

Compiler Design - 19z602

Assignment Presentation

A Simple Code Generator Algorithm

TEAM 3

Dheekshitha R - 22z216

Keerthi O - 22z243

Pramodini P - 22z244

Akash S - 22z255

Sanjitha R - 22z259

Table of contents

01

**Introduction and
Issues**

02

**The Target Language
Program and
Instruction Costs**

03

**Basic Blocks and
Flow Graphs**

04

**Simple Code Generator
Algorithm**

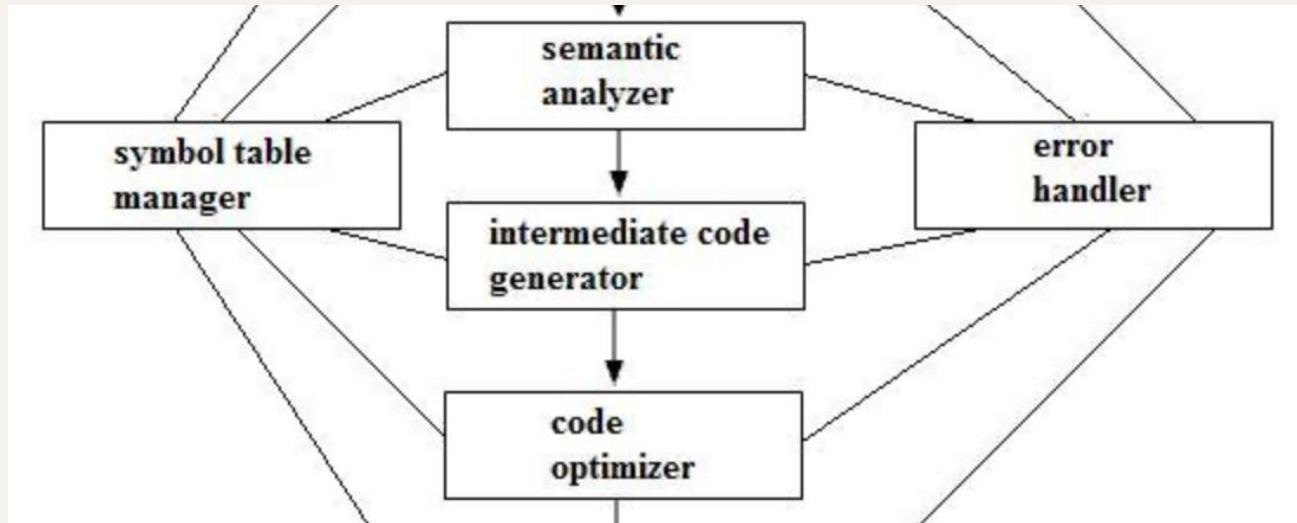
05

**Example Using Simple
Code Generator Algorithm**

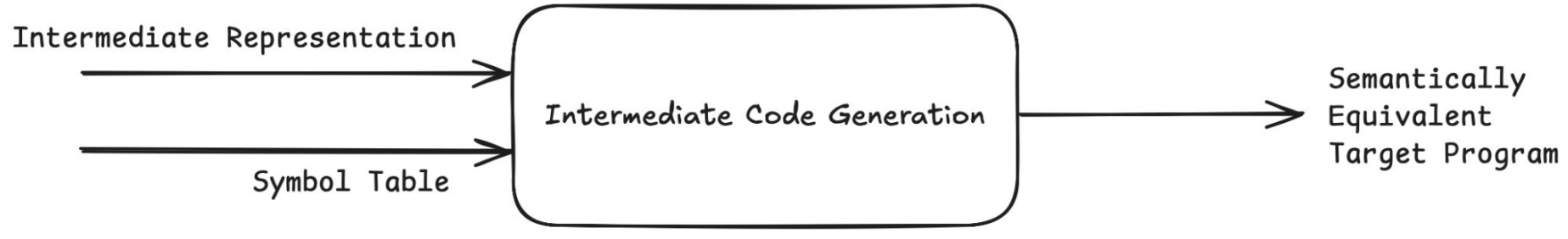
01

Introduction and Issues

Akash - 22z255



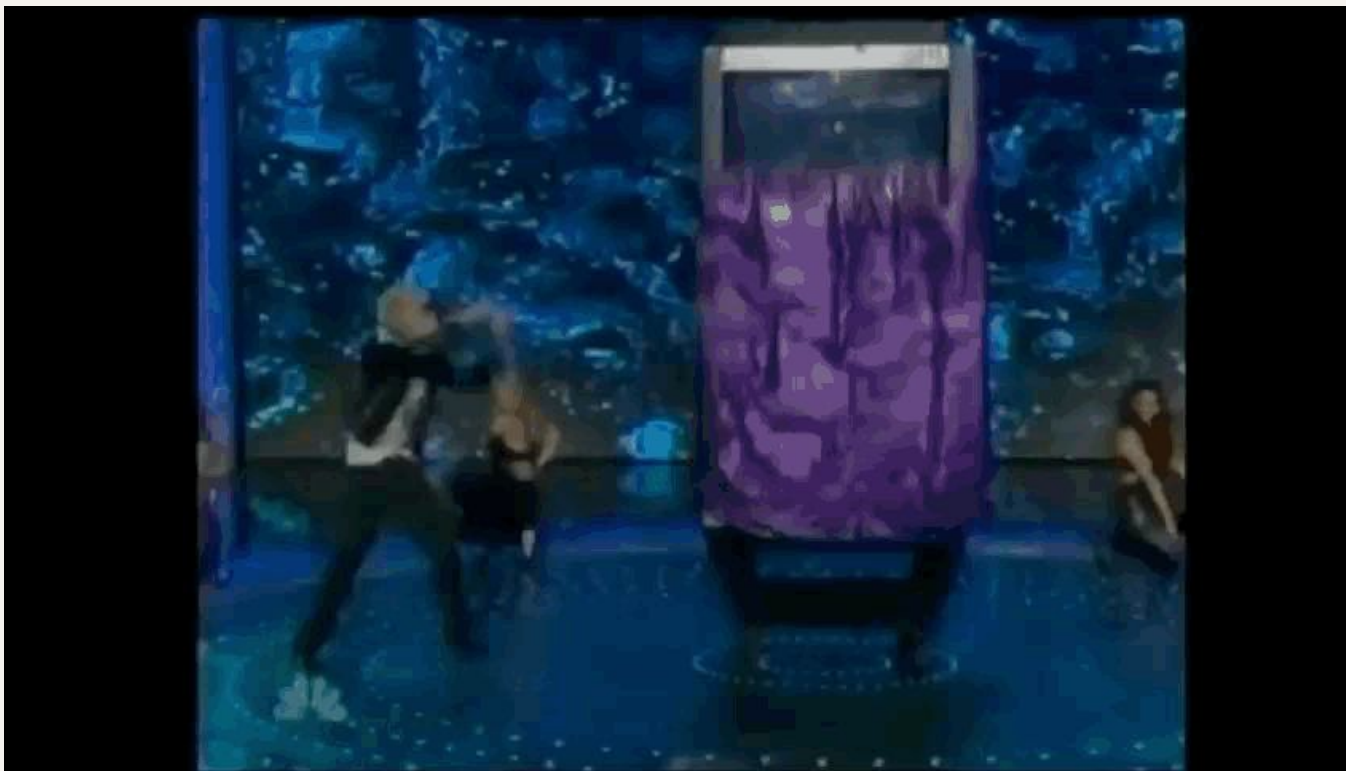
Right after semantic analyzer and before code optimizer



Right after semantic analyzer and before code optimizer



Input: 3 Address Code, Low Level Intermediate Representation



Output: Low-Level Register Based Intermediate Representation

Mathematically,

the problem of generating an optimal target program for a given source program is **undecidable**; many of the subproblems encountered in code generation such as register allocation are **computationally intractable**.

Heuristics

have advanced over the years to be capable of generating **much efficient** code compared to naive compilers that do not use them

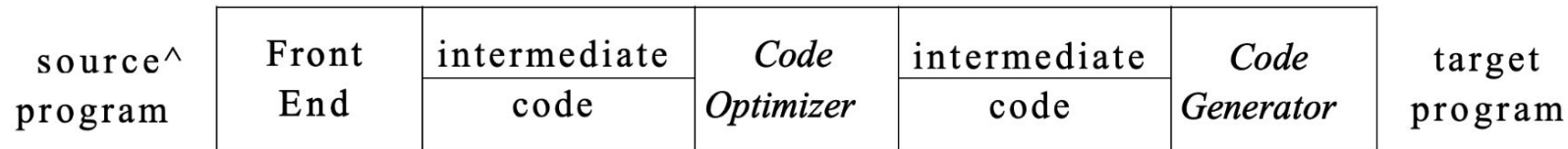


Figure 8.1: Position of code generator

5

Major issues with Code Generation

1. Input to Code Generator

The Intermediate Representation (IR) has various forms, including three-address representations, virtual machine representations, linear representations, and graphical representations. The front end has translated the source program into a low-level IR, detected errors, and performed type checking.

2. Target Program Output

The “Target Program” problem in code generation involves ensuring generated code is semantically equivalent to the source code and high-quality. The instruction-set architecture of the target machine, including RISC, CISC, and stack-based, impacts the difficulty of creating a good code generator and the quality of the machine code.

3. Choosing the right Instruction

Different Instructions might be better than other Instructions in certain context

3. Choosing the right Instruction

Consider the following equation:

$$x = y + z$$

which can be translated to the following code sequence :

```
LD R0, y
ADD R0, R0, z
ST x, R0
```

3. Choosing the right Instruction

Then for the following equations:

$$a = b + c$$

$$d = a + e$$

generated code sequence will be:

```
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```

3. Choosing the right Instruction

```
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```

3. Choosing the right Instruction

LD R0, b

ADD R0, R0, c

ST a, R0

LD R0, a

ADD R0, R0, e

ST d, R0

3. Choosing the right Instruction

Similarly for an increment operation, i.e.,

$$a = a + 1$$

which one should the compiler choose?

```
LD R0, a
ADD R0, R0, #1
ST a, R0
```

```
INC A
```

3. Choosing the right Instruction

The only solution for this problem is to know the exact cost of every single platform of every single platform and plan accordingly.

Well that's impossible and tedious

4. Register Allocation

Register allocation and assignment are key problems in code generation, determining which variables reside in registers and which specific registers they occupy. Finding an optimal assignment is **difficult**, even for single-register machines, and is further complicated by hardware and operating system requirements.

5. Evaluation Order

The "Evaluation Order" problem in code generation involves determining the most efficient sequence for computations, impacting register usage and complexity.

02

The Target Language Program and Instruction Costs

Dheekshitha R - 22z216

Target Language

Familiarity with the **target machine** and its **instruction set** is a prerequisite for designing a good code generator.

Unfortunately, in a general discussion of code generation it is **not possible to describe any target machine** in sufficient detail to generate good code for a complete language on that machine.

We shall use as a **target language assembly code for a simple computer** that is representative of many **register machines**.

However, this code generation techniques can be used on many **other classes of machines** as well.

Target Machine Overview

Key Points:

- The target machine is a **three-address machine**.
- It supports **load, store, computation, jump, and conditional jump** operations.
- The machine is **byte-addressable** with **`n` general-purpose registers: `R0, R1, ..., Rn-1`**.
- The instruction set is simplified to focus on core concepts, assuming all operands are integers

Instruction Format

- An **operator** (e.g., **`LD`**, **`ST`**, **`ADD`**, **`SUB`**).
- A **target** (destination location).
- A **list of source operands**
- A label may precede an instruction for branching purposes.
- **Example:** **`LD R1, x`** (load the value of **`x`** into register **`R1`**).

Load Operations

Load Instruction: ``LD dst, addr``

- Loads the value at memory location ``addr`` into ``dst``.
- **Example:** ``LD R1, x`` (load the value of ``x`` into register ``R1``).

Register-to-Register Copy:

- **Example:** ``LD R1, R2`` (copy the contents of ``R2`` into ``R1``).

Assignment Semantics: ``dst = addr``.

Store Operations

Store Instruction: ``ST x, r``

- Stores the value in register ``r`` into memory location ``x``.
- **Example:** ``ST x, R1`` (store the value of ``R1`` into ``x``).

Assignment Semantics: ``x = r``.

Load Computation Operations

Computation Instruction: `OP dst, src1, src2`

- `OP` can be `ADD`, `SUB`, etc.
- **Example:** `SUB R1, R2, R3` (compute `R1 = R2 - R3`).

Unary Operations:

- **Example:** `NEG R1, R2` (negate the value in `R2` and store in `R1`).

Assignment Semantics: `dst = src1 OP src2`.

Jump Operations

Unconditional Jump: `BR L`

- Branches to the instruction with label `L`.
- **Example:** `BR LOOP` (jump to the label `LOOP`).

Conditional Jump: `Bcond r, L`

- Branches to label `L` if the condition is met.
- **Example:** `BLTZ R1, L` (jump to `L` if `R1 < 0`).

Addressing Modes in the Target Machine

- **Direct Addressing:**

- **Example:** LD R1, x (load the value of x into R1).
- Refers to the memory location reserved for x (the l-value of x).

- **Indexed Addressing:**

- **Example:** LD R1, a(R2) (load the value at a + contents(R2) into R1).
- Useful for accessing arrays, where a is the base address and R2 holds the offset.

- **Indirect Addressing:**

- **Example:** LD R1, *100(R2) (load the value at the address stored in 100 + contents(R2)).
- Useful for following pointers.

- **Immediate Constant Addressing:**

- **Example:** LD R1, #100 (load the integer 100 into R1).
- **Example:** ADD R1, R1, #100 (add 100 to the value in R1).

Three-Address Statement to Machine Code

Three-Address Statement: $x = y - z$

Machine Code:

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

Optimization:

- If y or z is already in a register, skip the corresponding LD instruction.
- Avoid storing x if its value is only used in registers.

Array and Pointer Operations

Array Access Example

Three-Address Statement: $b = a[i]$

Machine Code:

```
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
```

Explanation: The second step computes the offset ($8 * i$), and the third step loads the array element.

Pointer Indirection Example

Three-Address Statement: $x = *p$

Machine Code:

```
LD  R1, p           // R1 = p
LD  R2, 0(R1)        // R2 = contents(0 + contents(R1))
ST  x, R2            // x = R2
```

Explanation: The second step dereferences the pointer p.

Conditional Jump and Optimization

- **Conditional Jump Example:**

- **Three-Address Statement:** if $x < y$ goto L

- **Machine Code:**

```
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
```

- **Explanation:** M is the label corresponding to the target of the jump (label L in the three-address code).

- **Optimization:**

- If x or y is already in a register, skip the corresponding LD instruction.
- Avoid storing intermediate results if they are only used for the jump.

- **Example:**

- If x is already in R1 and y is already in R2, the optimized code would be:

```
SUB   R1, R1, R2       // R1 = R1 - R2
BLTZ  R1, M           // if R1 < 0 jump to M
```

Program and Instruction Cost

- **Cost Measures:**
 - Compilation time.
 - Size of the target program.
 - Running time of the target program.
 - Power consumption of the target program.
- **Complexity:**
 - Finding an optimal target program for a given source program is undecidable in general.
 - Many subproblems involved are NP-hard.
 - Heuristic techniques are often used to produce good (but not necessarily optimal) target programs.
- **Instruction Cost:**
 - Each target-language instruction has an associated cost.
 - $\text{Cost} = 1 + \text{cost of addressing modes}$.
 - Register addressing: No additional cost.
 - Memory or constant addressing: Additional cost of 1 (due to extra words needed to store the operand)

Examples of Instruction Costs

- **Example 1:** LD R0, R1
 - Copies the contents of R1 into R0.
 - Cost = 1 (no additional memory words required).
- **Example 2:** LD R0, M
 - Loads the contents of memory location M into R0.
 - Cost = 2 (address of M is stored in the word following the instruction).
- **Example 3:** LD R1, *100(R2)
 - Loads the value at the address stored in 100 + contents(R2) into R1.
 - Cost = 3 (one for the displacement value, and one for accessing memory).
- **Program Cost:**
 - The cost of a target-language program is the sum of the costs of the individual instructions executed.
 - Goal of Code Generation: Minimize the sum of instruction costs on typical inputs.
- **Optimal Code:**
 - In some cases, optimal code can be generated for expressions on certain classes of register machines.

RISC
VS
CISC

No.	Characteristic	RISC	CISC
1	Instruction Complexity	● Simple instructions taking one cycle	● Complex instructions taking multiple cycles
2	Memory Reference	● Very few instructions refer memory	● Most of instructions may refer memory
3	Instruction Execution	● Instructions are executed by hardware	● Instructions are executed by microprogram
4	Instruction Format	● Fixed format instructions	● Variable format instructions
5	Number of Instructions	● Many instructions	● Few instructions
6	Addressing Modes	● Few addressing modes, most instructions have register to register addressing mode	● Many addressing modes
7	Complex Addressing	● Complexed addressing modes are synthesized in software	● Supports complex addressing modes
8	Register Sets	● Multiple register sets	● Single register set
9	Pipelining	● Highly pipelined	● Not pipelined or less pipelined
10	Complexity Location	● Complexity is in the compiler	● Complexity is in the microprogram
11	Conditional Jump	● Conditional jump can be based on a bit anywhere in memory	● Conditional jump is usually based on status register bit

RISC (Load Store Arch.)

```
LOAD R1, A      ; Load value of A into register R1
LOAD R2, B      ; Load value of B into register R2
ADD R3, R1, R2   ; Add R1 and R2, store result in R3
STORE R3, C      ; Store the result in memory location C
```

CISC(Memory to Memory)

```
ADD C, A, B ;
```

Example: Loop Execution (Summing an Array of 10 Elements)

RISC Example

assembly

```
LOAD R1, #0      ; Initialize sum to 0
LOAD R2, #10     ; Loop counter
LOAD R3, #ARRAY  ; Load array base address
```

LOOP:

```
    LOAD R4, 0(R3) ; Load element from array
    ADD R1, R1, R4  ; Add to sum
    ADD R3, R3, #4  ; Move to next element
    SUB R2, R2, #1  ; Decrement loop counter
    BNEZ R2, LOOP   ; If counter  $\neq$  0, repeat
```

```
STORE R1, SUM    ; Store final sum in memory
```

CISC Example

assembly

```
MOV CX, 10      ; Loop counter
MOV SI, ARRAY   ; Load array base address
MOV AX, 0       ; Initialize sum
```

LOOP:

```
    ADD AX, [SI]  ; Add array element to sum
    ADD SI, 4     ; Move to next element
    LOOP LOOP     ; Loop automatically decrements CX and jumps
```

```
MOV SUM, AX     ; Store final sum
```



03

Basic Blocks and Flow Graph

Pramodini P - 22z244

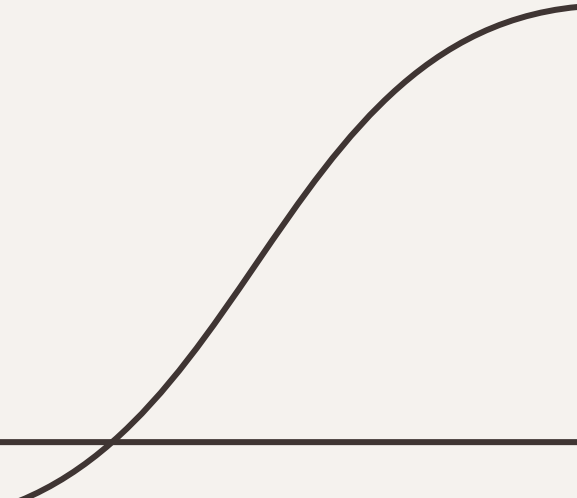


Introduction

A basic block is a sequence of consecutive three-address instructions with a single entry point and a single exit point.

A flow graph is a directed graph where nodes represent basic blocks and edges represent control flow between them.

Why is this important?

- Helps in **code generation** and **optimization**
 - Improves **register allocation**
 - Aids in **instruction selection**
- 

Example

```
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
```

Leaders:

Instruction 1 (First instruction)

Instruction 2 (Target of jump at 11)

Instruction 3 (Target of jump at 9)

Instruction 10 (Follows a jump)

Instruction 12 (Follows a jump)

Instruction 13 (Target of jump at 17)

Basic Block Properties



Single Entry

Control enters the block only at the first instruction.

This ensures a clear starting point for execution.



Single Exit

Execution proceeds sequentially, except for the last instruction.

This allows for a defined endpoint in the block.



Maximal Sequence

No jumps or labels are present within the block except at the end.

This characteristic maximizes the sequence of instructions.



Control Flow

Basic blocks represent a linear flow of control in execution.

They help in understanding the transitions between different parts of the code.



Code Optimization

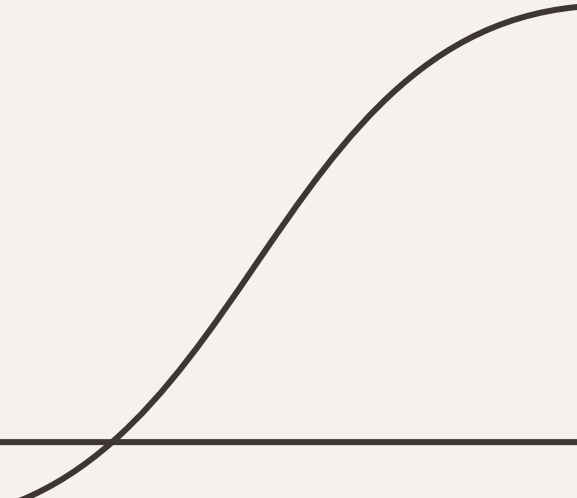
Basic blocks are essential for various optimization techniques in compilers.

They facilitate improvements in register allocation and instruction scheduling.

Algorithm for Identifying Basic Blocks

Input: A sequence of three-address instructions

Output: A list of basic blocks

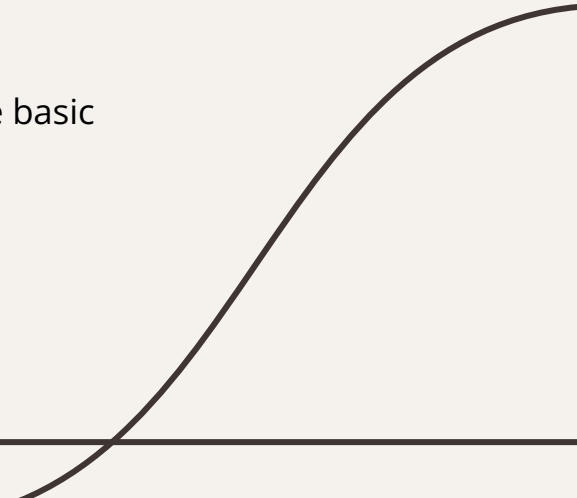


Algorithm for Identifying Basic Blocks

1) Identifying Leaders

- The first instruction in the program is a leader.
- Instructions that are targets of jumps or immediately follow jumps are also classified as leaders.

2) Forming Blocks

- Instructions are grouped under their respective leaders to create basic blocks.
 - This organization is essential for the compilation process.
- 

Example

```
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
```

Source code:

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i,j] = 0.0;
for i from 1 to 10 do
    a[i, i] = 1.0;
```

Forming Basic Blocks

Using the identified leaders, we form the following basic blocks:

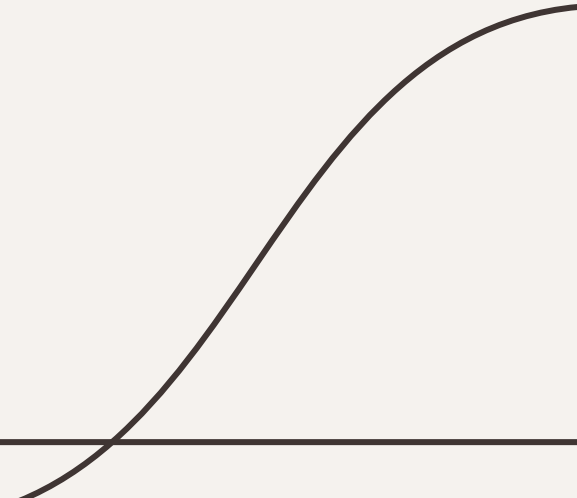
1. **Basic Block 1:** (1)
 2. **Basic Block 2:** (2)
 3. **Basic Block 3:** (3, 4, 5, 6, 7, 8, 9)
 4. **Basic Block 4:** (10, 11)
 5. **Basic Block 5:** (12)
 6. **Basic Block 6:** (13, 14, 15, 16, 17)
-

Interrupts and Basic Blocks

Normally, control flows from the beginning to the end of a basic block.

Interrupts and exceptions (like division by zero) may break this flow but are **ignored** when constructing flow graphs.

Why?

- Interrupts either return to the same instruction or halt the program.
 - Optimizations assume normal execution paths.
- 

Interrupts and Basic Blocks

Control Flow Impact

- Interrupts can disrupt the normal flow of execution in a program.
- Interrupts can redirect execution to an exception handler.
- This disruption is not considered when constructing flow graphs.
- Flow graphs assume a linear control flow without interruptions.

Execution Disruption

- Execution can be interrupted by events such as division by zero.
- These events can cause the program to stop executing.
- Interrupts may lead to a return to a previous state.
- Despite their impact, interrupts are ignored in flow graph construction.

Next-Use Information

Why is Next-Use Information Important?

- In code generation, efficient **register allocation** is crucial.
- If a variable's value will never be used again, its register can be reassigned to another variable.
- This reduces the number of registers needed and improves performance.

A variable **x** is **live** at a given statement **i** if:

- There is a later statement **j** in the same basic block where **x** is used as an operand.
- There is a control flow path from **i** to **j** **without any reassignments to x** in between

Next-Use Information Algorithm

Algorithm to Compute Liveness and Next-Use Information

1. **Start from the last statement** in the basic block and move **backward** to the first statement.
2. **At each statement** of the form $x = y + z$, do the following:
 - Attach the **current** next-use and liveness information for **x, y, and z** from the symbol table
 - Mark **x as “not live”** and set its **next use to “none”** in the symbol table (since its latest value is assigned here).
 - Mark **y and z as “live”** and set their **next-use** information to the current statement index (i).

Next-Use Information

Consider the following example:

1. $t = a - b$

2. $u = a - c$

3. $v = t + u$

4. $a = d$

5. $d = v + u$

Next-Use Information

- We assume all non-temporary variables (a, b, c, d, e, f, g) are live on exit.
- We process backward, updating the next-use and liveness information.

Before Processing (Initial State)

Variable	Live?	Next Use
a	Yes	?
b	Yes	?
c	Yes	?
d	Yes	?
t	No	None
u	No	None
v	No	None

Processing Statement 5: $d = v + u$

Variable	Live?	Next Use
a	Yes	?
b	Yes	?
c	Yes	?
d	No	None
t	No	None
u	Yes	5
v	Yes	5

Processing Statement 4: $a = d$

Variable	Live?	Next Use
a	No	None
b	Yes	?
c	Yes	?
d	Yes	4
t	No	None
u	Yes	5
v	Yes	5

Processing Statement 3: $v = t + u$

Variable	Live?	Next Use
a	No	None
b	Yes	?
c	Yes	?
d	Yes	4
t	Yes	3
u	Yes	3, 5
v	No	None

Processing Statement 2: $u = a - c$

Variable	Live?	Next Use
a	Yes	2
b	Yes	?
c	Yes	2
d	Yes	4
t	Yes	3
u	No	None
v	No	None

Processing Statement 1: $t = a - b$

Variable	Live?	Next Use
a	Yes	1, 2
b	Yes	1
c	Yes	2
d	Yes	4
t	No	None
u	No	None
v	No	None

Flow Graph

- **Flow Graphs** represent the flow of control between basic blocks in an intermediate-code program.
- **Nodes** are basic blocks, and **Edges** represent possible control flow between them.

There are two ways that such an edge could be justified:

- There's a jump (conditional or unconditional) from block B to C.
- C follows B in the program sequence, and B doesn't end in an unconditional jump.

Entry,Exit and Predecessors

Entry Node: Represents the start of the program, pointing to the first executable basic block.

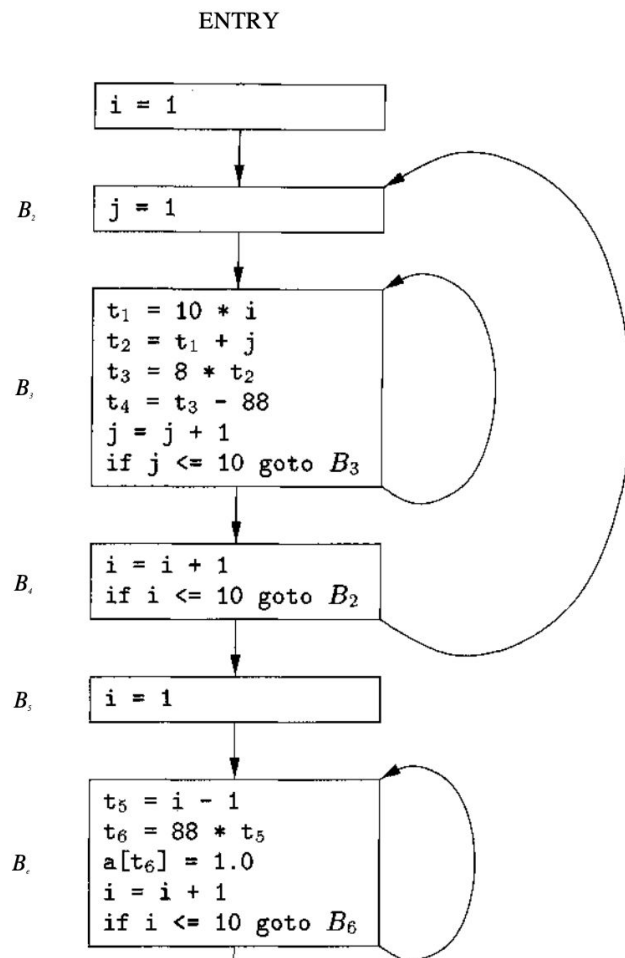
Exit Node: Represents the program's end, pointing from blocks that could be the final executed instruction.

Predecessors & Successors:

- **Predecessor:** A block that can transfer control to another block.
 - **Successor:** A block that can receive control from another block.
-

Example

```
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
```



Loops

A set of nodes **L** in a flow graph forms a loop if:

1. **Loop Entry:**

- There is a unique **entry node** in L.
- No other node in L has a predecessor outside L.
- Any path from the flow graph entry to L must pass through this entry node.

2. **Reachability:**

- Every node in L has a nonempty path **within L** leading to the loop entry.
-

Loops

The flow graph of given previous example has three loops:

1. B3 by itself.
2. B6 by itself.
3. {B2, B3, B4}.

- B2 is the only node in {B2, B3, B4} with a predecessor (B1) outside the loop.
- Each node in {B2, B3, B4} has a nonempty path leading back to B2.
- Example path: $B2 \rightarrow B3 \rightarrow B4 \rightarrow B2$.


Importance in Compiler Design

Code Motion: Move independent computations outside loops.

Dead Code Elimination: Remove unreachable or unnecessary computations.

Register Allocation: Assign registers efficiently across basic blocks.

Instruction Scheduling: Reorder instructions to improve performance.

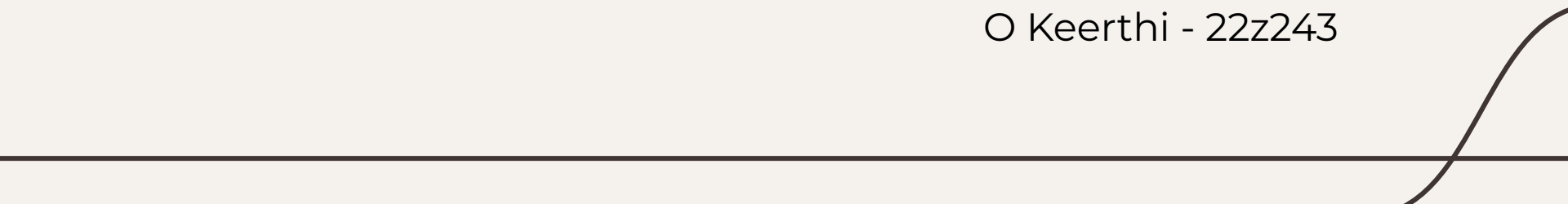




04

Simple Code Generator Algorithm

○ Keerthi - 22z243



Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.

In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so. We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored.

The desired data structure has the following definition:

Register and Address Descriptors

- For each available register, a **register descriptor** keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a block, we assume that initially all register descriptors are empty. As the code generation progresses, each register will hold the value of 0 or more names.
 - For each program variable, an **address descriptor** keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol table entry for that variable name
-

Instruction Generation

1. Call `getReg(OP x, y)` to obtain registers.
 2. Check register descriptor for `y`:
 3. If missing, load from memory (`LD Ry, y`).
 4. Repeat for `x`.
 5. Generate `OP Rx, Ry` (e.g., `ADD Rx, Ry`).
-



The Code-Generation Algorithm

An essential part of the algorithm is a function *getReg(I)*, which selects registers for each memory location associated with the three-address instruction *I*. Function *getReg* has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block. We shall discuss *getReg* after presenting the basic algorithm. While we do not know the total number of registers available for local data belonging to a basic block, we assume that there are enough registers so that, after freeing all available registers by storing their values in memory[^] there are enough registers to accomplish any three-address operation.

Instruction Generation

Machine Instructions for Operations

For a three-address instruction such as $x = y + z$, do the following:

1. Use *getReg*($x = y + z$) to select registers for x , y , and z . Call these R_x , R_y , and R_z .
2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction LD R_y, y' , where y' is one of the memory locations for y (according to the address descriptor for y).
3. Similarly, if z is not in R_z , issue an instruction LD R_z, z' , where z' is a location for z .
4. Issue the instruction ADD R_x, R_y, R_z .

Instruction Generation

Machine Instructions for Copy Statements

There is an important special case: a three-address copy statement of the form $x = y$. We assume that *getReg* will always choose the same register for both x and y . If y is not already in that register R_y , then generate the machine instruction LD R_y, y . If y was already in R_y , we do nothing. It is only necessary that we adjust the register description for R_y so that it includes x as one of the values found there.

Managing Register & Address Descriptors

1. Load (**LD** **R**, **x**)

- Assign **x** to register **R**.
- **Update descriptors:**
 - $\text{Desc}(\mathbf{R}) = \mathbf{x}$ (remove other values from **R**).
 - Add **R** to $\text{Desc}(\mathbf{x})$.
 - Remove **R** from $\text{Desc}(\mathbf{w})$ for all $\mathbf{w} \neq \mathbf{x}$.

Managing Register & Address Descriptors

1. Load (**LD** R , x)

1. For the instruction LD R_x

- (a) Change the register descriptor for register R so it holds only x .
- (b) Change the address descriptor for x by adding register R as an additional location.

Managing Register & Address Descriptors

2. Store (ST x , R)

- Save x from register R to memory.
- **Update Address Descriptor:** Add memory location of x to $\text{Desc}(x)$.

For the instruction ST x , R , change the address descriptor for x to include its own memory location.

Managing Register & Address Descriptors

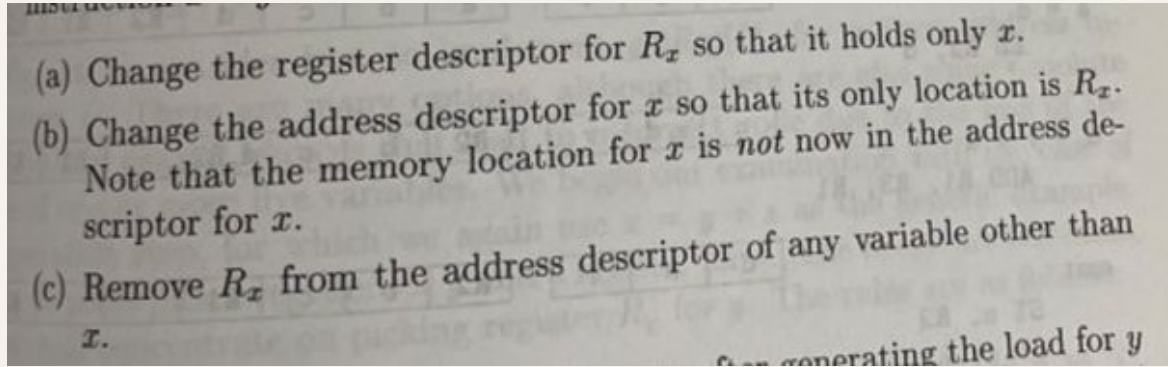
3. Operation (**OP Rx, Ry**)

- Performing $x = x \text{ OP } y$:
 - $\text{Desc}(Rx) = x$.
 - $\text{Desc}(y) = Ry$.
 - After execution, Rx holds $Rx \text{ OP } Ry$.

Managing Register & Address Descriptors

3. Operation (OP Rx, Ry)

- For an operation such as ADD Rx, Ry, Rz implementing a three-address instruction $x = y + z$



(a) Change the register descriptor for R_x so that it holds only x .

(b) Change the address descriptor for x so that its only location is R_x .
Note that the memory location for x is *not* now in the address descriptor for x .

(c) Remove R_x from the address descriptor of any variable other than x .

generating the load for y

Managing Register & Address Descriptors

4. Copy ($x = y$)

x.

4. When we process a copy statement $x = y$, after generating the load for y into register R_y , if needed, and after managing descriptors as for all load statements (per rule 1):

- (a) Add x to the register descriptor for R_y .
- (b) Change the address descriptor for x so that its only location is R_y .

Block consisting of the three-address

Special Case – Copy ($x = y$)

- If y is in a register, set $R_x = R_y$.
- If y was already in a register, **no code is generated**.
- Store y in memory if required at block exit.

Minimizing Register Usage

Reuse registers if:

1. **Temporary values** are no longer needed.
2. A variable **has no further use** in the program.
3. A variable is **redefined before next use**.

Handling Live Variables

- Variables **live at block exit** must be stored in memory.
- **Dead variables** at exit are ignored.

getReg Function

- so we shall concentrate on picking registers
1. If y is currently in a register, pick a register already containing y as R_y .
Do not issue a machine instruction to load this register, as none is needed.
 2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y .
 3. The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse. Let R be a candidate

getReg Function

- so we shall concentrate on picking registers
1. If y is currently in a register, pick a register already containing y as R_y .
Do not issue a machine instruction to load this register, as none is needed.
 2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y .
 3. The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse. Let R be a candidate

getReg Function

register, and suppose v is one of the variables that the register descriptor for R says is in R . We need to make sure that v 's value either is not really needed, or that there is somewhere else we can go to get the value of v . The possibilities are:

- (a) If the address descriptor for v says that v is somewhere besides R , then we are OK.
- (b) If v is x , the variable being computed by instruction I , and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.
- (c) Otherwise, if v is not used later (that is, after the instruction I , there are no further uses of v , and if v is live on exit from the block, then v is recomputed within the block), then we are OK.
- (d) If we are not OK by one of the first three cases, then we need to generate the store instruction $ST\ v, R$ to place a copy of v in its own memory location. This operation is called a *spill*.

getReg Function

Now, consider the selection of the register R_x . The issues and options are almost as for y , so we shall only mention the differences.

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x . This statement holds even if x is one of y and z , since our machine instructions allows two registers to be the same in one instruction.
2. If y is not used after instruction I , in the sense described for variable v in item (3c), and R_y holds only y after being loaded, if necessary, then R_y can also be used as R_x . A similar option holds regarding z and R_z .

The last matter to consider specially is the case when I is a copy instruction $x = y$. We pick the register R_y as above. Then, we always choose $R_x = R_y$.



05

Example Using Simple Code Generator Algorithm

Sanjitha R - 22z259



Example:

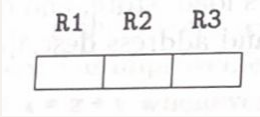
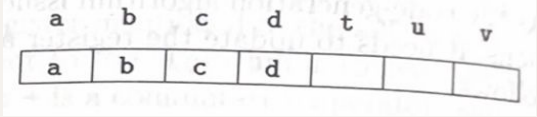
- 1) A basic block consisting of three-address statements

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

Assumptions:

- a) t, u and v are temporaries [local to the block]
- b) a, b, c and d are variables that are alive on exit from the block
- c) Initially, all the registers are empty

Example:

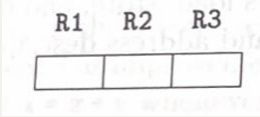
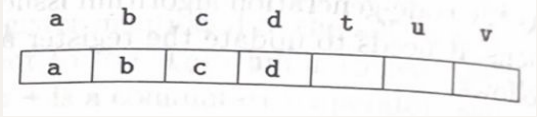
Three - address instruction	Generated machine-code instruction	Register Descriptor 	Address Descriptor 
t=a-b			

Managing Register & Address Descriptors

1. Load (**LD** R , x)

- For the instruction LD R , x
- (a) Change the register descriptor for register R so it holds only x .
 - (b) Change the address descriptor for x by adding register R as an additional location.

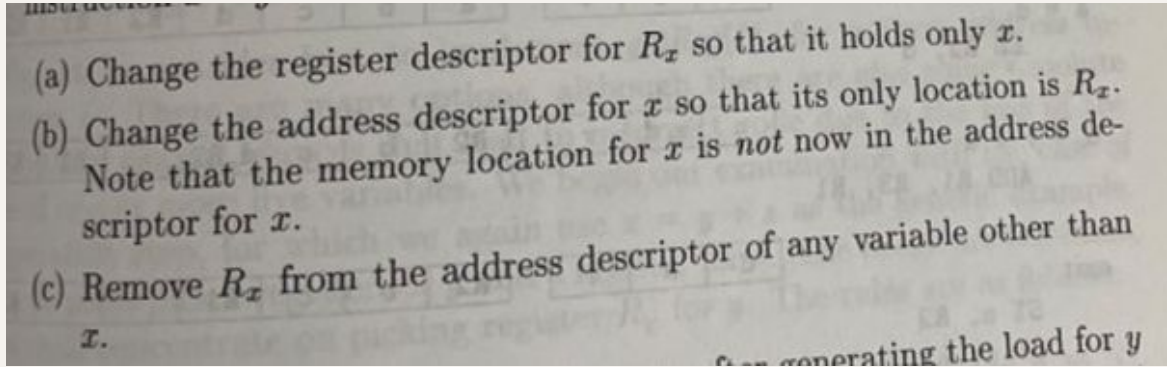
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
			
t=a-b	<pre>LD R1, a LD R2, b SUB R2, R1, R2</pre>		

Managing Register & Address Descriptors

3. Operation (OP Rx, Ry)

- For an operation such as ADD Rx, Ry, Rz implementing a three-address instruction $x = y + z$



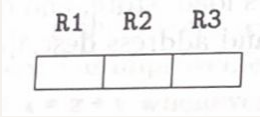
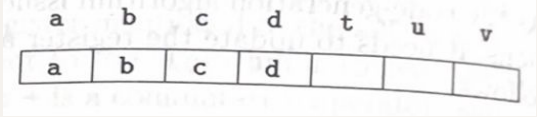
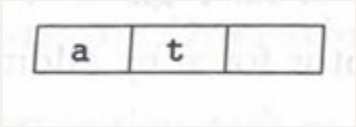
(a) Change the register descriptor for R_x so that it holds only x .

(b) Change the address descriptor for x so that its only location is R_x .
Note that the memory location for x is *not* now in the address descriptor for x .

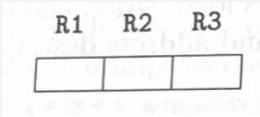
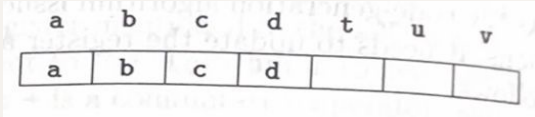
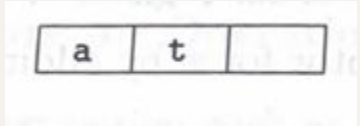
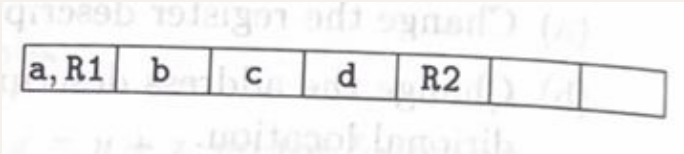
(c) Remove R_x from the address descriptor of any variable other than x .

...generating the load for y

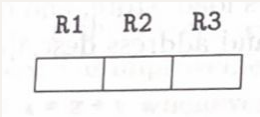
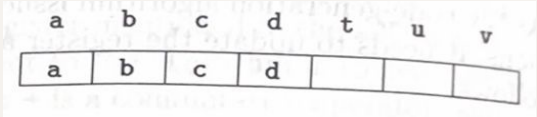

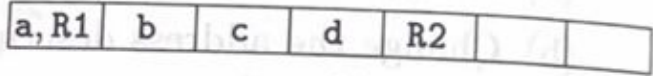
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	<pre>LD R1, a LD R2, b SUB R2, R1, R2</pre>		
			

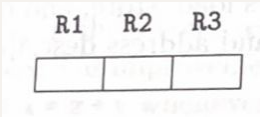
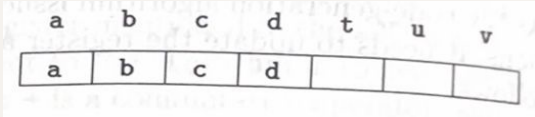
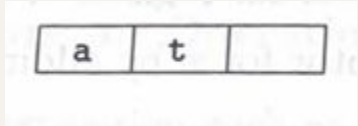
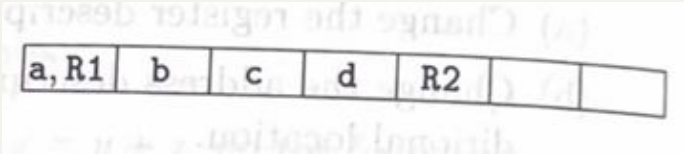
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	<pre>LD R1, a LD R2, b SUB R2, R1, R2</pre>		
			

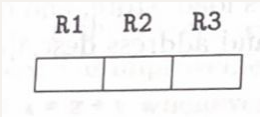
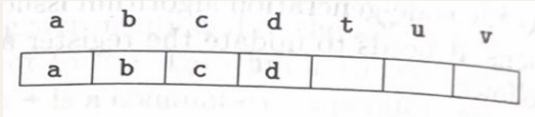

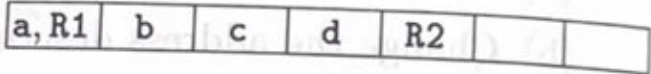
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	<pre>LD R1, a LD R2, b SUB R2, R1, R2</pre>		
u=a-c			

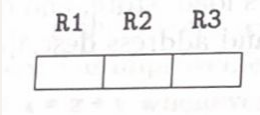
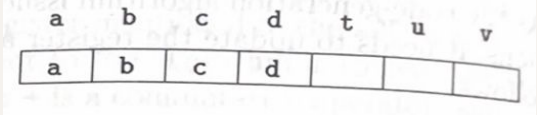

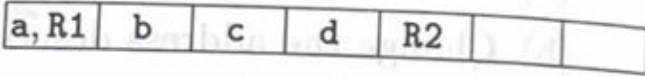
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		

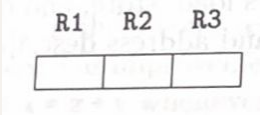
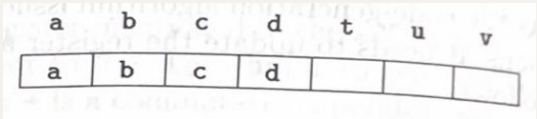

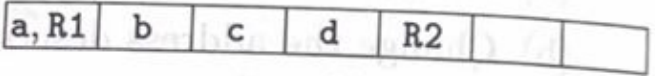
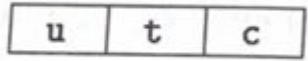
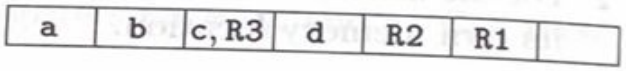
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		

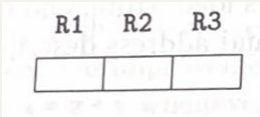
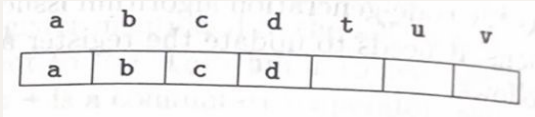

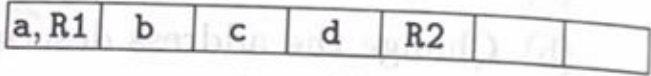
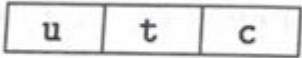
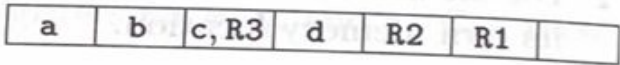
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		

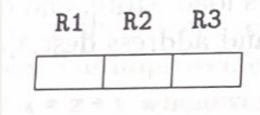
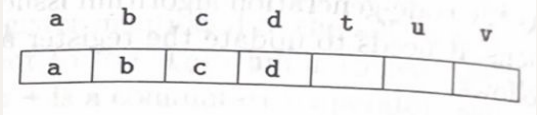

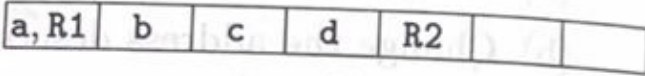
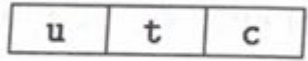
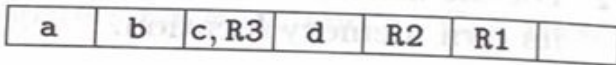
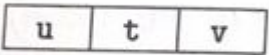
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		
v=t+u			

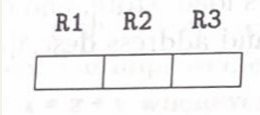
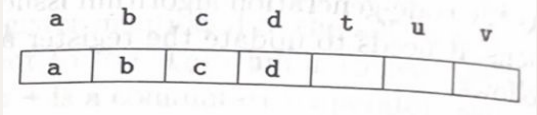

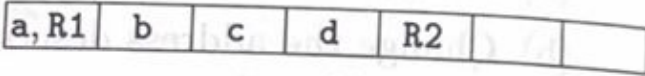
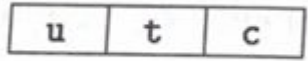
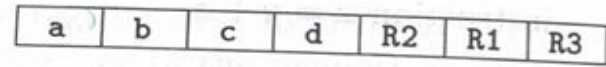
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		
v=t+u	ADD R3, R2, R1		

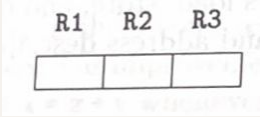
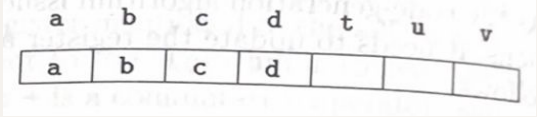
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		
v=t+u	ADD R3, R2, R1		
v=t+u	ADD R3, R2, R1		

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2		
u=a-c	LD R3, c SUB R1, R1, R3		
v=t+u	ADD R3, R2, R1		

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor 	Address Descriptor 
a=d			

Managing Register & Address Descriptors

4. Copy ($x = y$)

x.

4. When we process a copy statement $x = y$, after generating the load for y into register R_y , if needed, and after managing descriptors as for all load statements (per rule 1):

- (a) Add x to the register descriptor for R_y .
- (b) Change the address descriptor for x so that its only location is R_y .

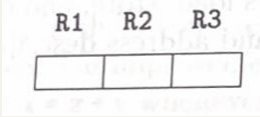
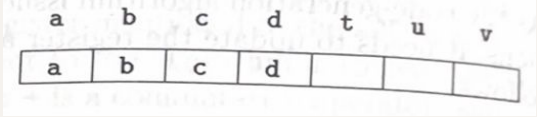
Block consisting of the three-address

Managing Register & Address Descriptors

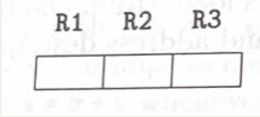
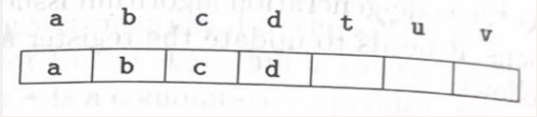
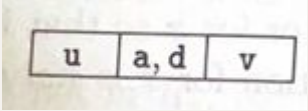
1. Load (**LD** R , x)

- For the instruction LD R , x
- (a) Change the register descriptor for register R so it holds only x .
 - (b) Change the address descriptor for x by adding register R as an additional location.

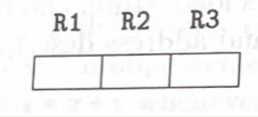
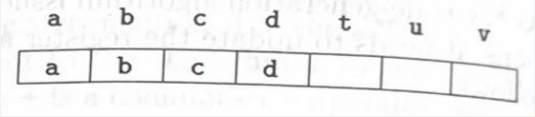
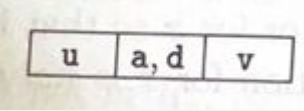
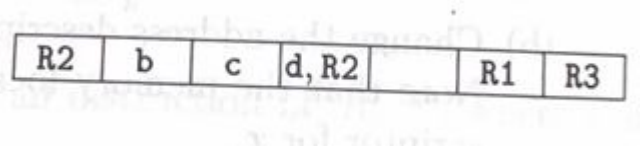
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
			
a=d	<code>LD R2, d</code>		

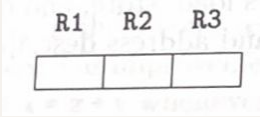
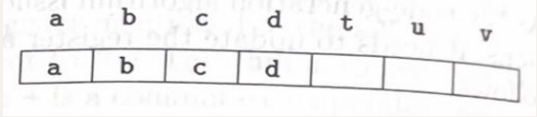
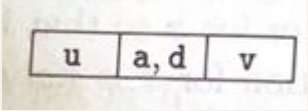
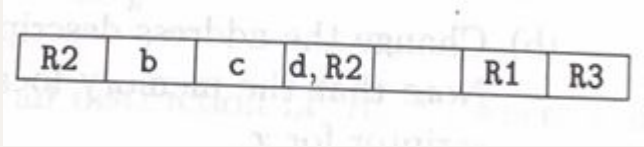
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
			

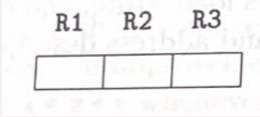
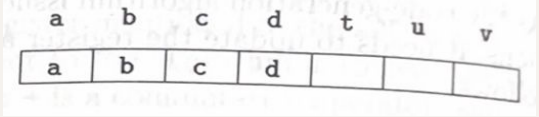
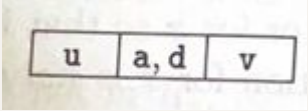
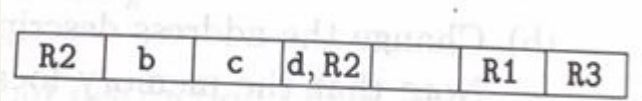
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
			

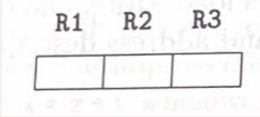
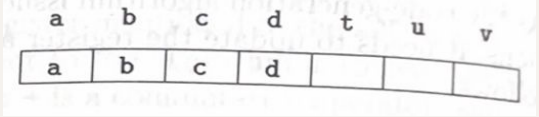
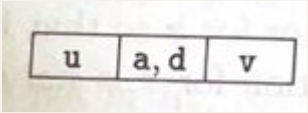
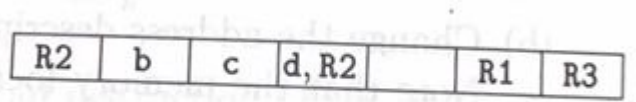
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u			

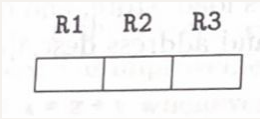
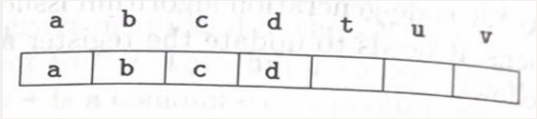
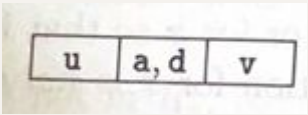
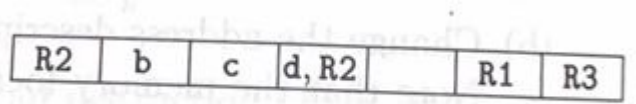
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		

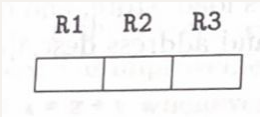
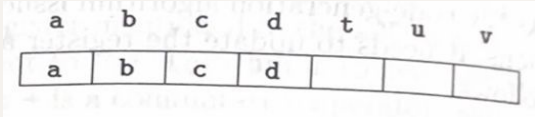
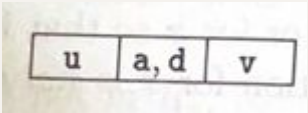
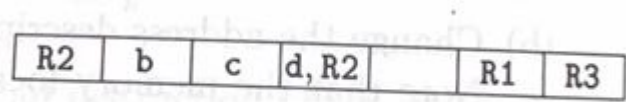
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		

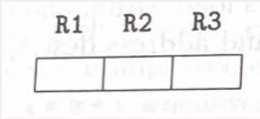
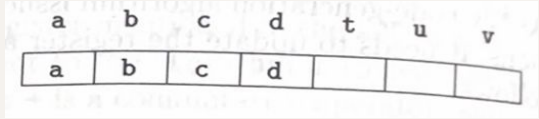
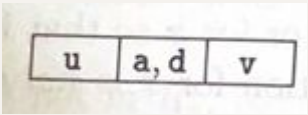
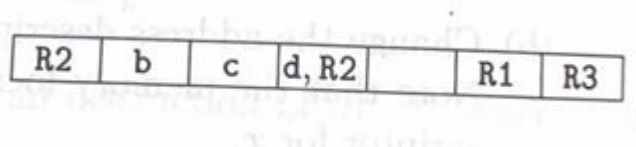
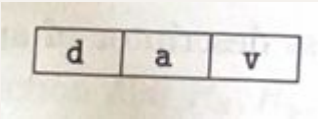
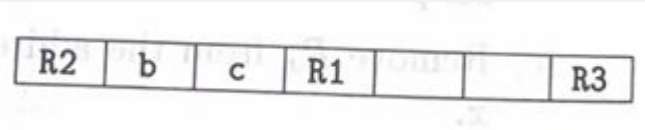
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		

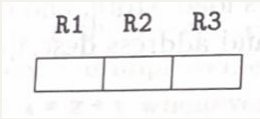
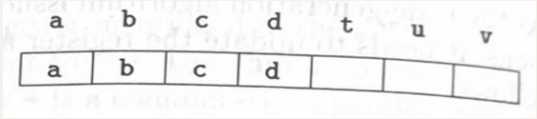
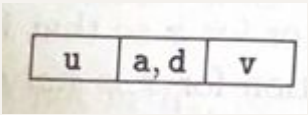
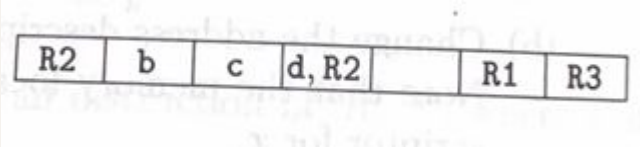
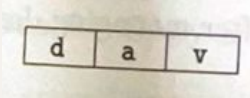
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		
exit			

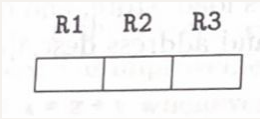
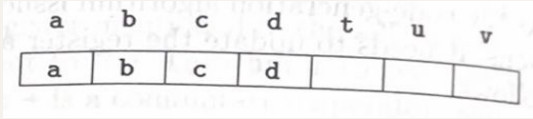
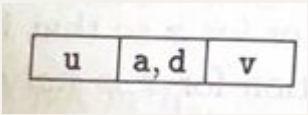
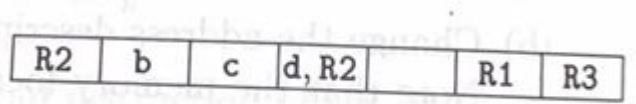
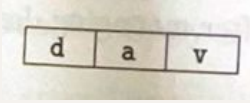
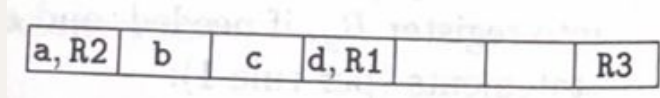
Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		
exit	ST a, R2 St d, R1		

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		
exit	ST a, R2 St d, R1		

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d		
d = v + u	ADD R1, R3, R1		
exit	ST a, R2 St d, R1		

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor (Initially all the three registers - R1, R2 and R3 are empty)	Address Descriptor (Initially a,b,c and d are stored in the memory)
t=a-b	LD R1, a LD R2, b SUB R2, R1, R2	R1 contains a R2 contains b R2 contains t and R1 contains a	a is in R1 b is in R2 t is in R2 and a is in R1
u=a-c	LD R3, c SUB R1, R1, R3	R3 contains c R1 contains u	c is in R3 u is in R1
v=t+u	ADD R3, R2, R1	R3 contains v	v is in R3

Example:

Three - address instruction	Generated machine-code instruction	Register Descriptor	Address Descriptor
a=d	LD R2, d	R2 contains d and a	d and a are in R2
d = v + u	ADD R1, R3, R1	R1 contains d	d is in R1
exit	ST a, R2 St d, R1	No change in register contents	a and d are stored in memory

THANK YOU!
