and C’s next-use fields to i and active fields to “active.”

$$\begin{tikzcd} \underline{(i)}: \quad A:=B \text{ op } C \\ \underline{(i+1)}: \quad D:=A \text{ op } E \arrow[u, bend right] \end{tikzcd}$$

Basic Block

1. **T:=A-B**
2. **U:=A-C**
3. **V:=T+U**
4. **W:=V+U**

- Let W be the set of variables active after the basic block exit
- Construct the next-use linked list and active variable linked list as follows

- Next-use / active information table attached to quadruples:

NO	Quadruple	Left	Left Operand	Right Operand
(4)	W:=V+U	(^,y)	(^,^)	(^,^)
(3)	V:=T+U	(4,y)	(^,^)	(4,y)
(2)	U:=A-C	(3,y)	(^,^)	(^,^)
(1)	T:=A-B	(3,y)	(2,y)	(^,^)

Variable	Initial state→Information chains (next-use/active)
T	(^,^) → (3,y) → (^,^)
A	(^,^) → (2,y) → (1,y)
B	(^,^) → (1,y)
C	(^,^) → (2,y)
U	(^,^) → (4,y) → (3,y) → (^,^)
V	(^,^) → (4,y) → (^,^)
W	(^,y) → (^,^)

T:=A-B
U:=A-C
V:=T+U
W:=V+U

□Attach A’ s **next-use** and **active** information
□Set A’ s next-use and active fields to “^,^”
□Attach B and C’ s next-use and active
□Set B and C’ s next-use and active fields to “i,active”

Quiz-Canvas

- **ch11 Target Code Generation: Next-use Information**

REGISTER ALLOCATION ALGORITHM

Register Allocation Principles

- **Keep if possible:** When generating code for a variable, keep it in a register as much as possible.
- **Use if possible:** Later code should use the variable’ s value in the register rather than accessing memory.
- **Free promptly:** At basic block exit, store any current register values back to memory.

- **Use a register exclusively holding B if possible.**
- **Use a free register if possible.**
- **Preempt a non-free register.**

GETREG(i: A := B op C)

Goal: Return a register to hold the result of A.

- **Rule 1:** If a register Ri currently holds **only B** in RVALUE[Ri] **and**:
  - B and A are the same variable (B will immediately get a new value), **or**
  - B’s current value will not be used (next-use information)

  then select Ri as the required register R and return it.
- **Notes:**
  - Avoid generating unnecessary Store instructions for B and possible future Load instructions.

- Use a register exclusively holding B if possible.
- Use a free register if possible.
- Preempt a non-free register.

Rule 2: If there is an unallocated register, select one R_i from them as the required register R (return).

```
If RVALUE(Ri)={ }
Then return(Ri)
```

- **Use a register exclusively holding B if possible.**
- **Use a free register if possible.**
- **Preempt a non-free register.**

Rule 3: Preempt Ri – Selection Criteria

(1) The variable in Ri does not need to be stored to memory and can be preempted directly.

- already in memory
- or will no longer be used.

(2) The variable in Ri will only be referenced in the distant future.

- Store any needed variable to memory before preempting.

What to do with the variable in a preempted register R_i

Think about it…

- **Case 1:** Statement A := B op C
  - RVALUE(Ri) = {B, C, D} → Do B, C, D need to be stored to memory?
  - **Answer: Yes**
- **Case 2:** Statement A := B op A
  - RVALUE(Ri) = {A} → Does A need to be stored to memory?
  - **Answer: Yes**. Because the operation uses A′ s current value, and Ri will be overwritten by B, so A must be stored first

Think about it…

- Case 3: A := B op A
  - RVALUE(Ri) = {A, B}
  - **Answer** : **No**, A is already in Ri and will soon be overwritten with a new value.
  - B needs to be stored, otherwise, if Ri is preempted, B′ s value would be lost.
- Case 4: A := B op C
  - RVALUE(Ri) = {A}
  - **Answer**: **No**, the old value of A will not be used again and will be replaced immediately by the new result.

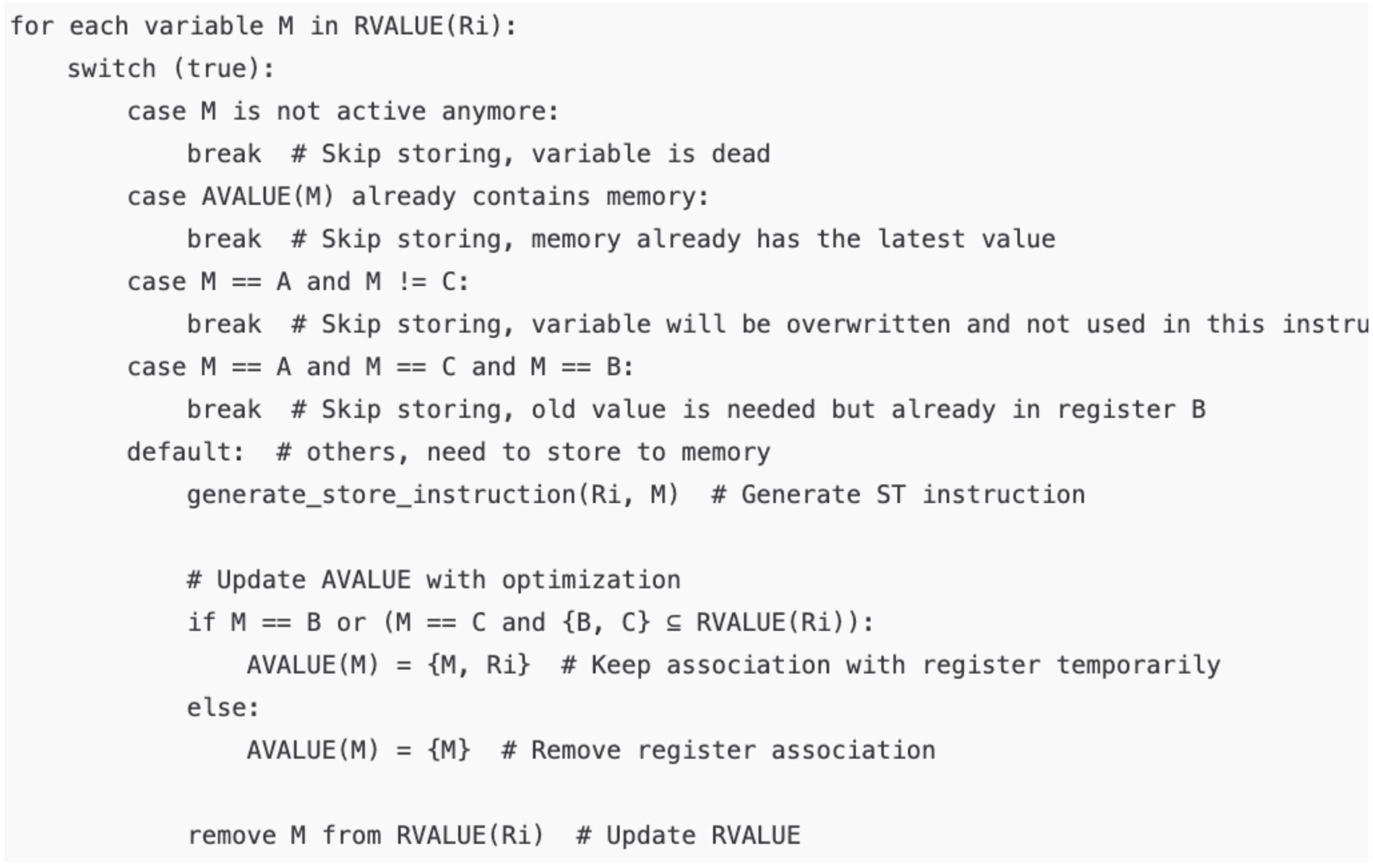
Think about it…

- What is principle for choosing registers?
  - minimize overhead — try to do as few **loads** and **stores** as possible.
- When to update RVALUE and AVALUE?
  - Update RVALUE when the value in a register changes.
  - Update AVALUE when the memory location (variable) is updated from a register.
- When does a RVALUE change?
  - When a new value is **loaded** from memory or somewhere else.
  - When a **computation** updates its value.
- When does an AVALUE change?
  - When a value is **stored** from register to memory.
  - When a **computation** occurs..

Situations where **store to memory is not needed**

- **Variable is dead** – it will not be used later.
- Memory already has the **latest value** – AVALUE(M) = {M}.
- Variable will be **overwritten** by this instruction (M = A):
    - Not used in this instruction (M ≠ C).
    - Used in this instruction (M = C) but its value is already in a register (M = B).

Situations where **store to memory is not needed**



```
for each variable M in RVALUE(Ri):
    switch (true):
        case M is not active anymore:
            break  # Skip storing, variable is dead
        case AVALUE(M) already contains memory:
            break  # Skip storing, memory already has the latest value
        case M == A and M != C:
            break  # Skip storing, variable will be overwritten and not used in this instru
        case M == A and M == C and M == B:
            break  # Skip storing, old value is needed but already in register B
        default:  # others, need to store to memory
            generate_store_instruction(Ri, M)  # Generate ST instruction

            # Update AVALUE with optimization
            if M == B or (M == C and {B, C} ⊆ RVALUE(Ri)):
                AVALUE(M) = {M, Ri}  # Keep association with register temporarily
            else:
                AVALUE(M) = {M}  # Remove register association

            remove M from RVALUE(Ri)  # Update RVALUE
```

CODE GENERATION ALGORITHM

Code generation algorithm

1. **Quadruple: A := B op C**
2. **Quadruple: A := B**
3. **End of basic block**

Code generation algorithm

Case 1: For each quadruple i: **A := B op C**, perform the following 5 steps:

- **Step 1: Allocate a register**
  - Call the function **GETREG**(i: A := B op C) with the quadruple as parameter.
  - It returns a register Ri to hold operand B and the result A.
- **Step 2: Locate the current values of B and C**
  - Use **AVALUE[B] and AVALUE[C]** to determine where the current values of B and C are stored (B' and C').
  - If the current value is already in a **register**, use that register as B' or C'.

Code generation algorithm

- **Step 3**: Generate target code
  - If B' ≠ R, generate:
    - LD R, B'   ; Load B' into R
    - op R, C'   ; Perform operation with C′
  - Otherwise, generate:
    - op R, C'   ; Perform operation directly in R
  - If B' or C' equals R, remove R from AVALUE[B] or AVALUE[C].
- **Step 4**: Update address and register descriptors
  - Set AVALUE[A] = {R} and RVALUE[R] = {A}.
- **Step 5**: Free registers of unused operands
  - If the current value of B or C is no longer used in the basic block and not live after the block, and its current value is in some register R_k, then:
    - Remove B or C from RVALUE[R_k]
    - Remove R_k from AVALUE[B] or AVALUE[C]
  - This frees the register so it is no longer occupied by B or C.

Generate target code

Case 2: Handling the quadruple **A := B**

- Important point: if the current value of B is already in a register Ri, there is no need to generate target code.
- Simply:
  - Add A to RVALUE(Ri) (i.e., assign Ri to both B and A).
  - Update AVALUE(A) to Ri.

Prerequisite	A:=B
RVALUE(Ri)={B}	**RVALUE(Ri)={B,A}**

AVALUE(B)={Ri}	AVALUE(B)={Ri}
	AVALUE(A)={Ri}

Generate target code

- what if B is not in any current register?
  - Generate: **LD R_i, B**
  - Update the register descriptor RVALUE(R_i) to {B, A} (replace)
  - Update the address descriptor AVALUE(B) to {xxx, Ri} (append)

Generate target code

Case 3: Code at the end of a basic block

At the end of a basic block, flush live variables to memory

- Register allocation is limited to the scope of a basic block, so once all quadruples in the block are processed, for each variable M whose current value is in a register:
  - If M is **live** after the block, **store** its value from the register into its memory location.

    ST R_i, M

  - Update the variable address descriptor: **AVALUE**(M) = {xxx, M} (append).

Basic Block

1. T:=A-B
2. U:=A-C
3. V:=T+U
4. W:=V+U

Assume **W** is the set of active variables after the basic block exit, and only **R0** and **R1** are available registers. Generate the target code along with the corresponding **RVALUE** and **AVALUE**.

1. Use B’ s exclusive register if possible.
2. Use free registers if possible.
3. Preempt non-free registers.

Intermediate code	Target code	RVALUE	AVALUE
T:=A−B	Get R_0 LD R_0, A SUB R_0, B	R_0: T	T: R_0
U:=A−C	Get R_1 LD R_1, A SUB R_1, C	R_0: T R_1: U	T: R_0(not Release U: R_1
V:=T+U	Get R_0 ADD R_0, R_1	R_0: V R_1: U	V: R_0(not Release U: R_1 T: (Release R1)
W:=V+U	Get R_0 ADD R_0, R_1 ST R_0, W	R_0: W R_1: (Release)	W: R_0, W U: (Release) V: (Release R0)

Quiz-Canvas

- **ch11 Target Code Generation: Register**

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

धन्यवाद
Hindi

감사합니다
Korean

תודה רבה
Hebrew

Tack så mycket
Swedish

Obrigado
Brazilian Portuguese

谢谢
Chinese

Dankon
Esperanto

Thank You !

ありがとうございます
Japanese

Trugarez
Breton

Danke
German

Tak
Danish

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic

寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器

1 尽可能用B独占的寄存器
2 尽可能用空闲寄存器
3 抢占用非空闲寄存器

1 {如果B的现行值在某个寄存器R_i中，RVALUE[R_i]中只包含B}，此外(And)，{或者B与A是同一个标识符，或者B的现行值在执行四元式A:=B op C之后不会再引用(此处要求B在后面是无用的变量或B是A，目的都是避免生成Store指令以及后续还可能的Load指令。只要这个B在后续还要用到，就不选择Ri寄存器。)}，则选取R_i为所需要的寄存器R(返回)；

规则一:
If (B独占Ri & B不需要刷到内存) Then return(Ri)

其中，B不会再被用到的情况:
(1) B就是A, 马上会有新的值，不会再被用到
(2) B的待用信息显示其不会再被用到

- 寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器

1 尽可能用B独占的寄存器
2 尽可能用空闲寄存器
3 抢占用非空闲寄存器

2 如果有尚未分配的寄存器，则从中选取一个R_i为所需要的寄存器R (返回) ；

规则二:

```
If RVALUE(Ri)={ }
Then return(Ri)
```

■ 寄存器分配：GETREG(i: A:=B op C) 返回一个用来存放A的值的寄存器

1 尽可能用B独占的寄存器
2 尽可能用空闲寄存器
3 抢占用非空闲寄存器

3 从已分配的寄存器中选取一个R_i为所需要的寄存器R。最好使得R_i满足以下条件：

占用R_i的变量的值也同时存放在该变量的贮存单元中，或者在基本块中要在最远的将来才会引用到或不会引用到。

规则三:
抢占Ri，挑选标准:
（1）Ri的变量不需要被刷到内存，可以直接抢
(可能有两种情况：已经在内存了，或者不会被用到了)
（2）Ri的变量要在最远的将来才会引用到
(把需要的变量刷到内存)

为被抢的R_i中的变量生成存数指令

假设1 语句 A:=B op C
RVALUE(Ri)={B,C,D}，B,C,D是否需要刷到内存
答案: 需要。

假设2 语句 A:=B op A
RVALUE(Ri)={A}，A是否需要刷到内存
答案: 需要。因为这个运算需要用到A值，且Ri会被B覆盖，A需要事先存储下来。

假设3 语句 A:=B op A
RVALUE(Ri)={A,B}，A是否需要刷到内存
答案: 不需要。虽然要用A，但是此时Ri不会被覆盖，A仍然在Ri中。

假设4 语句 A:=B op C
RVALUE(Ri)={A}，A是否需要刷到内存
答案: 不需要。因为A马上会有新值，且在此之前，A值不会再用到了。

对RVALUE(R_i)中每一变量M，如果M不是A(假设1)，或者如果M是A又是C，但不是B并且B也不在RVALUE[R_i]中(假设2)，则

(1) 生成目标代码 ST R_i，M（刷到内存）

例外：如果AVALUE(M)={M}，则不需要Store（不用刷内存）

(2) 更改变量地址描述数组 AVALUE(M)={M}（脱离Ri）

例外：如果M是B，AVALUE(M)={M，Ri}（如假设2，暂时不用脱离Ri）

例外：如果M是C & RVALUE(R_i)={B,C}，则AVALUE(M)={M，Ri}（如假设4，暂时不用脱离Ri）

(3) 删除RVALUE(R_i)中的M（更新寄存器描述数组）**返回**