

Department of Computer Science & Engineering

SUBJECT: Compiler Design

Subject Code: 18PC1CS11

Topic Name: Code Generation

III year-I sem, sec:

Dr.M.Gangappa

Associate Professor

Email: gangappa_m@vnrvjiet

<Web link of your created resource if any>

UNIT -6 Agenda



1. Role of the code generator
2. Issues in the design of code generator
3. Object code formats
 - Assembly code
 - Relocatable code
 - Absolute code
4. Target machine description, cost of an instruction set
5. Register allocation and assignment techniques
6. Simple code generation algorithm
7. GENERATING CODE FROM DAGs
 - Rearranging order
 - Heuristic ordering (node listing algorithm)
 - Labelling Algorithm
8. Machine dependent code optimization(peephole optimization)

Role of the Code generator

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

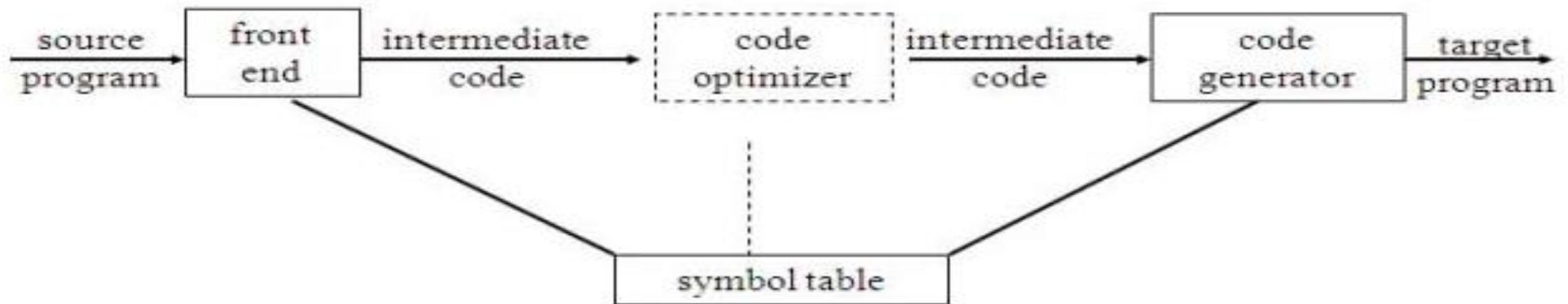


Fig. : Position of code generator

Issues in the design of a code generator

The code generator needs to address so many issues before actually generating the code. The following issues arise during the code generation phase:

1. Input to code generator
2. Target program → ~~2 marks~~ (object code formats) Imp
3. Memory management → ~~4 marks~~ (~~stack, stack, heap~~) unit ~~storage organization / allocation~~
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator

The code generator gets the three-address code as input. The format in which this three-address code is fed as input need to be decided. The following are some of the ways of feeding input to the code generator.

- Linear – The input could be a string of characters where we use **postfix notation** to express the input.
- Tables – As discussed in the previous modules, the three-address code is typically represented using **Quadruples, Triples or Indirect triples** and this table could be served as input
- Non-linear – Abstract Syntax tree (**AST**) or Directed Acyclic Graph (**DAG**) could be used as input to the code generator after converting the input into AST or DAG representation

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program or object code formats

The (back-end) code generator of a compiler may generate different forms of code, depending on the machine requirements. The target program code needs to be specified as Absolute code, Relocatable machine code or Assembly language code. The following are some of the forms.

~~exe code (run time)~~ X

1. Absolute machine language : Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed.

2. Relocatable machine code: Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader, to link the separately compiled program segments.

Target program or object code formats



3. Assembly language:

Producing an assembly language program as output makes the process of code generation somewhat easier.

Example:

**MOV R0, R1
ADD R1, R2**



3. Memory management

- Names in the source program are mapped to addresses of data objects in run-time memory.
- Address mapping defines the mapping between intermediate representations to address in the target code.
- These addresses are based on the runtime environment used like **static, stack or heap**.
- The identifiers are stored in symbol table during declaration of variables or functions, along with type.

4. Instruction selection

The factors to be considered during instruction selection are:

- The **uniformity and completeness** of the instruction set.
- **Instruction speed and machine idioms.**
- **Size of** the instruction set.

Instruction selection is important to obtain efficient code. Suppose we need to translate the following three-address code

$$x := y + z$$

The following would be the instructions with a total cost of 6 ($2+2+2$)MW(memory words):

**MOV y,R0
ADD z,R0
MOV R0,x**

4. Instruction selection (continue...)

Consider another instruction **a:=a+1** and if we adopt the same strategy to convert this instruction to target code, the following would be the result with a cost of 6 (2+2+2)MW”

```
MOV a,R0  
ADD #1,R0  
MOV R0,a
```

if the above code is replaced by the following instruction, the cost would be 2 (1 for INC and 1 for “a”) .

```
INC a
```

5. Register allocation and Assignment



One of the important issues in code generation is **register allocations**. The number of registers in any architecture is **limited** to match the number of **variables** in a high-level program. Efficient utilization of the **limited set of registers** is important to generate good code. Registers are assigned by

- Register allocation*** to select the set of variables that will reside in registers at a point in the code
- Register assignment*** to pick the specific register that a variable will reside in

However, finding an optimal register assignment in general is NP-complete.

6. Evaluation order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

Target Machine Description



- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has n general-purpose registers, **R0, R1, . . . , Rn-1**.
- It has two-address instructions of the form:

op source, destination

Where, op is used as an **op-code** and **source** and **destination** are used as a data field.

It has the following **op-codes**:

ADD (add source to destination)

SUB (subtract source from destination)

MOV (move source to destination)



Target Machine Instruction Set



The **source** and **destination** of an instruction can be specified by the combination of **registers** and **memory location** with address modes.

Addressing MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	C+ contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+ contents(R))	1
literal	#c	c	1

Here, cost 1 means that it occupies only one word of memory.

Instruction	Operation	Cost	Cost computation
MOV R0,R1	Store <i>content(R0)</i> into register R1	1	R0, R1 involves 0 cost and cost of 1 is incurred for MOV
MOV R0,M	Store <i>content(R0)</i> into memory location M	2	MOV costs 1 + one memory address in the instruction costs 1 thus totaling to 2
MOV M,R0	Store <i>content(M)</i> into register R0	2	MOV costs 1 + one memory address in the instruction costs 1 thus totaling to 2
MOV 4(R0),M	Store <i>contents(4+contents(R0))</i> into M	3	4(R0) costs 1, memory costs 1 and MOV costs 1 thus totaling to a cost of 3
MOV *4(R0),M	Store <i>contents(contents(4+contents(R0)))</i> into M	3	*4(R0) costs 1, memory costs 1 and MOV costs 1 thus totaling to a cost of 3
MOV #1,R0	Store 1 into R0	2	#1 costs 1, MOV costs 1 which totals to a cost of 2
ADD 4(R0),*12(R1)	Add <i>contents(4+contents(R0))</i> to <i>contents(12+contents(R1))</i>	3	ADD costs 1, 4(R0) costs 1, *12(R1) costs 1 thus totaling to 3

Instruction costs :

Instruction cost = $1 + \text{cost for source and destination address modes}$.

Examples:

1. Find the cost of Moving register to memory $R0 \rightarrow M$

MOV R0, M

cost = $1 + 0 + 1 = 2$ MW (memory words)

2. Indirect indexed mode:

MOV * 4 (R0), M

cost = 1 plus indirect index plus instruction word
 $= 1 + 1 + 1 = 3$

3. Indexed mode:

MOV 4(R0), M

cost = $1 + 1 + 1 = 3$

4. Literal mode:

MOV #1, R0

cost = $1 + 1 = 2$

5. Move memory to memory

MOV m, m

cost = $1 + 1 + 1 = 3$

Register allocation and assignment

Instructions involving only **register operands** are faster than those involving **memory operands**. Therefore, efficient utilization of registers is vitally important in **generating good code**. This section presents **various strategies** for deciding at each point in a program **what values** should reside in registers (register allocation) and in **which register** each value should reside (register assignment). The following are the register allocation and assignment strategies:

- 1 **Global Register Allocation** ✓
- 2 **Usage Counts** ✓
- 3 **Register Assignment for Outer Loops** ✓
- 4 **Register Allocation by Graph Coloring** ✓

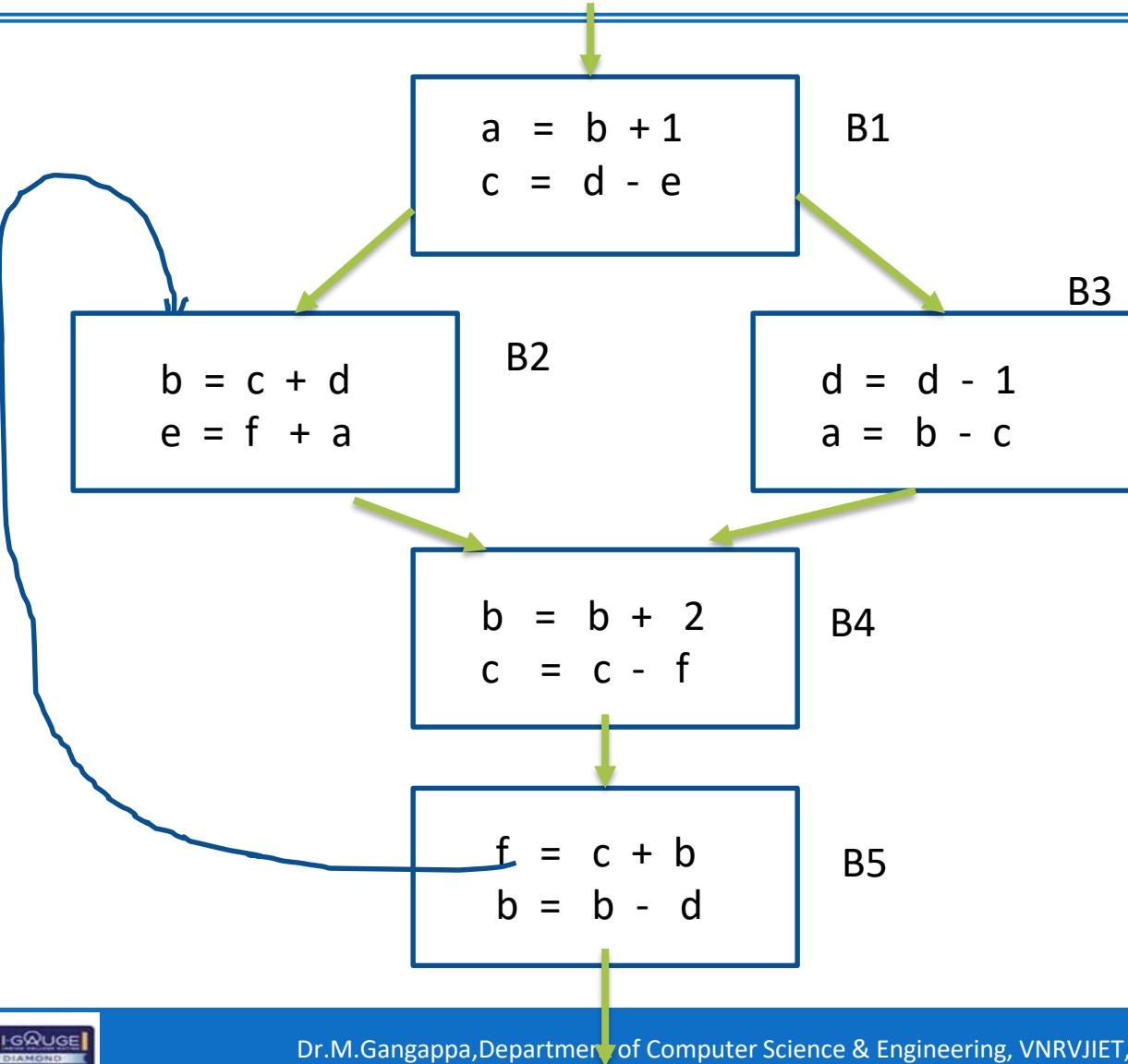
Global Register Allocation



The following are the strategies of global register allocation :

- The global register allocation has a strategy of **storing the most frequently used variables in *fixed registers*.**
- Another strategy is to assign **some fixed number of global registers** to hold the ***most active values*** in each inner loop.
- Registers **not already allocated** may be used to hold values local to one block .
- With early C compilers, a programmer could do some register allocation explicitly by using register declarations to keep certain values in registers for the duration of a procedure.

2.Usage count



The formula to compute **usage count** of a **variable** x in block B of a flow graph F is given by

$$\sum_{\forall B \text{ in } F} [use(x, B) + 2 * live(x, B)]$$

Formulas for usage count method



In the 3AC, $x = y + z$, the variable x is called **defined** and the variables y and z are called **used** variables.

$\text{Use}(x, B)$ = No of times variable x is used in basic block B before defining x in B.

$\text{Live}(x, B)$ is computed using the formula

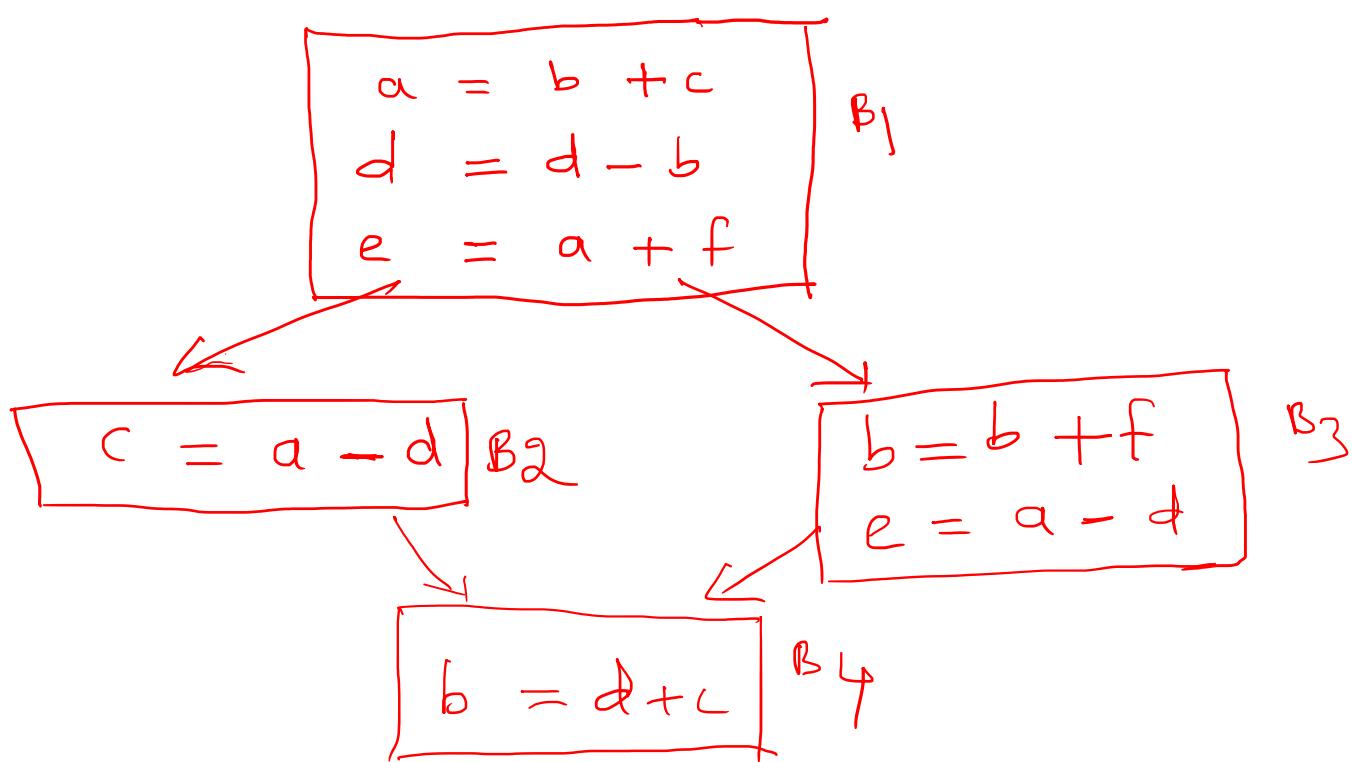
$$x = \begin{cases} 1 & \text{if } x \text{ is live} \\ 0 & \text{if } x \text{ is killed} \end{cases}$$

B_1

$$\begin{aligned} x &= x + y \\ a &= y + b \\ z &= z + y \end{aligned}$$

$$\begin{aligned} \text{use}(x, B_1) &= 1 \\ \text{use}(y, B_1) &= 2 \\ \text{use}(a, B_1) &= 0 \\ \text{use}(b, B_1) &= 1 \end{aligned}$$

Find the usage count of each variable in the flow graph
and assign the register(s) (R_0, R_1).



HW
submit — on or before
04/02/2021

Compute the usage count for variable “a”

Basic Block	Use(a, B)	Live(a, B)	Usage count
B1	0	1	2
B2	1	0	1
B3	0	1	2
B4	0	0	0
B5	0	0	0
Total count		5	

Compute the usage count for variable “b”

Basic Block	Use(b, B)	Live(b, B)	Usage count
B1	1	0	1
B2	0	1	2
B3	1	0	1
B4	1	1	3
B5	2	1	4
Total count		11	

Final usage count table

Assuming that there are only Three registers available :

Variable	Usage Count	Register or Memory assignment
a	5	M (Memory)
b	11	R1
c	08	R2
d	06	R3
e	03	M (Memory)
f	04	M(Memory)



3.Register assignment for outer loop

Machine instructions:

1. Store : is used to store the data from register to memory / variable.

MOV R0, a or MOV R0,M

2. Load : is used to load the data from variable to register .

MOV a, R0

$$R_0 = \text{data}$$

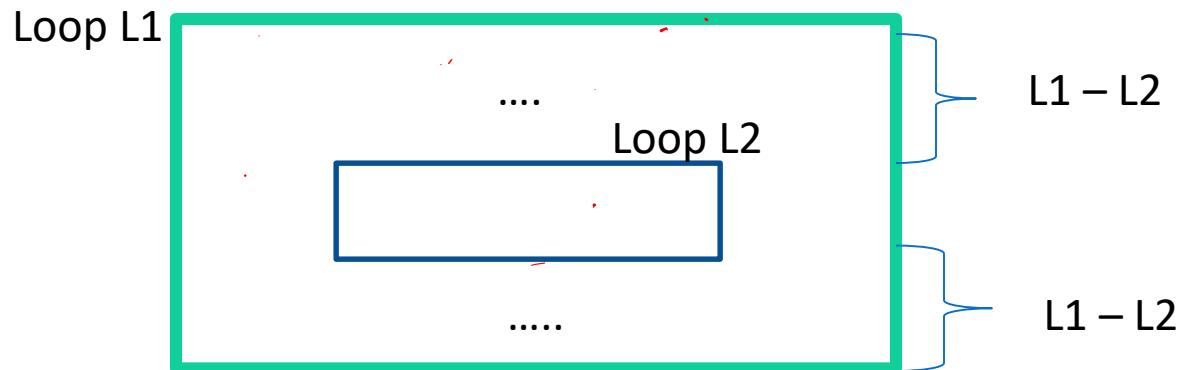
$$R_0 \rightarrow a$$

$$R_0 \rightarrow M$$

Register assignment for outer loop:

Consider that there are two loops, L1 is an outer loop and L2 is an inner loop and allocation of variable “a” is to be done some registers. The approximate scenario is shown in the next slide.

3.Register assignment for outer loop



The following are the criteria for register assignment for outer loop:

1. If “a” is allocated in Loop L2 , then it should not be allocated in L1-L2.
2. If “a” is allocated in L1 and it is not allocated in L2, then store “a” on the entrance of L2 and Load “a” while leaving L2.

**MOV R0, a
MOV a, R0**

3. If “a” is allocated in L2 and it is not in L1, then load “a” on the entrance of L2 and store “a” on the exit from L2.

**MOV a, R0
MOV R0, a**

4.Register allocation by graph coloring



Global register allocation can be seen as a graph coloring problem.

Basic idea:

1. Identify the **live range of each variable**
2. Build an **interference graph** that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
3. Try to **assign as many colors** to the nodes of the graph as there are registers so that two neighbors have different colors

Live range of variable determination



3AC for a code fragment

```
e = d + a  
f = b + c  
f = f + b  
if 10 <= e goto _L0  
d = e + f  
Goto _L1;  
  
_L0:  
    d = e - f  
  
_L1:  
    g = d
```

flow graph

```
e = d + a  
f = b + c  
f = f + b  
if 10 <= e
```

```
d = e + f
```

```
d = e - f
```

```
g = d
```

live variables

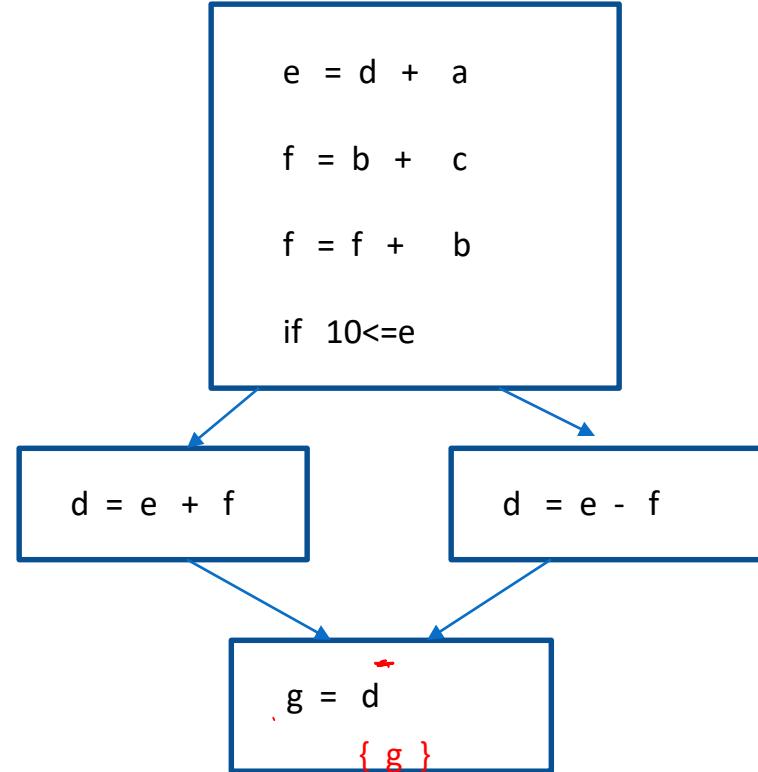
Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;

_L0:
d = e - f

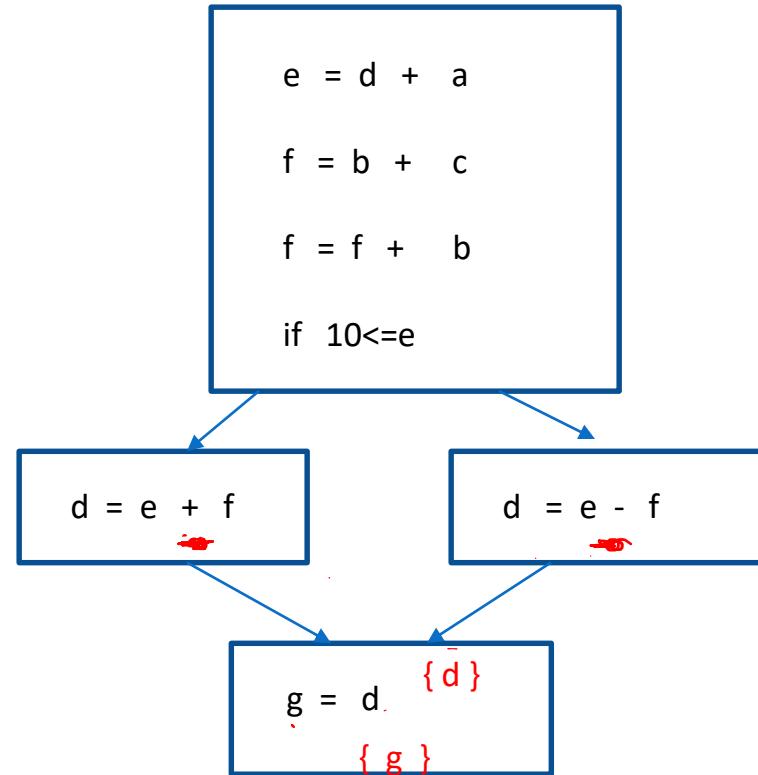
_L1:
g = d
  
```



Live variable determination



```
e = d + a  
f = b + c  
f = f + b  
if 10 <= e goto _L0  
d = e + f  
Goto _L1;  
  
_L0:  
    d = e - f  
  
_L1:  
    g = d
```



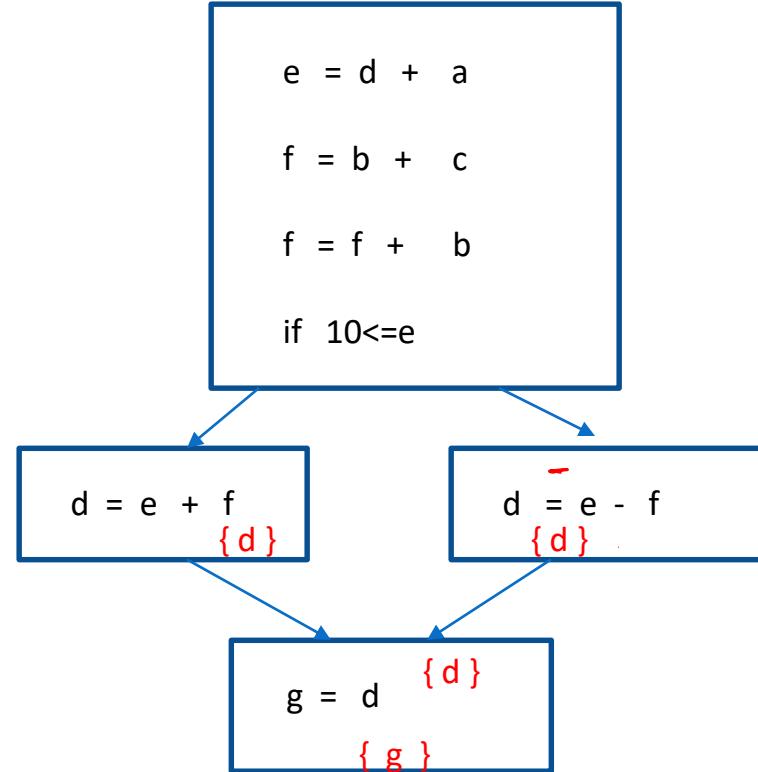
Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;

_L0:
d = e - f

_L1:
g = d
  
```



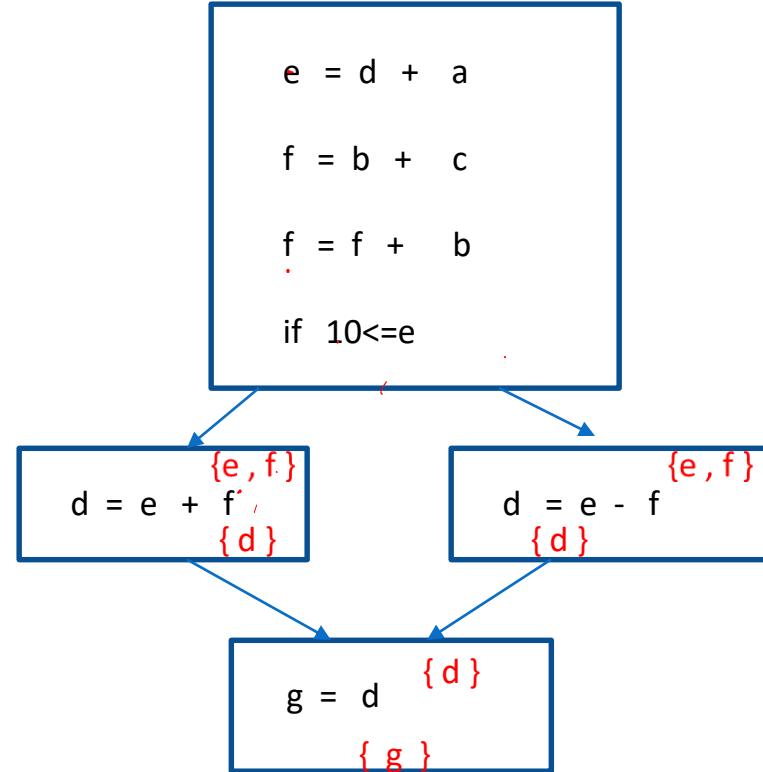
Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;

_L0:
d = e - f

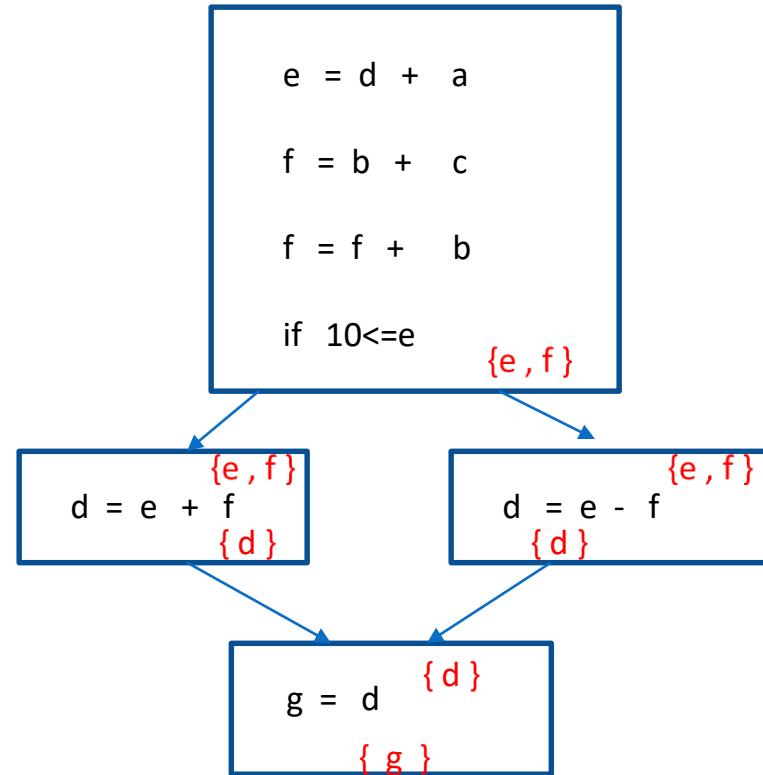
_L1:
g = d
  
```



Live variable determination



```
e = d + a  
f = b + c  
f = f + b  
if 10 <= e goto _L0  
d = e + f  
Goto _L1;  
  
_L0:  
    d = e - f  
  
_L1:  
    g = d
```



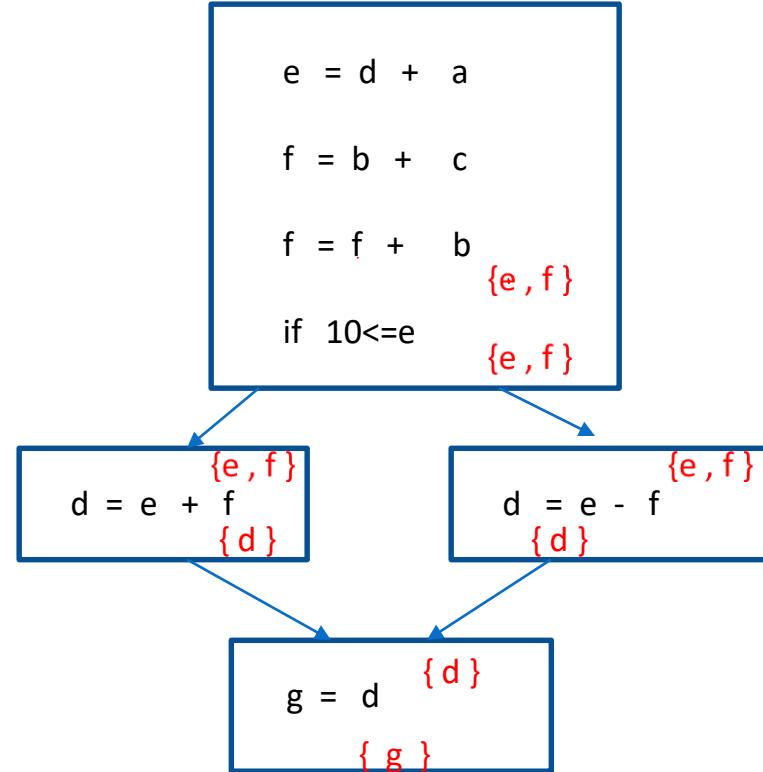
Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;

_L0:
d = e - f

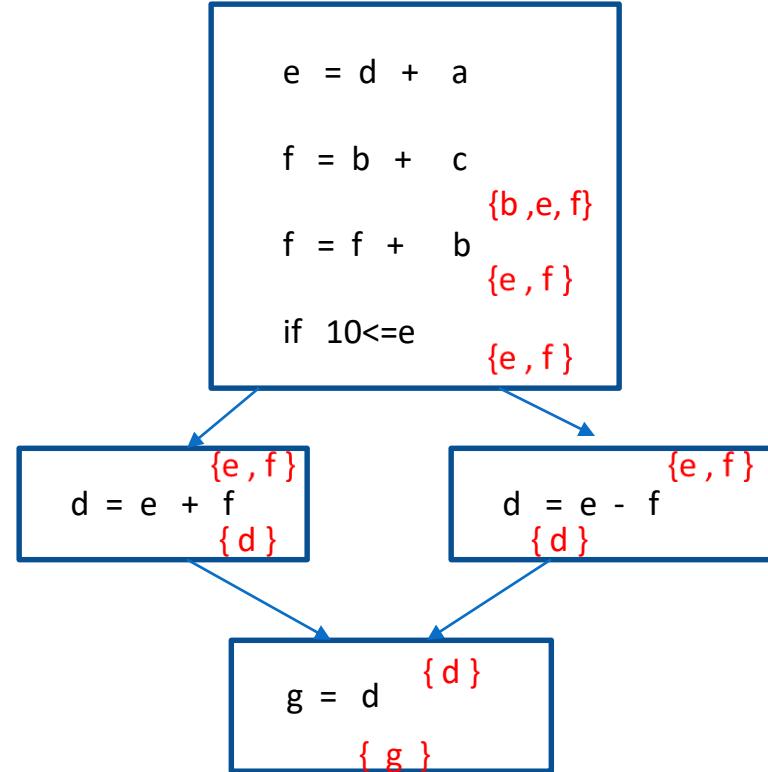
_L1:
g = d
  
```



Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;
_L0:
d = e - f
_L1:
g = d
  
```



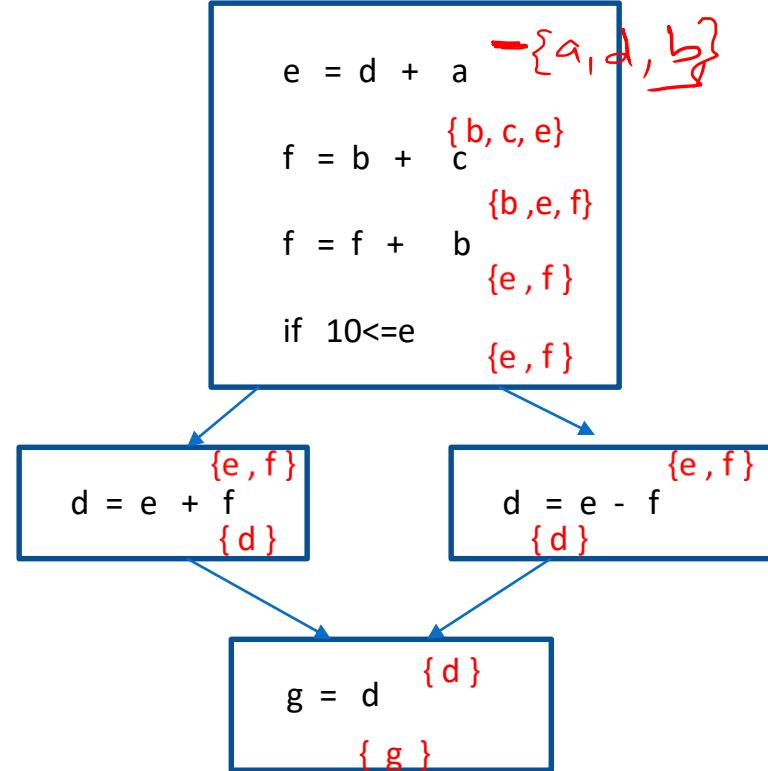
Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;

_L0:
d = e - f

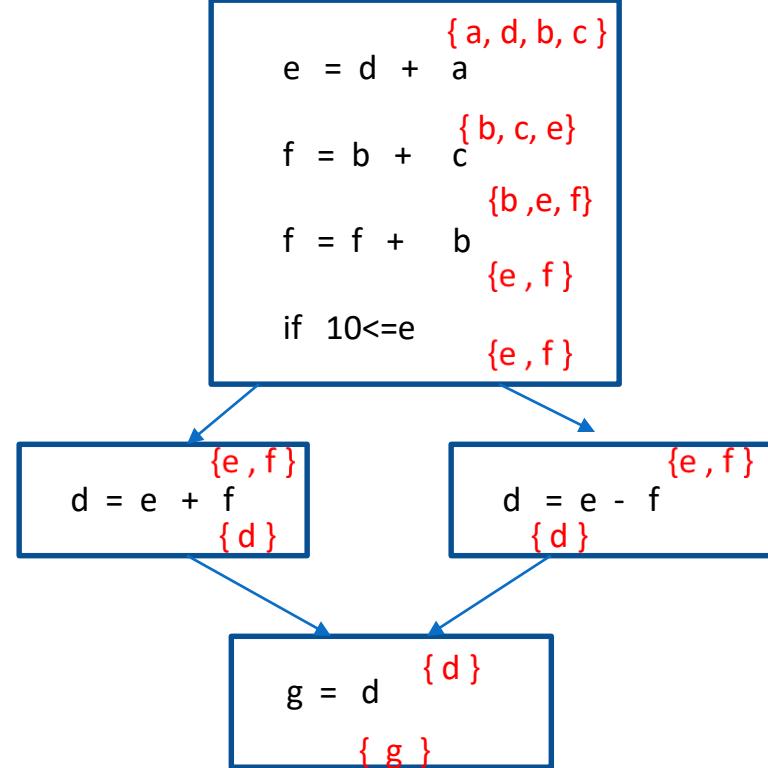
_L1:
g = d
  
```



Live variable determination

```

e = d + a
f = b + c
f = f + b
if 10 <= e goto _L0
d = e + f
Goto _L1;
_L0:
d = e - f
_L1:
g = d
  
```



The Register Interference Graph



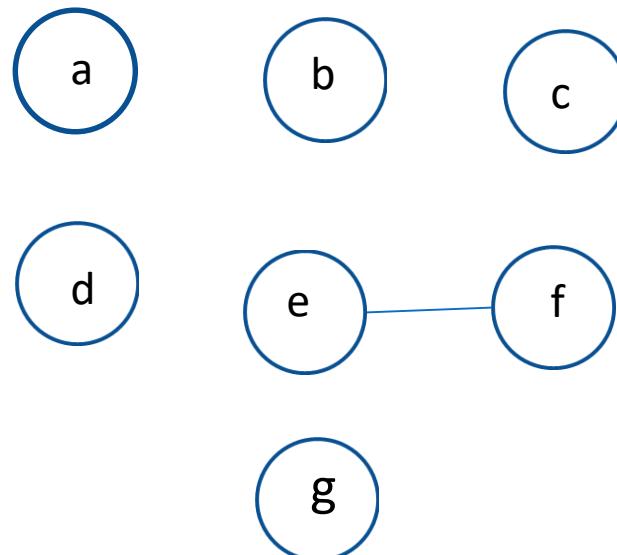
- The **register interference graph** (RIG) of a control-flow graph is an undirected graph where
 - Each node is a variable.
 - There is an edge between two variables that are live at the same program point.
- Perform register allocation by assigning each variable a different register from all of its neighbors.

write all compatibility blocks and draw the interference graph

The following are the computability blocks :

{g}, {e, f}, {b, e, f}, {b, e, c}, {a, d, b, c}
`)

Draw the computability diagram

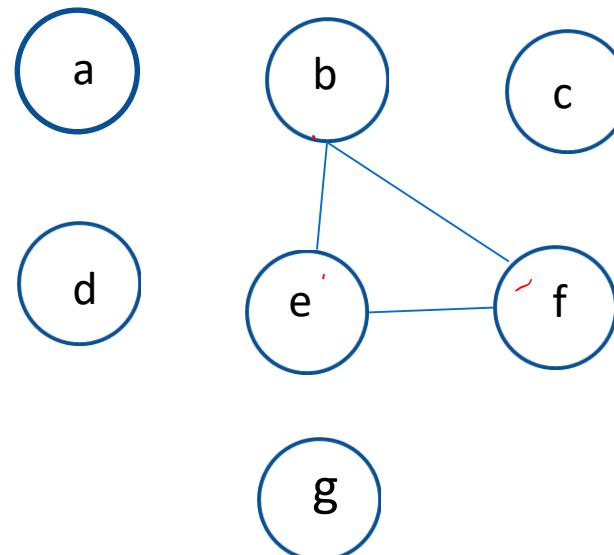


write all compatibility blocks and draw the interference graph

The following are the computability blocks :

{g}, {e, f}, {b, e, f}, {b, e, c}, {a, d, b, c}

Draw the computability diagram

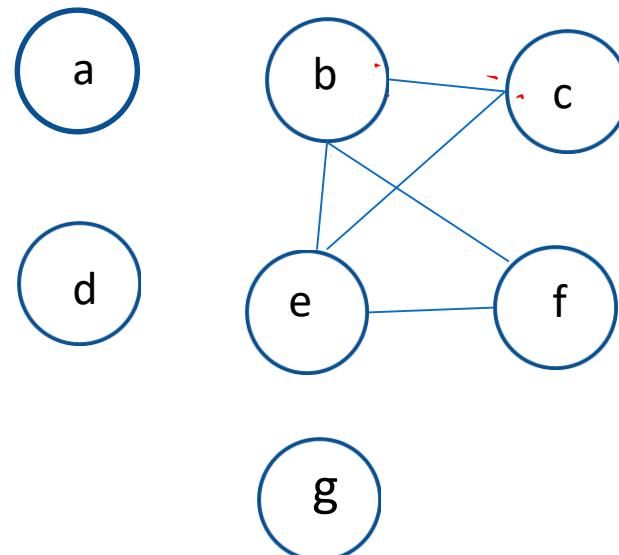


write all compatibility blocks and draw the interference graph

The following are the computability blocks :

{g}, {e, f}, {b, e, f}, {b, e, c}, {a, d, b, c}

Draw the computability diagram

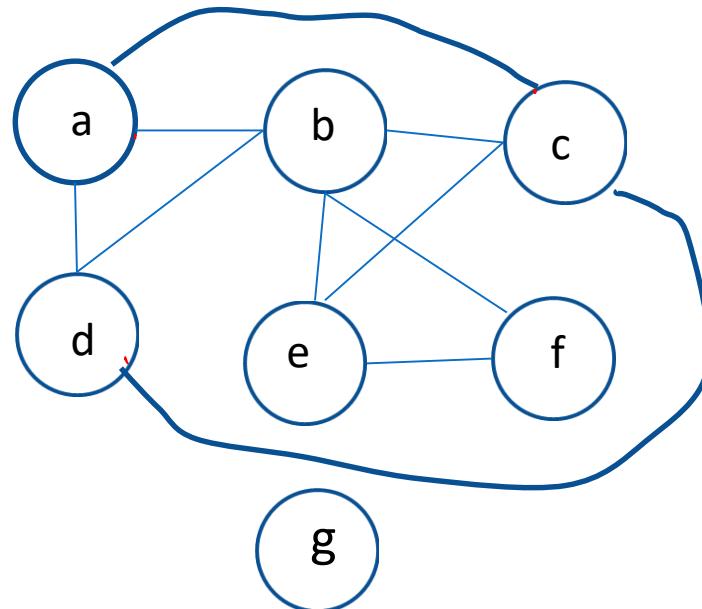


write all compatibility blocks and draw the interference graph

The following are the computability blocks :

{g}, {e, f}, {b, e, f}, {b, e, c}, {a, d, b, c}

Draw the computability diagram



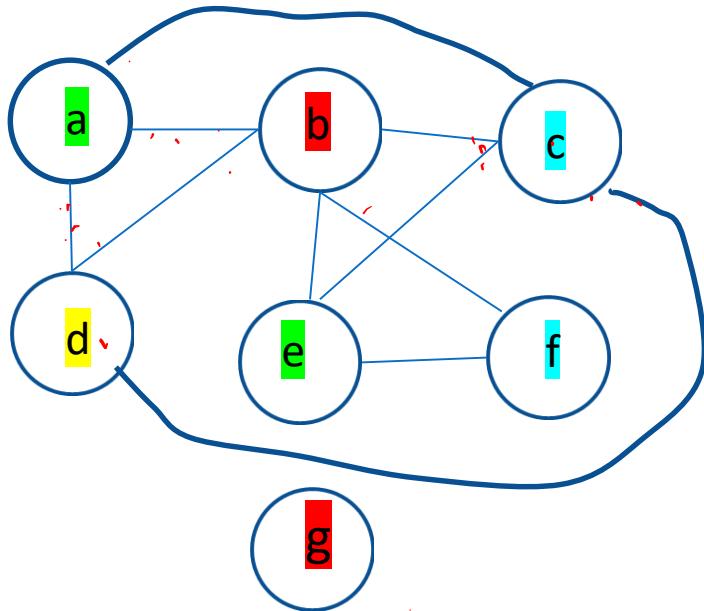
Assign the colors to the nodes of interference graph



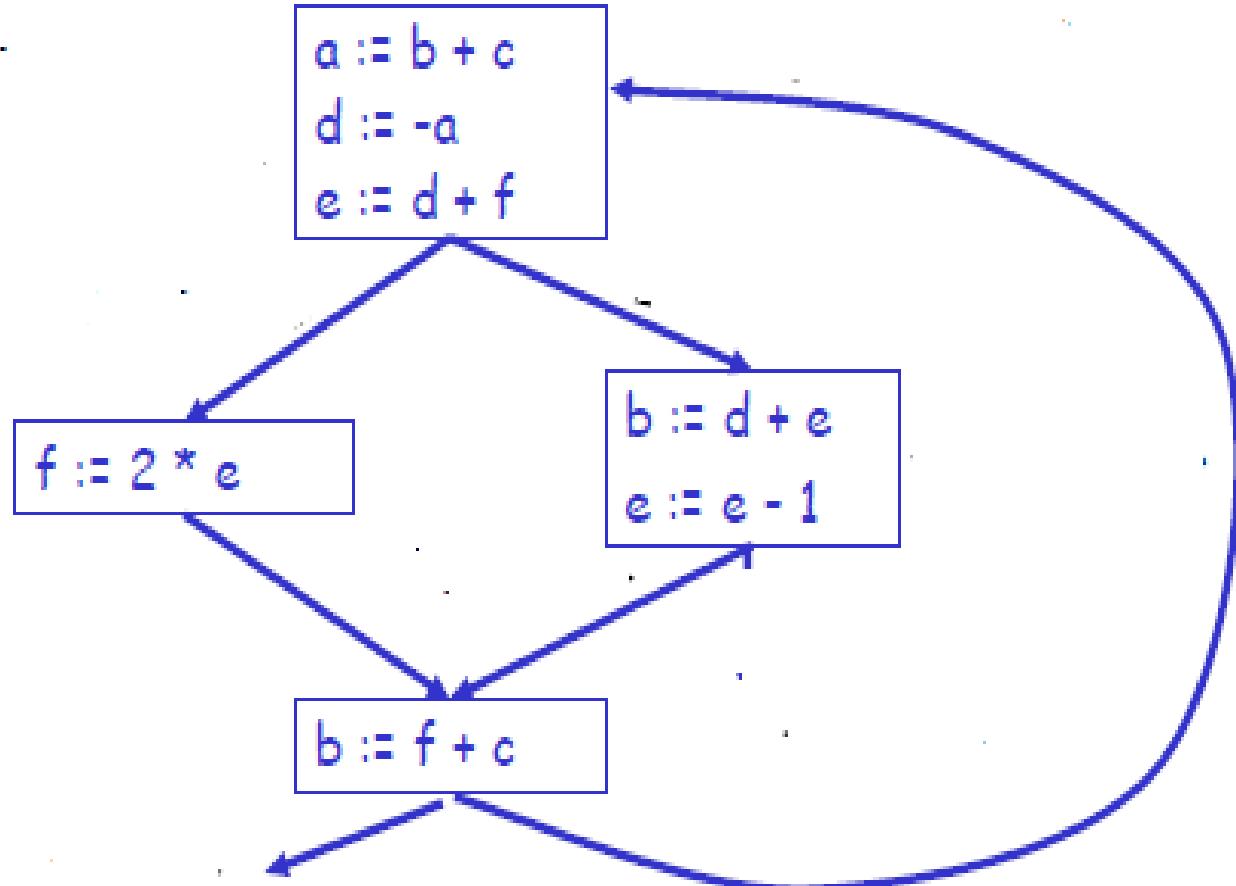
$\{g\}, \{e, f\}, \{b, e, f\}, \{b, e, c\}, \{a, d, b, c\}$

Draw the computability diagram

The number of different colors is 4. Therefore, the minimum registers required is 4. They are R1,R2,R3, and R4.



Find the minimum number of registers required for the given flow of data



submit on or before
06 / 02 / 2021

Simple code generation algorithm



The simple code generator algorithm generates **target code** for a **sequence of three-address statements**. The code generator algorithm works by considering individually all the basic blocks. It uses the next-use information to decide on whether to keep the computation in the **register** or move it to a **variable** so that the register could be reused.

Data structures for the Simple code generator algorithm

This algorithm uses two data structures for generating code :

1. **Address descriptor** : is used to keep track of **the location** where **the current value** of the **variable** can be found at run time.
2. **Register descriptor** : is used to keep track of **which variable** is currently **stored in a register**
- .

The Code Generation Algorithm



Algorithm SimpleCodeGenerator()

Input : Sequence of 3-address statements from a basic block.

Output: Assembly language code

For each statement $x := y \text{ op } z$

1. Set location $L = \text{getreg}(y, z)$ to store the result of $y \text{ op } z$

2. If $y \notin L$ then generate **MOV y', L**

where y' denotes one of the locations where the value of y is available - choose register if possible

3. Generate

OP z', L

where z' is one of the **locations of z**;

Update **register/address descriptor** of x to include L

The Code Generation Algorithm



4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

Algorithm getreg ()

Input: Request for a register

Output: A register or the memory location

1. If y is stored in a register R and R only holds the value y, and y has no next use, then return R;

 Update address descriptor: value y no longer in R

2. Else, return a new empty register if available.

3. Else, find an occupied register R;

 Store contents (register spill) by generating

 MOV R,M

 for every M in address descriptor of y; Return register R.

4. Else Return a memory location.

example



Generate the target code for $d := (a-b) + (a-c) + (a-c)$

Step1 :

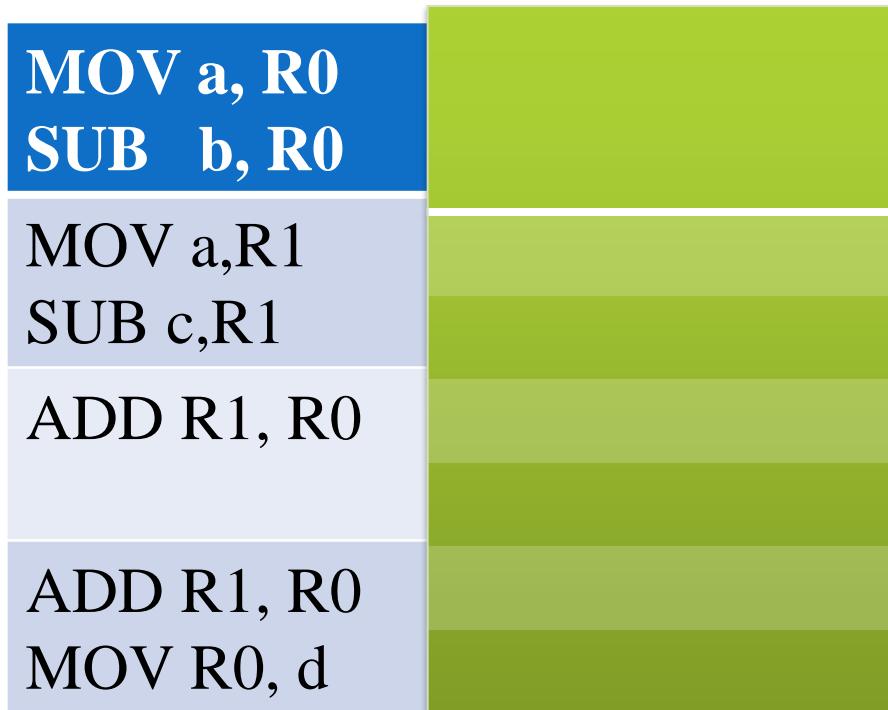
Intermediate code

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

Step 2:

Intermediate code	Target code	Register Descriptor	Address Descriptor
$t = a - b$	MOV a, R0 SUB b, R0	Registers empty R0 contains t	t in R0
$u := a-c$	MOV a,R1 SUB c,R1	R0 contains t R1 contains v	t in R0 u in R1
$v := t + u$	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
$d := v + u$	ADD R1, R0 MOV R0, d	R0 contains d	d in R0

Cost of the machine instructions



Generating code for other types of statements

The code for indexed assignment statements: $a := b[i]$ and $a[i] := b$

Statement	i in register R_i		i in Memory M_i		i in stack	
	code	cost	code	cost	code	cost
$a := b[i]$	MOV $b(R_i)$, R	2	MOV M_i , R MOV $b(R)$, R	4	MOV $Si(A)_i$, R MOV $b(R)$, R	4
$a[i] := b$	MOV $b,a(R_i)$	3	MOV M_i , R MOV $b, a(R)$	5	MOV $Si(A)_i$, R MOV $b, a(R)$	5

Practice Problems:

- Generate the code sequence for the pointer assignments $a = *p$ and $*p = a$, when p is in the register, P is in the Memory and p is in the Stack.
- Generate the code for the conditional statement : if $x < y$ goto z.
- Generate the code for $x := y + z$
if $x < 0$ goto z

1. Generate the code for the assignment statement given below

$$X = (a - b) + (e + (c - d))$$

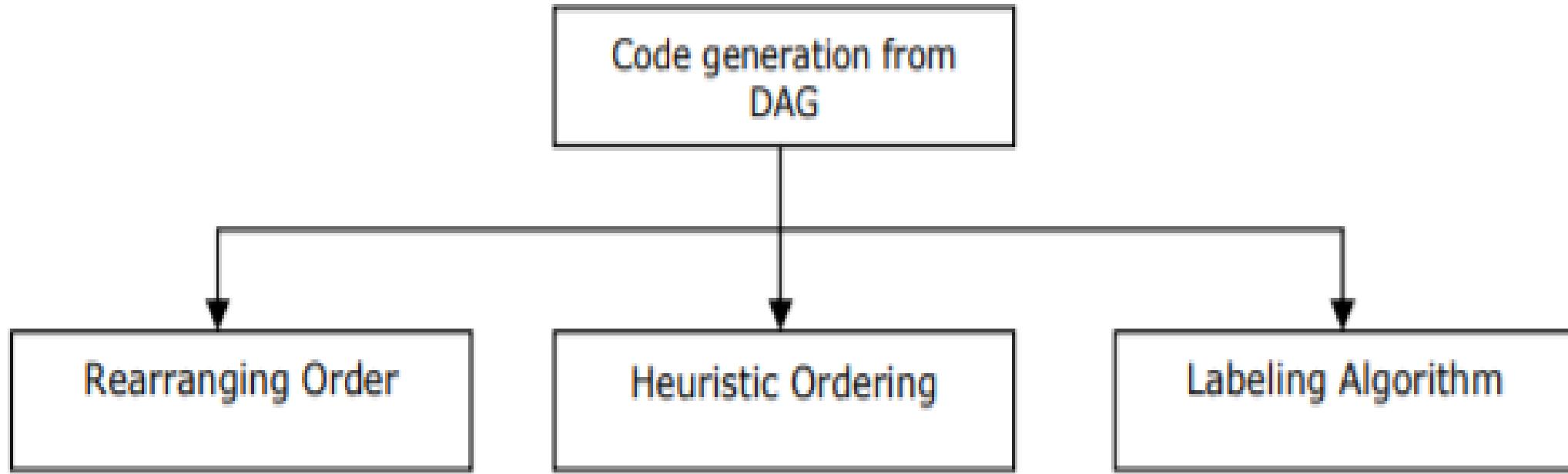
~~HW~~ (22/2/2021)
~~submit~~

2. Generate improved target code for the assignment and also compute the instructions cost:

$$a = b + c$$

$$d = a + e$$

GENERATING CODE FROM DAGs



GENERATING CODE FROM DAGs



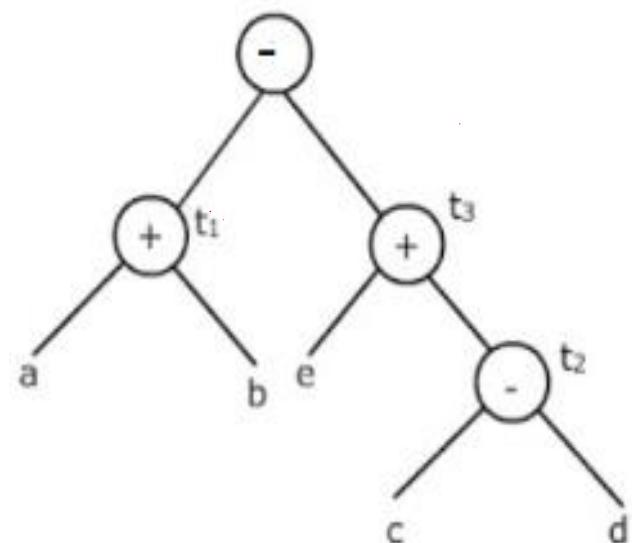
- **Rearranging the order** – To optimize the code generation, the instructions are rearranged and this is referred to as heuristic reordering
- **Labeling the tree** for register information – To know the number of registers required to generate code, the labels of the nodes are numbered which indicate the number of registers required to evaluate that node.
- **Heuristic reordering**– find a solution close to the best one and they find it fast and easily.

1.Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

```
t1 := a + b
t2 := c - d
t3 := e + t2
t4 := t1 + t3
```



DAG for $(a + b) - (e + (c - d))$

The generated code sequence :



```
MOV a , R0
ADD b , R0
MOV c , R1
SUB d , R1
MOV R0 , t1
MOV e , R0
ADD R0 , R1
MOV t1 , R0
ADD R1 , R0
MOV R0 , t4
```

1.Rearranging the order

Rearranged basic block:

Now t1 occurs immediately before t4.

```
t2 := c - d
t3 := e + t2
t1 := a + b
t4 := t1 + t3
```

Revised code sequence:

```
MOV c , R0
ADD d , R0
MOV e, R1
ADD R0 , R1
MOV a , R0
ADD b , R0
ADD R1 , R0
MOV R0 , t4
```

In this order, two instructions **MOV R0 , t1** and **MOV t1 , R0** have been saved.

2. Heuristic ordering for Dags

The term **heuristic** is used for algorithms which **find solutions among all possible ones**, but they do not guarantee that the best will be found, therefore they may be considered as **approximately** and not accurate algorithms. These algorithms, **usually find a solution close to the best one** and they find it fast and easy.

2. Heuristic ordering for Dags

Node listing algorithm

Obtain all interior nodes, Consider these interior nodes as unlisted interior nodes

while (**unlisted interior nodes remain**) do
begin

select an **unlisted node n**, whose **parents have been listed** ;
list n;

while (the **leftmost child m of n** has **no unlisted parents** and is **not a leaf**)do

/* since n was just listed, m is not yet listed*/

begin

list m ;

n = m ;

end

end



2. Heuristic ordering for Dags

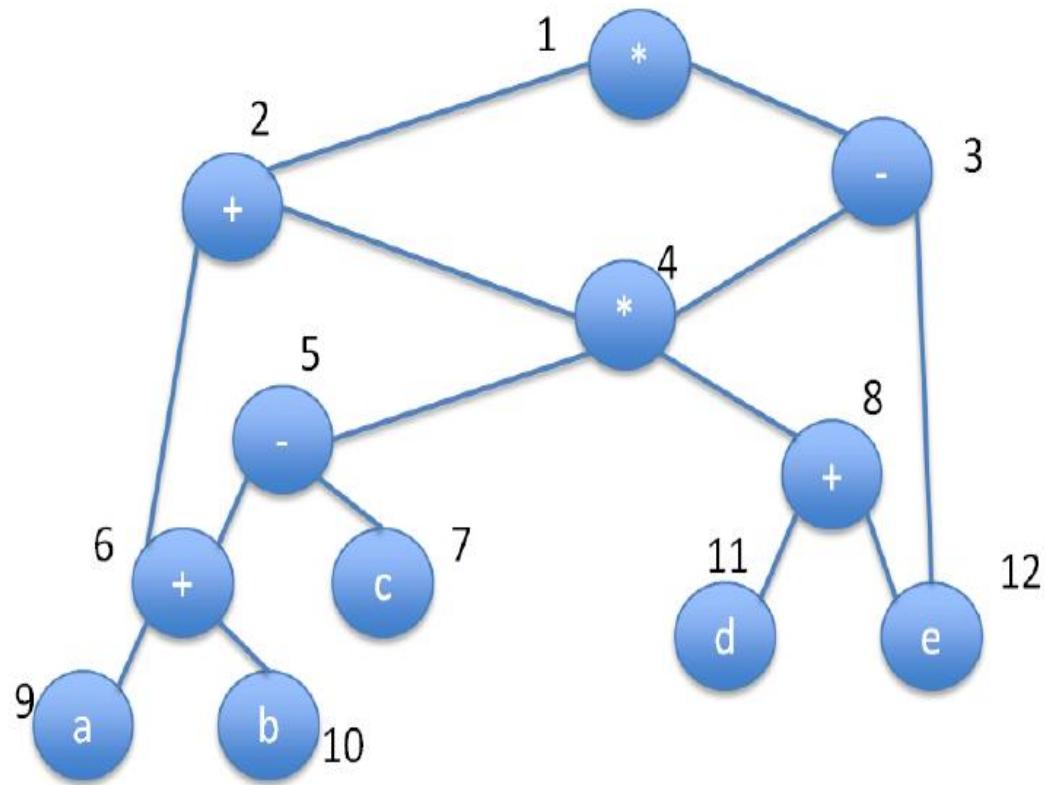


Figure . : DAG heuristic reordering example

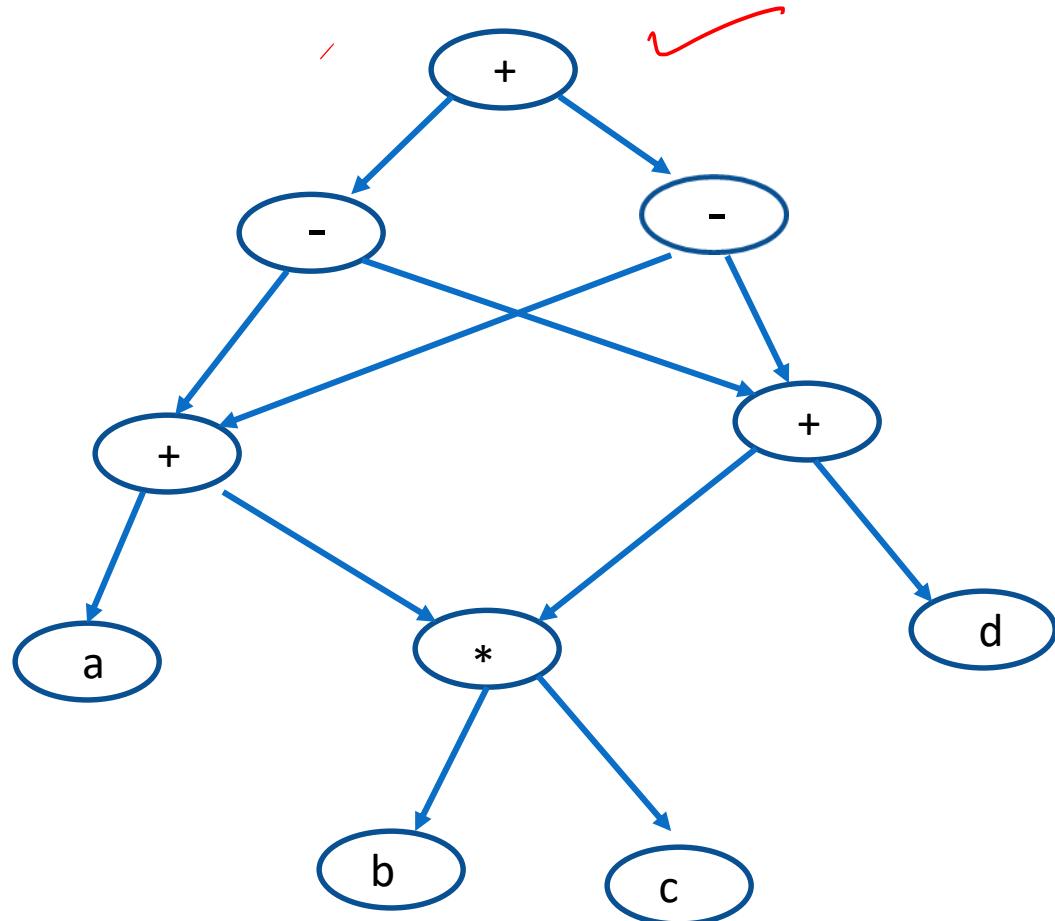
2. Heuristic ordering for Dags



Generated target code:

```
t8 = d + c  
t6 = a + b  
t5 = t6 - c  
t4 = t5 * t8  
t3 = t4 - e  
t2 = t6 + t4  
t1 = t2 + t3
```

Apply the node listing algorithm and generate the target code for



$$\{(a+(b*c)) - ((b*c)+d)\} + \{(a+(b*c)) - ((b*c)+d)\}$$

The Labelling Algorithm — DAG

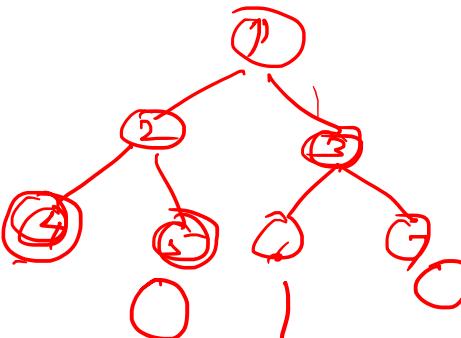


This algorithm works on the tree representation of a sequence of three-address statements. This algorithm has two parts:

- The first part labels each node of the tree from bottom to top, with an integer that denotes the minimum number of registers required to evaluate the tree.
- The second part of the algorithm is a tree traversal that generates the code during the tree traversal.

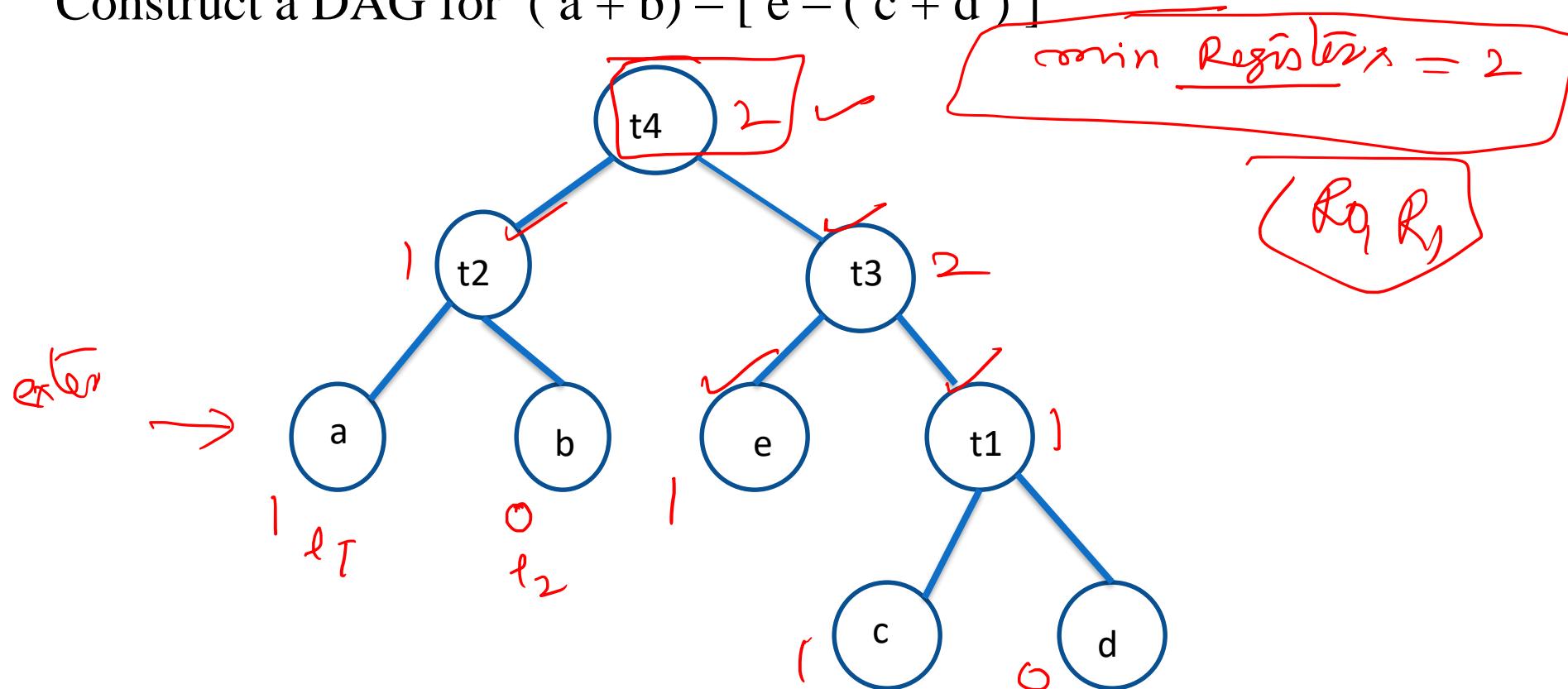
Assigning the labels

- For **leaf nodes (external nodes)**:
 - If n is the left most child of its parent then
 - $\text{Label}(n) = 1$
 - else
 - $\text{Label}(n) = 0$
- For **internal nodes**:
 - $\text{Label}(n) = \begin{cases} \max(l_1, l_2), & l_1 \neq l_2 \\ l_1 + 1, & l_1 == l_2 \end{cases}$
 - Where l_1 is the label for left child and l_2 is label for right child of the parent node n .



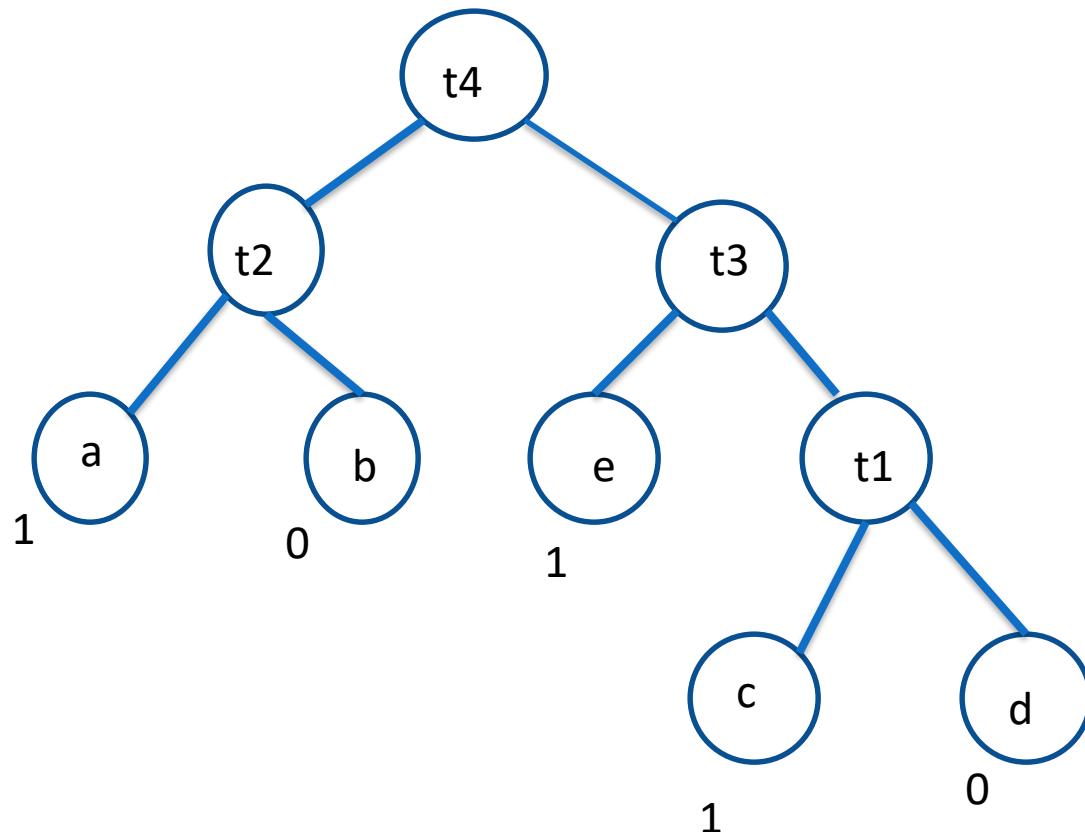
Example:

Construct a DAG for $(a + b) - [e - (c + d)]$



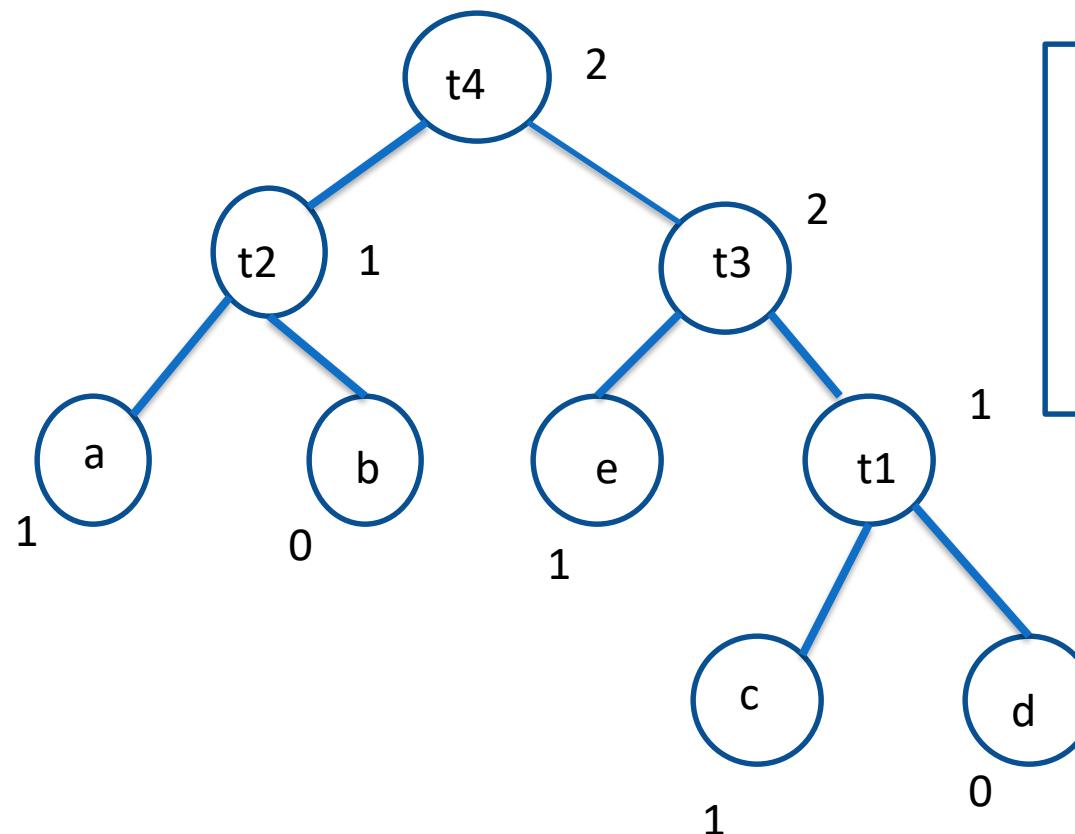
Example:

Construct a DAG for $(a + b) - [e - (c + d)]$



Example:

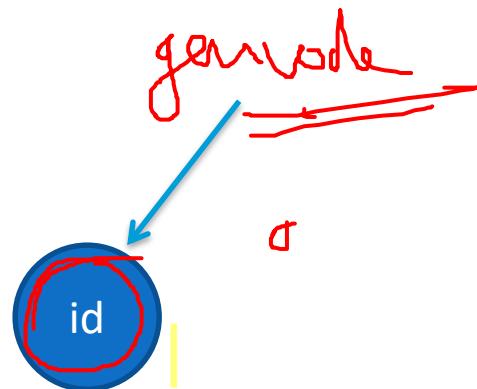
Construct a DAG for $(a + b) - [e - (c + d)]$



Therefore, the minimum number of registers required is 2.

Procedure for Labelling algorithm

Case  0



Case Analysis

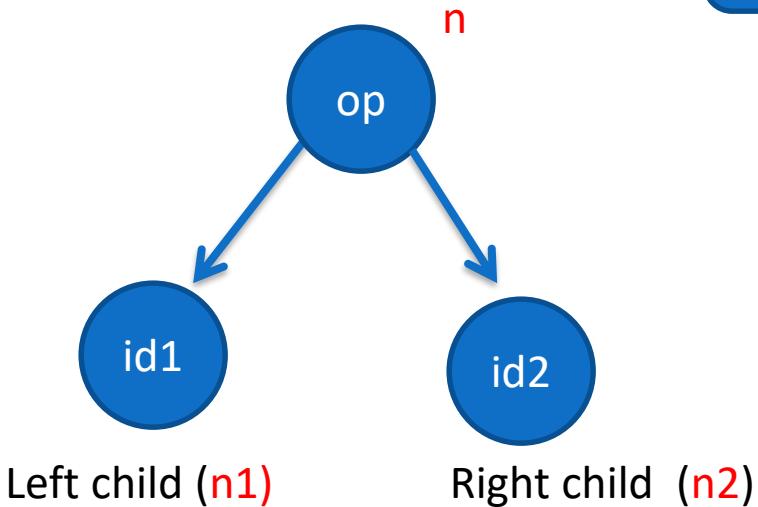
Code :

```
MOV id, Rstack(top)
```

Procedure for Labelling algorithm

Case 1:

Case Analysis



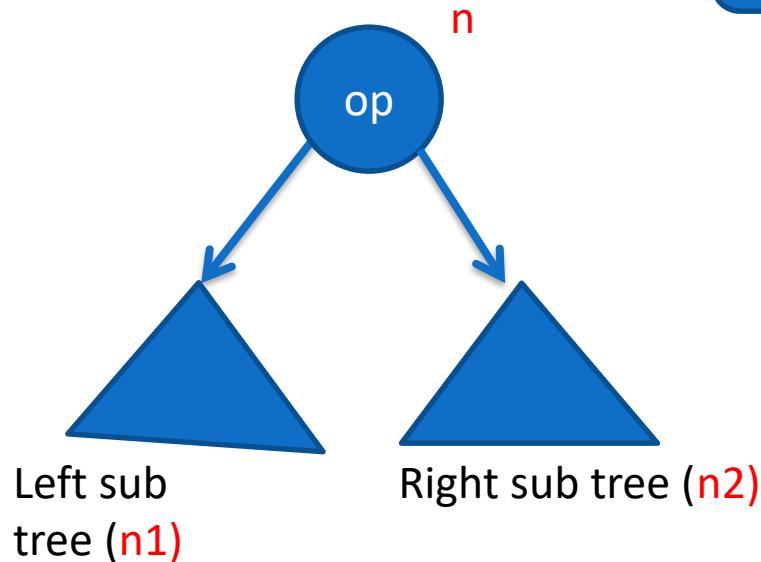
Code :

MOV	$\text{id1}, \text{R}_{\text{stack}(\text{top})}$
Op	$\text{id2}, \text{R}_{\text{stack}(\text{top})}$

Procedure for Labelling algorithm

Case 2:

Case Analysis



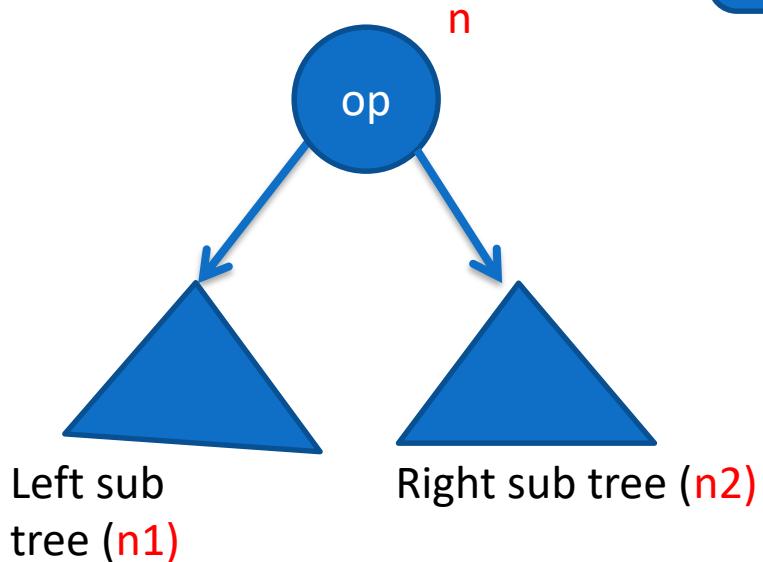
if $I(n2) > I(n1)$ and $I(n) < \text{number of registers}$

1. Swap top two registers
2. `gencode(RST)`
3. `gencode(LST)`

Procedure for Labelling algorithm

Case 3:

Case Analysis



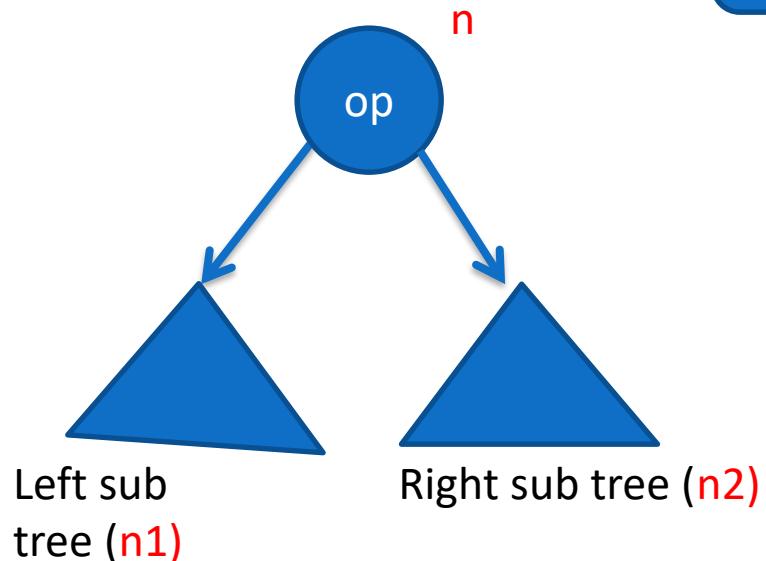
if $I(n_1) < I(n_2)$ and $I(n) <$ number of registers

1. `gencode(LST)`
2. `gencode(RST)`

Procedure for Labelling algorithm

Case 4:

Case Analysis

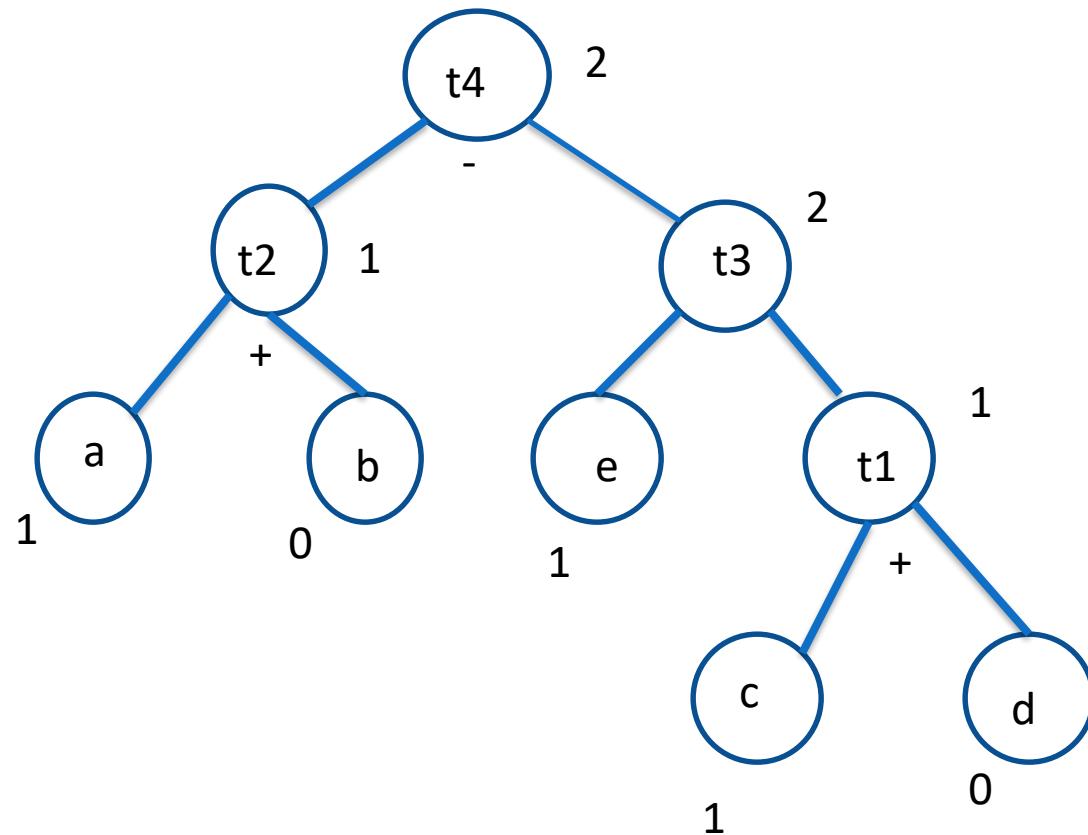


if $I(n_1) = I(n_2)$ and $I(n) <$ number of registers

1. `gencode(LST)`
2. `gencode(RST)`

Example:

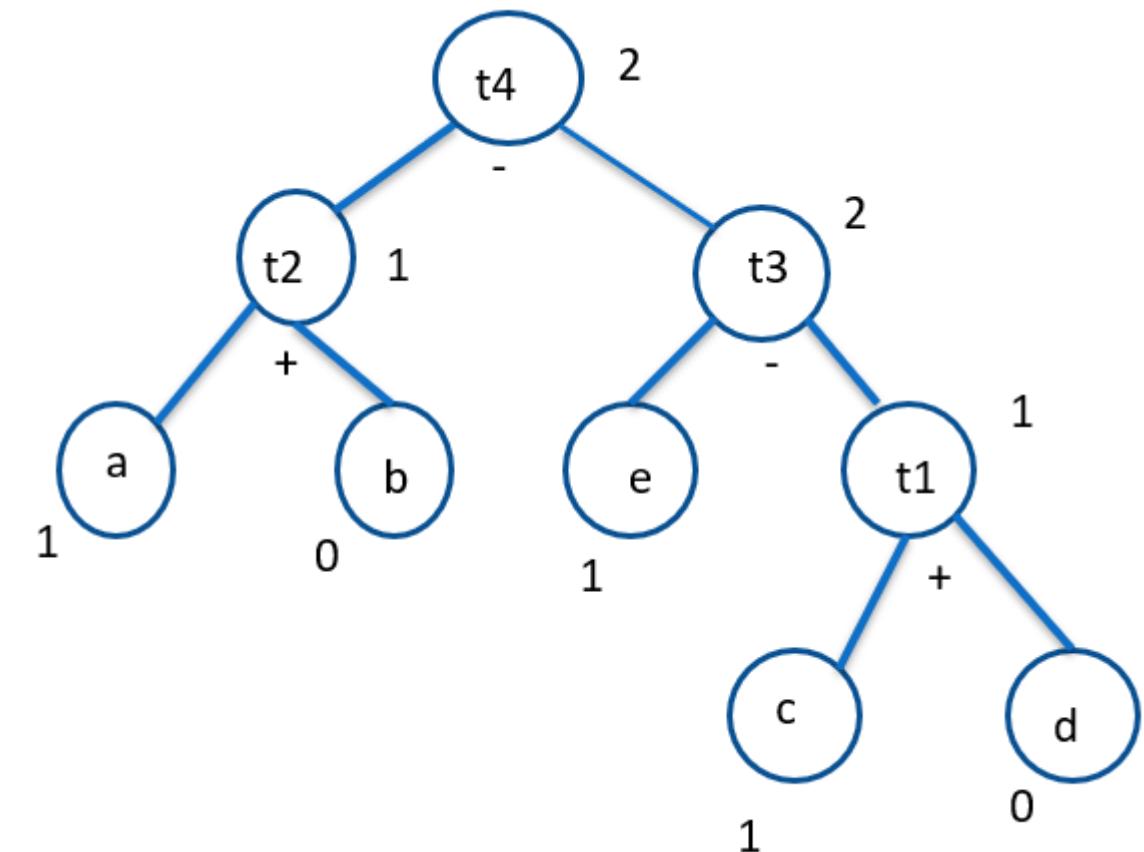
Generate the code for $(a + b) - [e - (c + d)]$



Labelling algorithm for target code generation



Target code

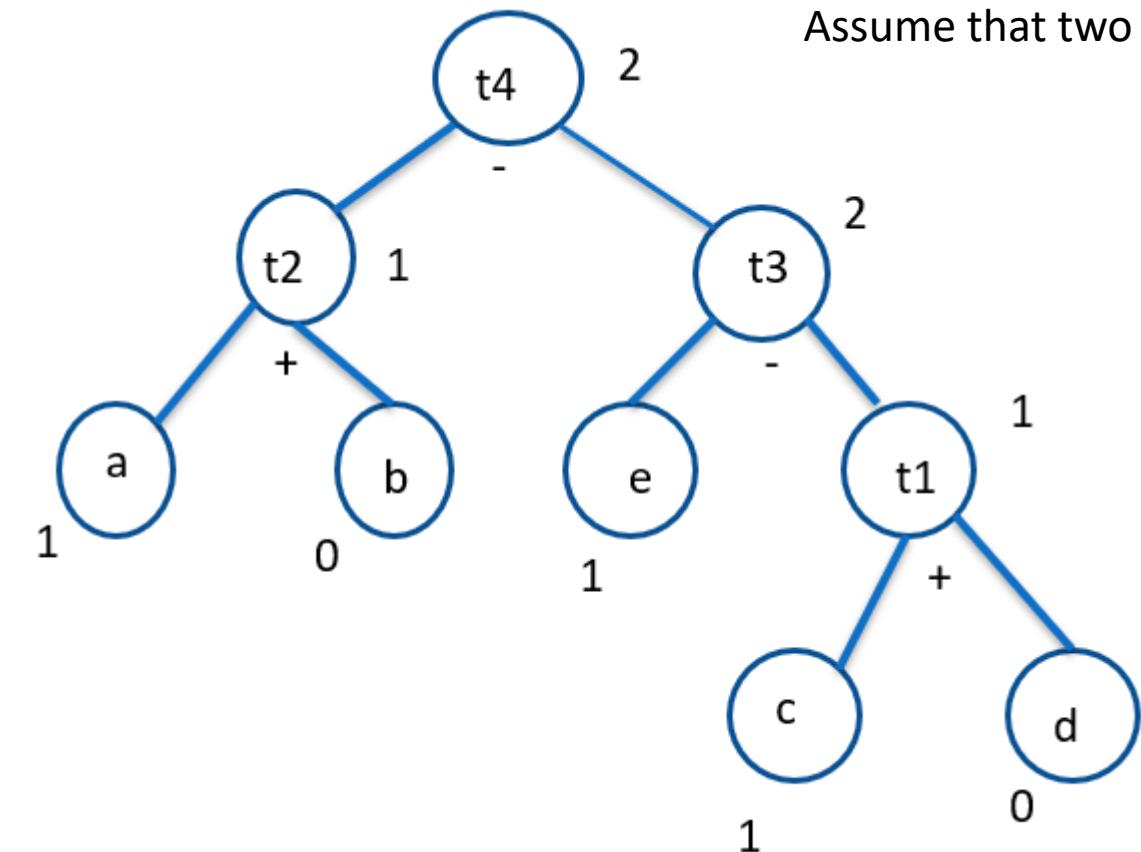


Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

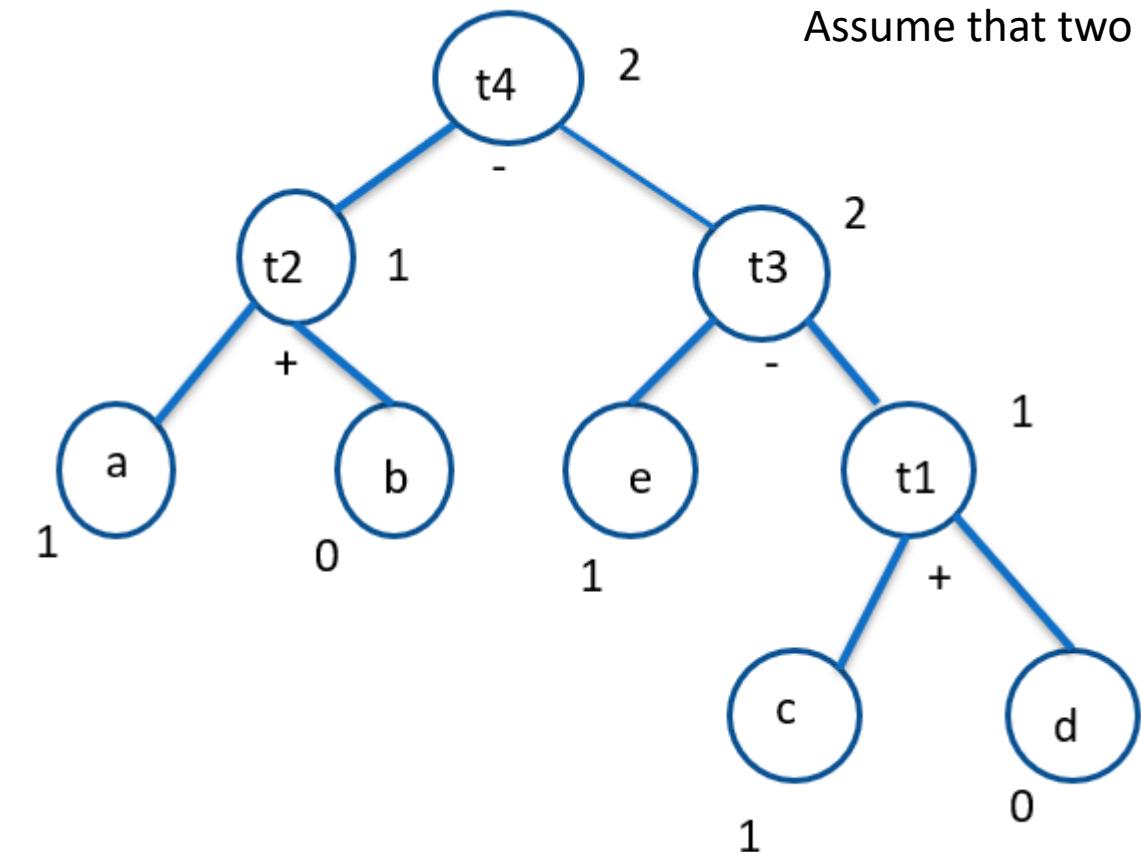
[R0,R1]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

[R0,R1]

gencode(t3)

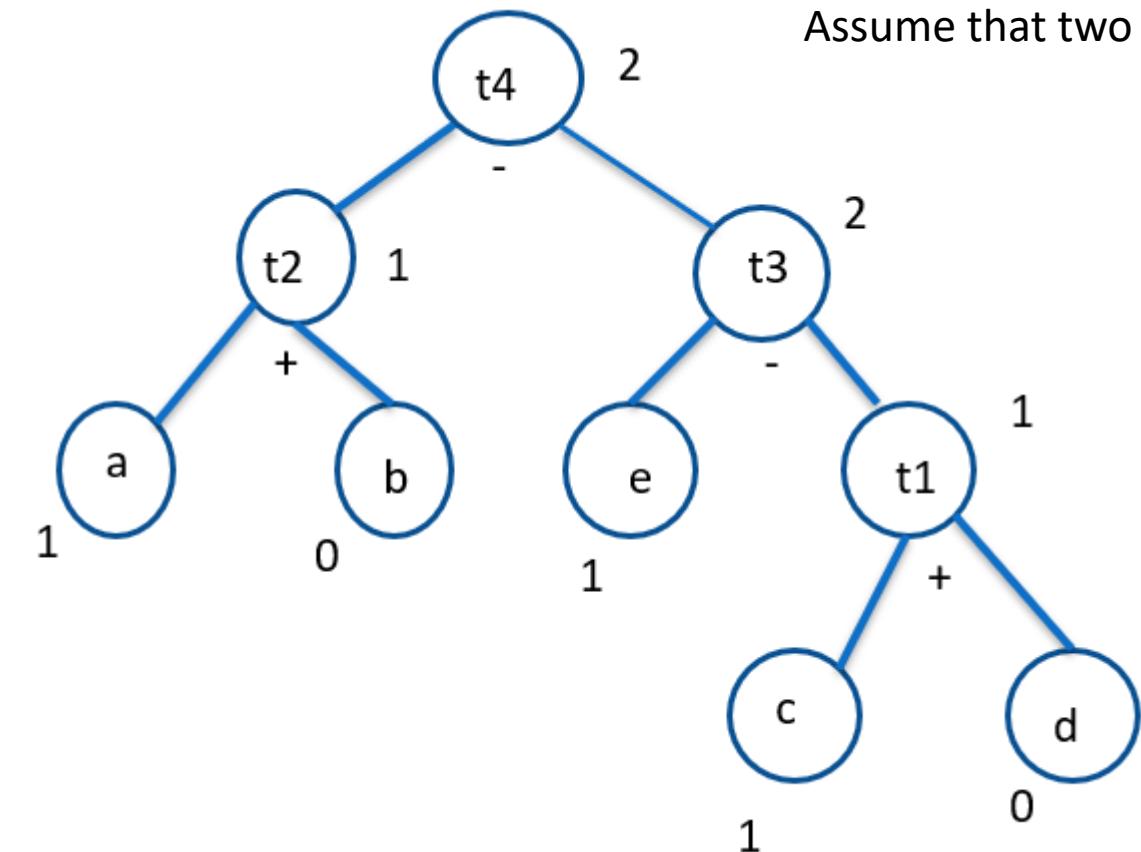
[R1,R0]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4) [R0,R1]

gencode(t3) [R1,R0]

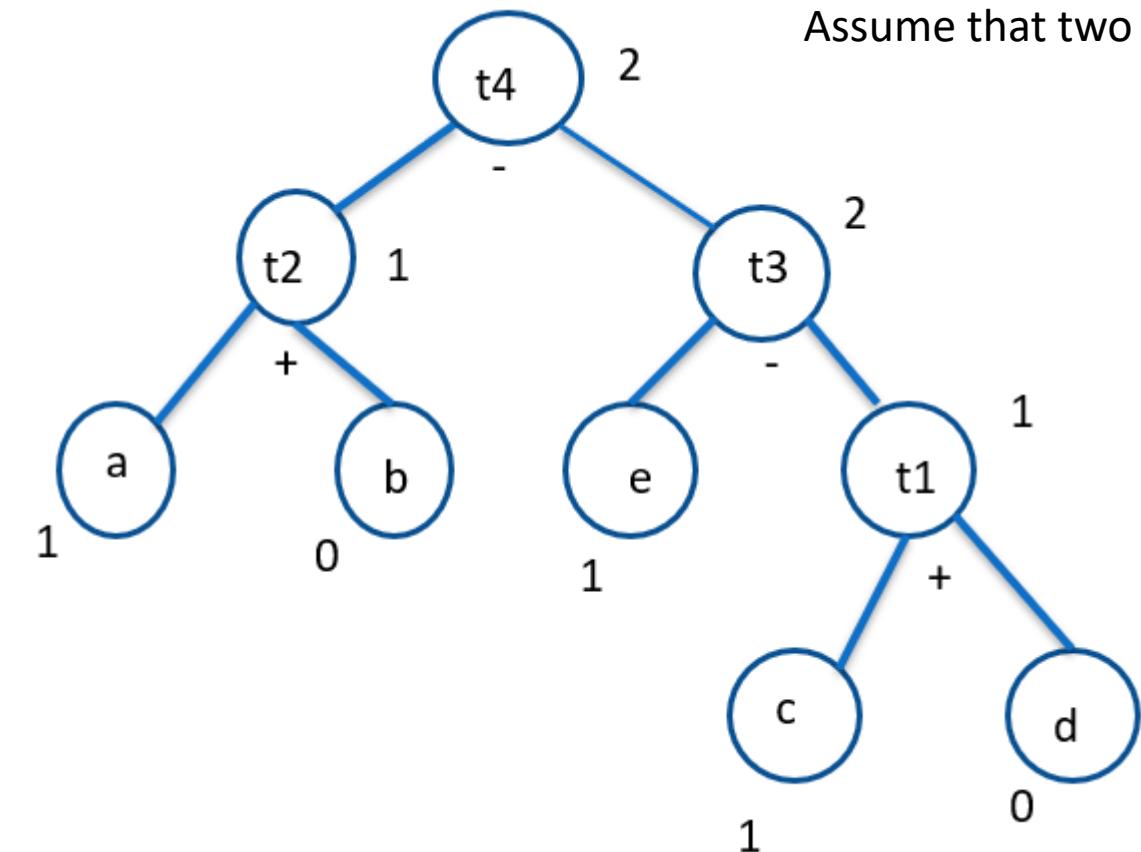
gencode(e) [R1,R0]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4) [R0,R1]

gencode(t3) [R1,R0]

gencode(e) [R1,R0]

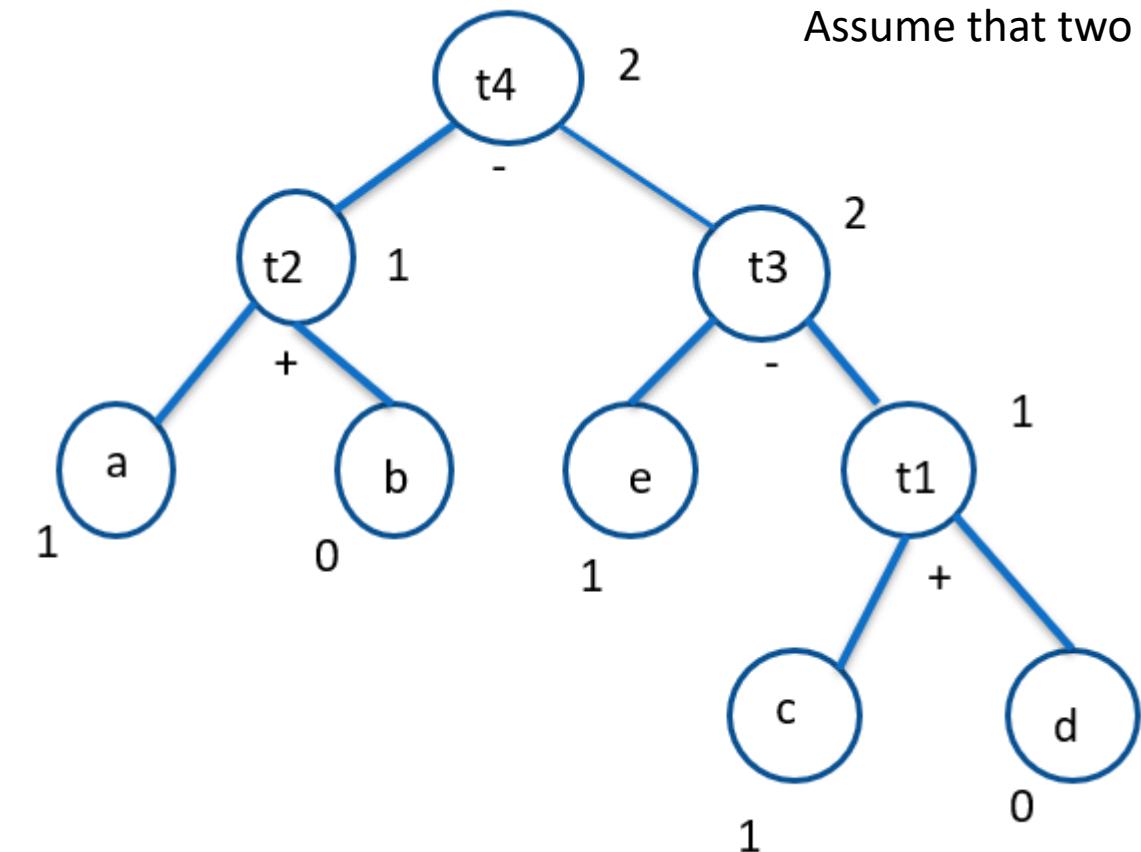
MOV e , R0

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4) [R0,R1]

gencode(t3) [R1,R0]

gencode(e) [R1,R0]

MOV e , R0

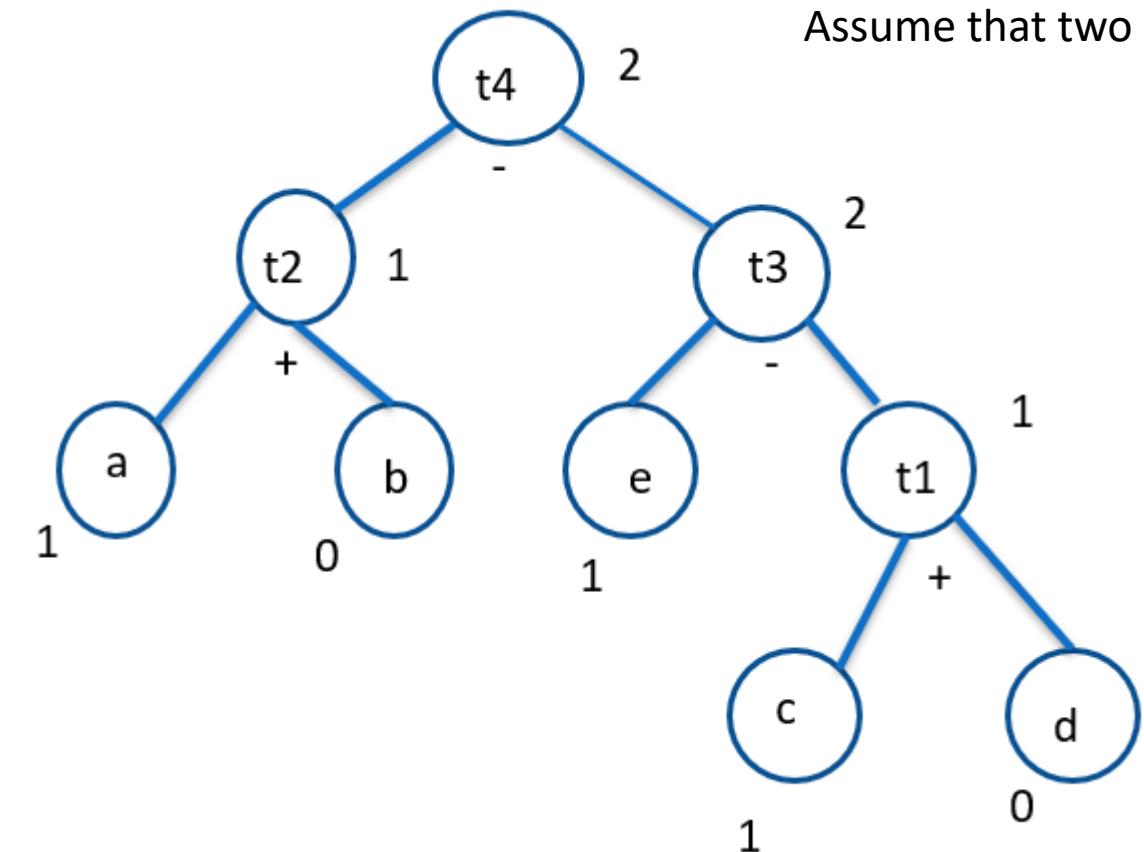
gencode(t1) [R1]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



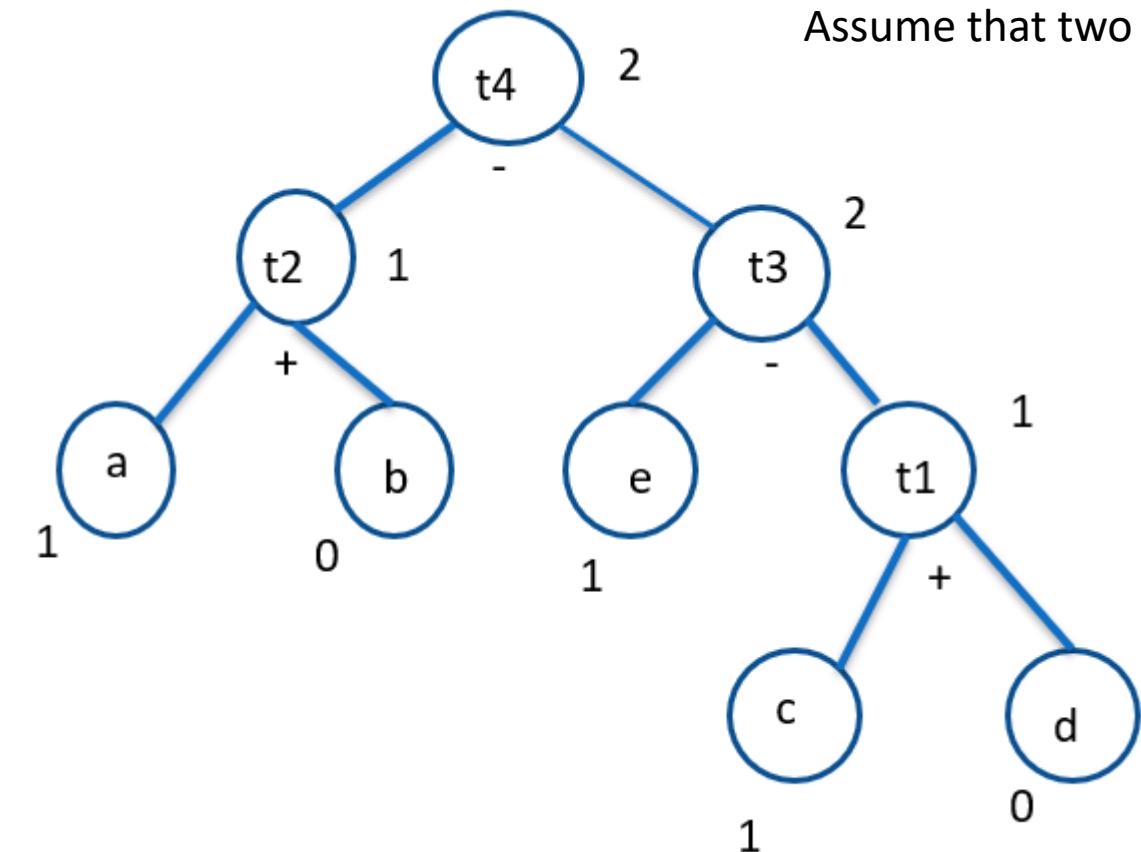
gencode(t4) [R0,R1]
gencode(t3) [R1,R0]
gencode(e) [R1,R0]
MOV e , R0
gencode(t1) [R1]
gencode(c) [R1]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

[R0,R1]

gencode(t3)

[R1,R0]

gencode(e) [R1,R0]

MOV e , R0

gencode(t1) [R1]

gencode(c) [R1]

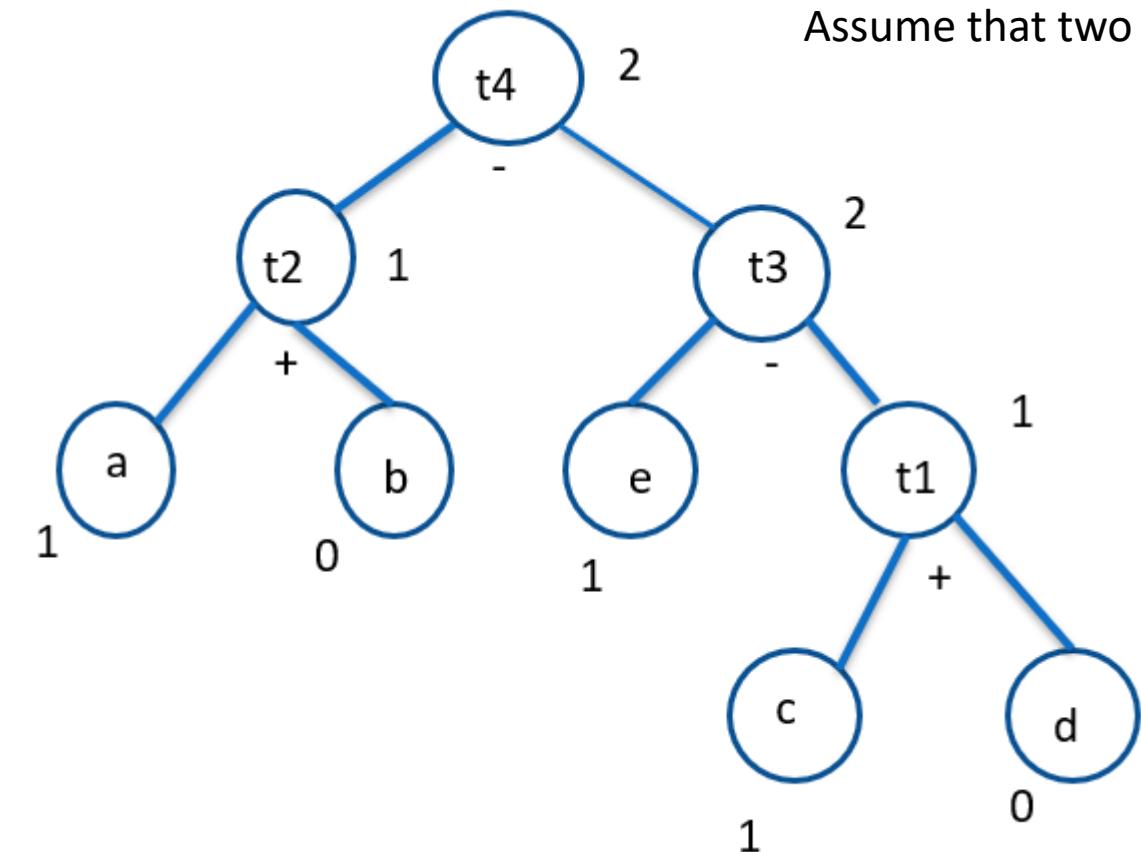
MOV c , R1

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

[R0,R1]

gencode(t3)

[R1,R0]

gencode(e) [R1,R0]
MOV e , R0

gencode(t1) [R1]

gencode(c) [R1]
MOV c , R1

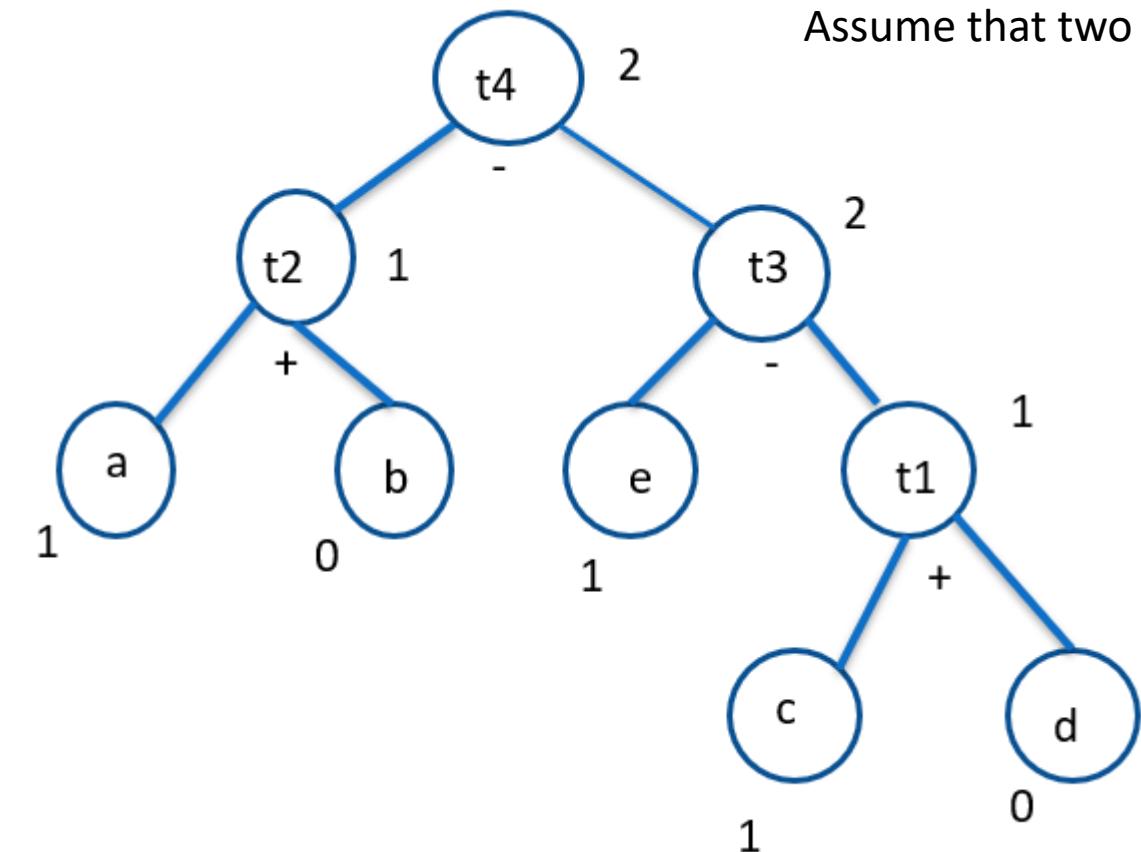
ADD d , R1

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

[R0,R1]

gencode(t3)

[R1,R0]

gencode(e)

[R1,R0]

MOV e , R0

gencode(t1)

[R1]

gencode(c)

[R1]

MOV c , R1

ADD d , R1

SUB R0 , R1

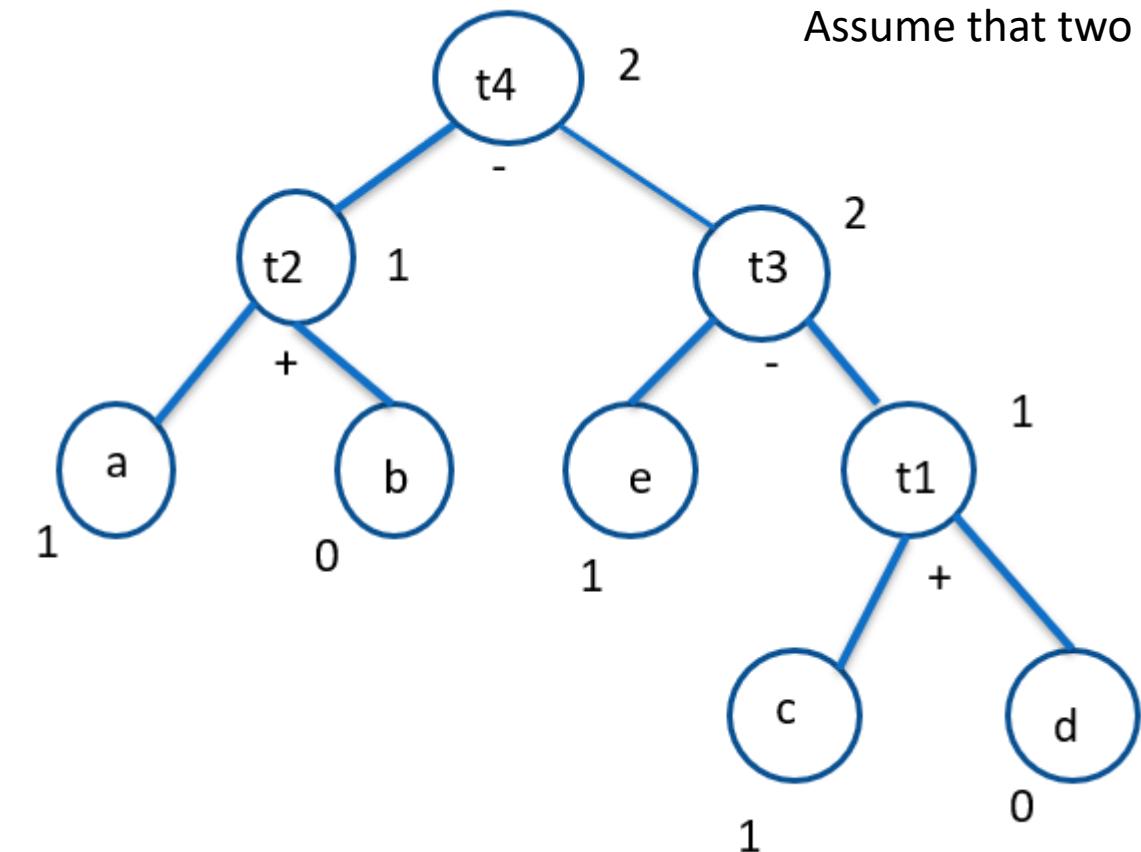
[R0]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)

[R0,R1]

gencode(t3)

[R1,R0]

gencode(e) [R1,R0]

MOV e , R0

gencode(t1) [R1]

gencode(c) [R1]

MOV c , R1

ADD d , R1

SUB R0 , R1
gencode (t2)

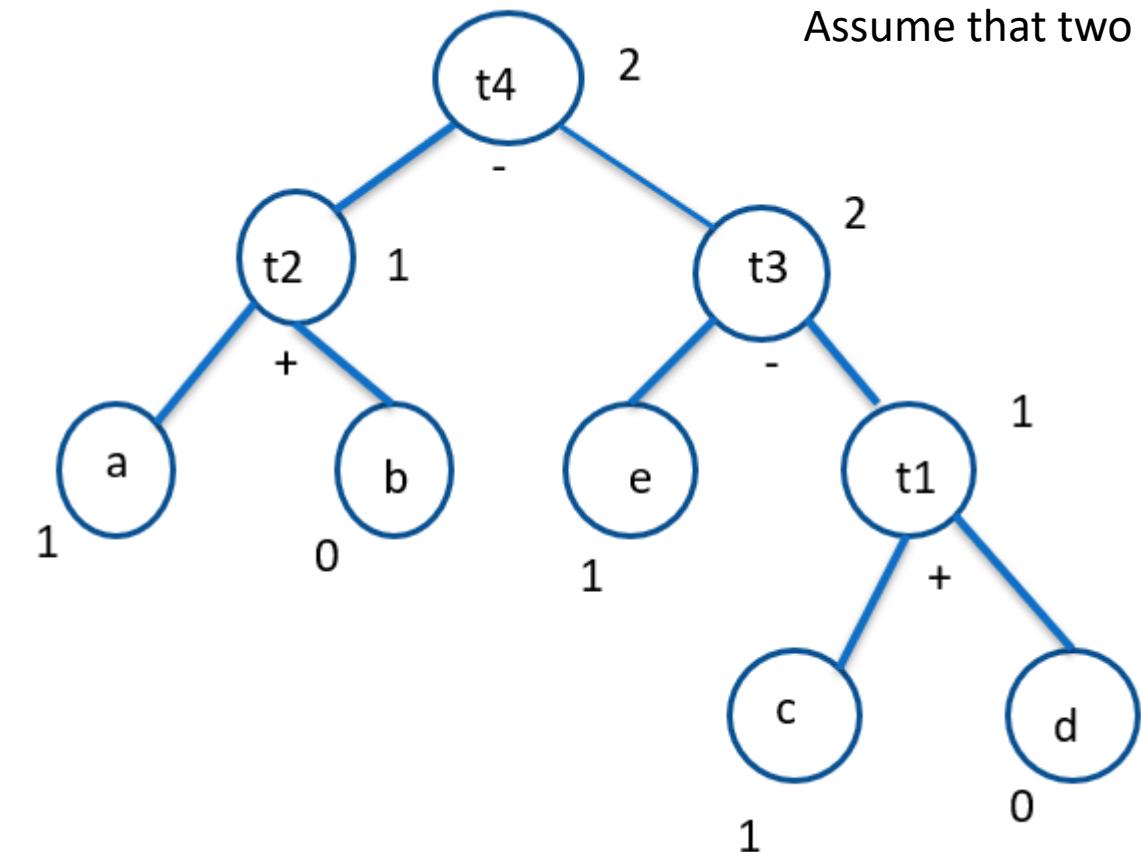
[R0]
[R0]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



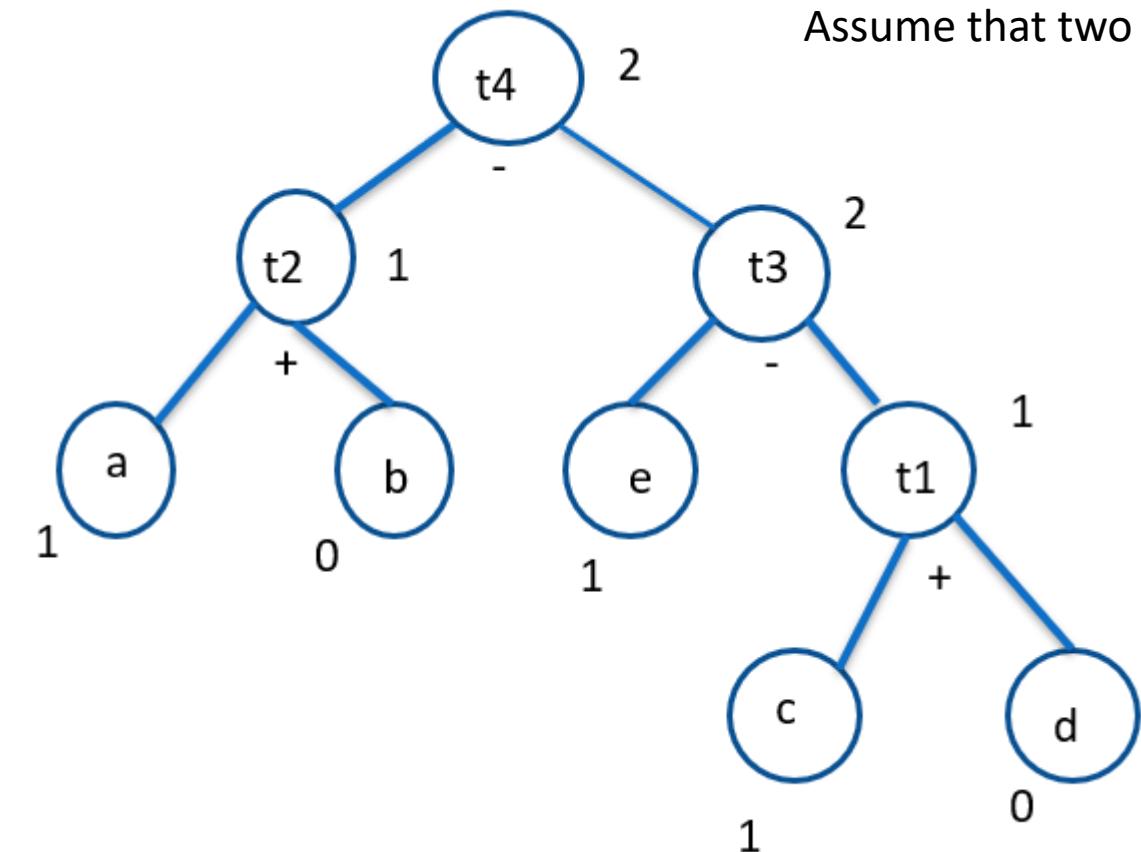
gencode(t4)	[R0,R1]
gencode(t3)	[R1,R0]
gencode(e)	[R1,R0]
MOV e , R0	[R1,R0]
gencode(t1)	[R1]
gencode(c)	[R1]
MOV c , R1	
ADD d , R1	
SUB R0 , R1	[R0]
gencode (t2)	[R0]
gencode (a)	

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



gencode(t4)	[R0,R1]
gencode(t3)	[R1,R0]
gencode(e)	[R1,R0]
MOV e , R0	[R1,R0]
gencode(t1)	[R1]
gencode(c)	[R1]
MOV c , R1	
ADD d , R1	
SUB R0 , R1	
gencode (t2)	
gencode (a)	
MOV a , R0	

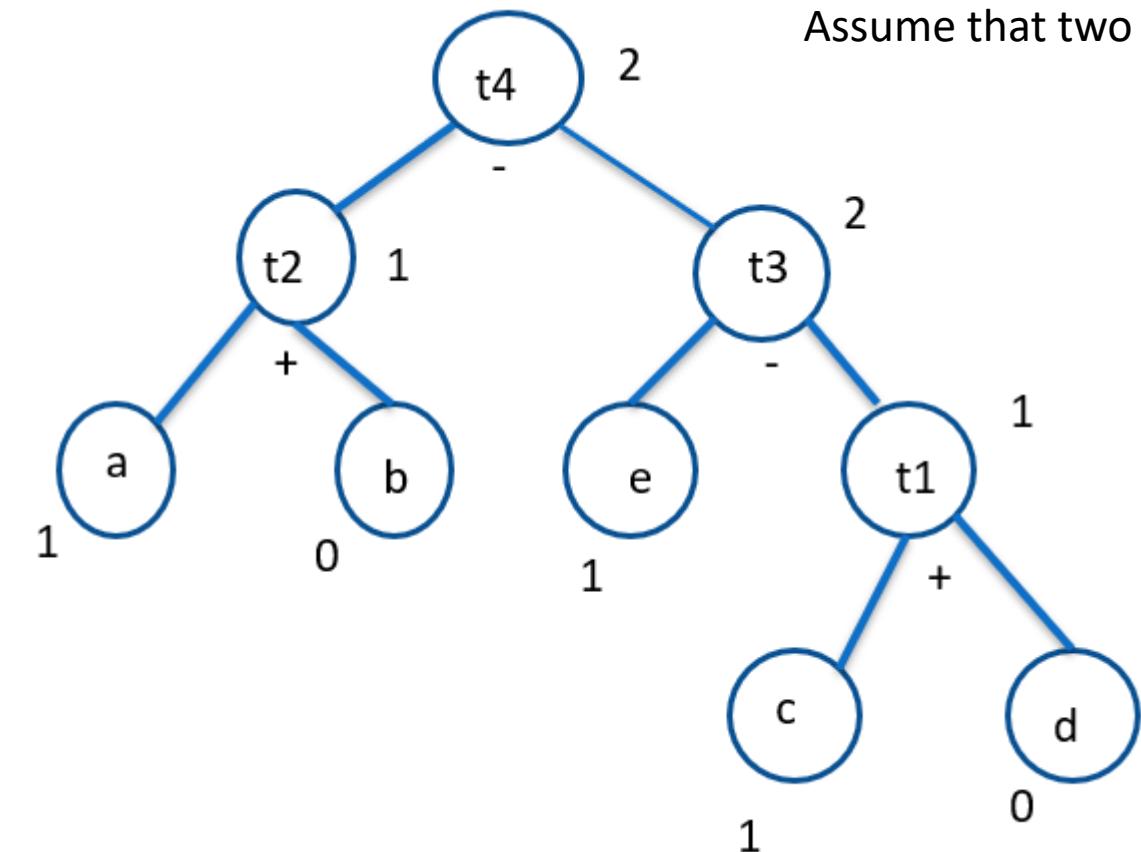
[R0]
[R0]

Labelling algorithm for target code generation



Target code

Assume that two registers R0 and R1 are available.



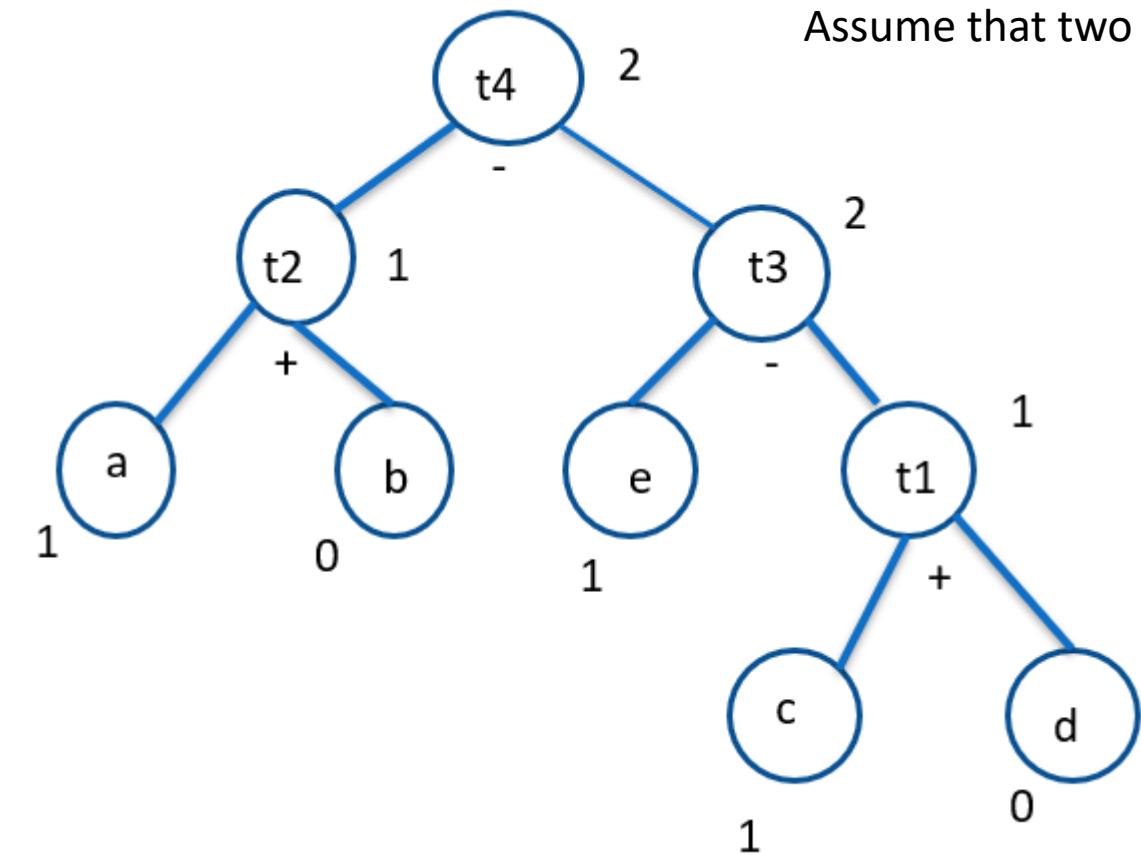
gencode(t4)	[R0,R1]
gencode(t3)	[R1,R0]
gencode(e)	[R1,R0]
MOV e , R0	[R1,R0]
gencode(t1)	[R1]
gencode(c)	[R1]
MOV c , R1	
ADD d , R1	
SUB R0 , R1	
gencode (t2)	
gencode (a)	
MOV a , R0	
ADD b , R0	

[R0]
[R0]

Labelling algorithm for target code generation

Target code

Assume that two registers R0 and R1 are available.



gencode(t4)	[R0,R1]
gencode(t3)	[R1,R0]
gencode(e)	[R1,R0]
MOV e , R0	[R1,R0]
gencode(t1)	[R1]
gencode(c)	[R1]
MOV c , R1	
ADD d , R1	
SUB R0 , R1	
gencode (t2)	
gencode (a)	
MOV a , R0	
ADD b , R0	
SUB R0,R1	
	[R0]
	[R0]

Finally , the generated code :



```
MOV e, R0
MOV c, R1
ADD d, R1
SUB R0, R1
MOV a, R0
ADD b, R0
SUB R0, R1
```

PEEPHOLE OPTIMIZATION



- A statement-by-statement code-generations strategy often produces target code that **contains redundant instructions**. The quality of such target code can be improved by applying “optimizing” transformations to the target program.
- A simple but effective technique for improving the target code is **peephole optimization**, a method for trying to **improving the performance** of the target program by examining a **short sequence of target instructions** (called the **peephole**) and replacing these instructions by a shorter or faster sequence, whenever possible.

Characteristics of peephole optimizations

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable

Redundant Loads And Stores:



If we see the instructions sequence

- (1) MOV R0, a
- (2) MOV a, R0

we can delete instructions (2) because whenever (2) is executed (1) will ensure that the value of a is already in register R0.

Eliminating Multiple Jumps(**flow of control optimization**)



If we have jumps to other jumps, then the unnecessary jumps can be eliminated.

If we have a jump sequence:

`goto L1`

 ...

`L1: goto L2`

then this can be replaced by:

`goto L2`

 ...

`L1: goto L2`

If there are now no jumps to *L1*, then it may be possible to eliminate

Eliminating Unreachable Code



```
#define debug 0  
...  
if (debug)  
{  
    print debugging information  
}
```

The above “if“ statement can be translated in the intermediate code to:

```
If debug = 1 goto L1  
goto L2  
L1: print debugging information  
L2 :
```



Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

$x := x + 0$

or

$x := x * 1$

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Strength Reduction



Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$$X^2 \rightarrow X*X$$

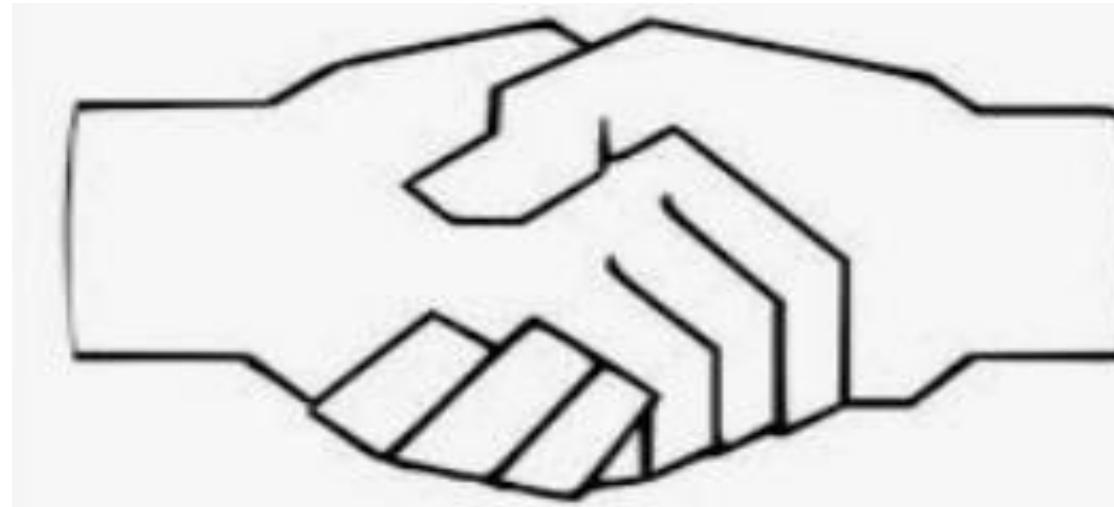


Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

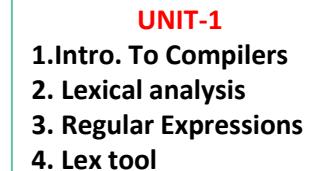
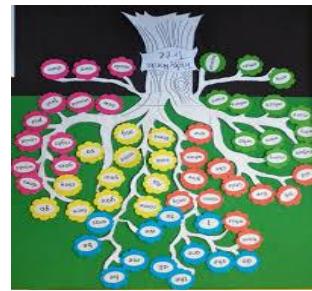
$i := i + 1 \rightarrow i++$
 $i := i - 1 \rightarrow i - -$

END OF UNIT 6



COMPILER DESIGN

WIT and WIL



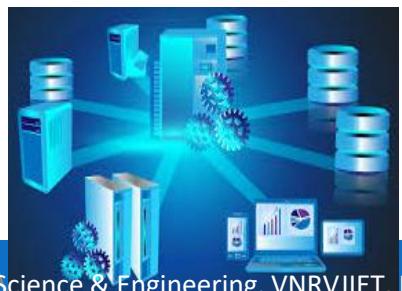
UNIT-2

1. Parser
 2. Top-down & Bottom up parser tree



UNIT-3

1. SDD
 2. Intermediate code
 3. Runtime environment



Practice Questions

- What are the target code representation formats?
- Illustrate the use of graph coloring in register allocation?
- List out the DAG based code generation algorithms.
- Do we really need a compiler?

References

- Compilers principles ,tools and techniques by Aho, Sethi, and Ullman, Chapters 1, 2, 3

<https://parasol.tamu.edu/~rwerger/Courses/434/lec1.pdf>

<https://www.wmlcloud.com/windows/algorithms-for-compiler-design-using-dag-for-code-generation/>

