CODE GENERATION

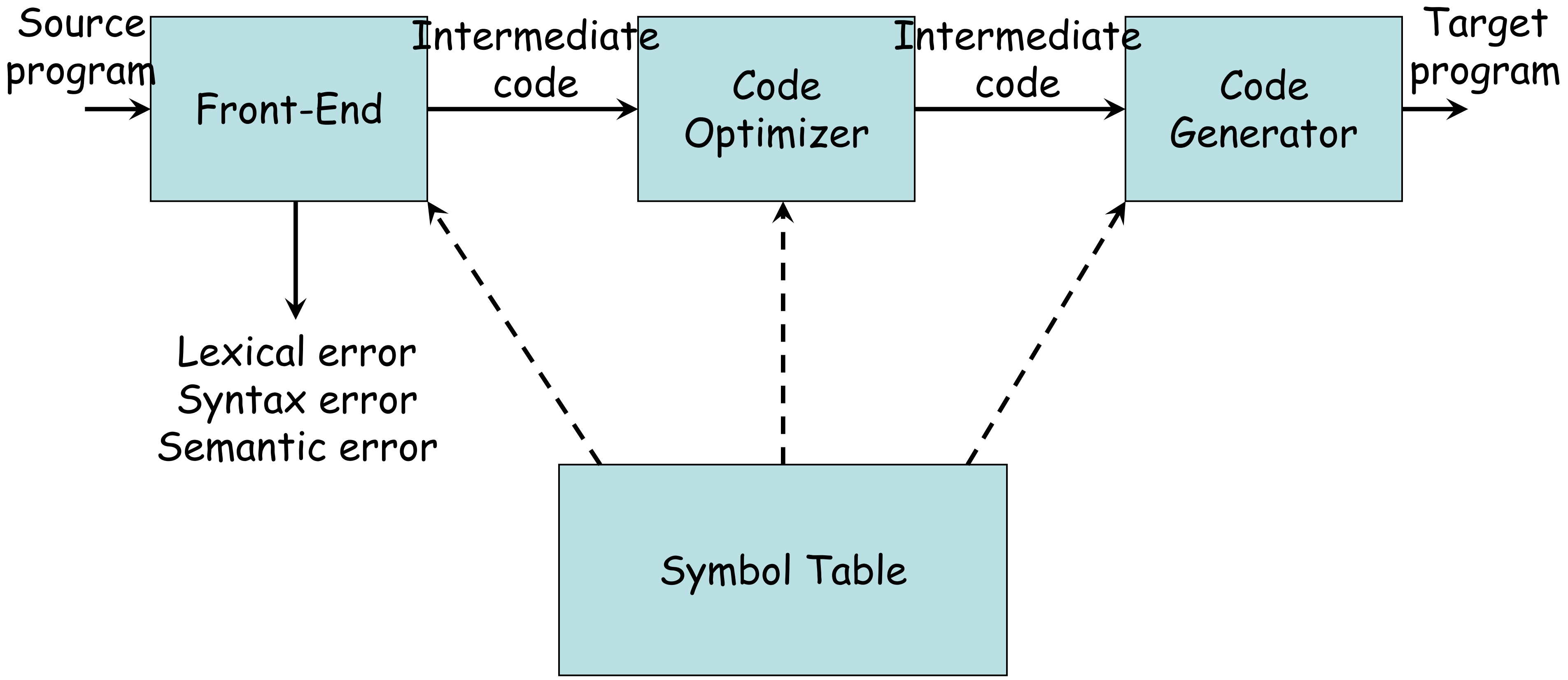
Unit - IV

Design of Code Generator

Overview

- Issues in code generation
- Basic blocks and Flow graph

Position of a Code Generator in the Compiler Model



Code generation techniques can be used whether or not an optimizing phase occurs before code generation.

Code Generation

- Code produced by compiler must be correct
  - Source-to-target program transformation should be *semantics preserving*
- Code produced by compiler should be of high quality
  - Effective use of target machine resources
  - Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is undecidable

ISSUES IN THE DESIGN OF A CODE GENERATOR

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

Input to code generator

- The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
- Intermediate representation can be :
  - Linear representation such as postfix notation
  - Three address representation such as quadruples
  - Virtual machine representation such as stack machine code
  - Graphical representations such as syntax trees and dags.
- 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.

Target program

- The output of the code generator is the target program. The output may be :
- Absolute machine language
  - It can be placed in a fixed memory location and can be executed immediately.
- Relocatable machine language
  - It allows subprograms to be compiled separately.
- Assembly language
  - Code generation is made easier.

Target Program Code

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - Absolute machine code (executable code)
  - Relocatable machine code (object files for linker)
  - Assembly language (facilitates debugging)
  - Byte code forms for interpreters (e.g. JVM)

Memory management

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions.
- Example: **j : goto i** generates jump instruction as follows :
  - if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.
  - if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

Instruction selection

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Hypothetical machine:
  - Byte-addressable (word = 4 bytes)
  - Has *n* general purpose registers **`R0`**, **`R1`**, …, **`R`***n*-1
  - Two-address instructions of the form

op source, *destination*

The Target Machine: Op-codes and Address Modes

- Op-codes (*op*), for example
  - **`MOV`** (move content of *source* to *destination*)
  - **`ADD`** (add content of *source* to *destination*)
  - **`SUB`** (subtract content of *source* from *dest.*)
- Address modes

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*	N/A	1

Instruction Costs

- Define the cost of instruction
  = 1 + cost(*source*-mode) + cost(*destination*-mode)

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(`R0`) into register `R1`	1
`MOV R0,M`	Store *content*(`R0`) into memory location `M`	2
`MOV M,R0`	Store *content*(`M`) into register `R0`	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(`R0`)) into `M`	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(`R0`))) into `M`	3
`MOV #1,R0`	Store 1 into `R0`	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(`R0`)) to *contents*(12+*contents*(`R1`))	3

Instruction Selection

- Instruction selection is important to obtain efficient code
- Suppose we translate three-address code

x:=*y*+*z*

to:

```
MOV y,R0
ADD z,R0
MOV R0,x
```

```
a:=a+1
```

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

Cost = 6

Better

```
ADD #1,a
```

Cost = 3

Best

```
INC a
```

Cost = 2

Instruction Selection

- Suppose we translate three-address code

```
x:=y+z
```

to:

```
MOV y,R0
ADD z,R0
MOV R0,x
```

- Then, we translate

```
a:=b+c
d:=a+e
```

to:

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0   ← Redundant
ADD e,R0
MOV R0,d
```

Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- Efficient utilization of the limited set of registers is important to generate good code
- The use of registers is subdivided into two subproblems :
  - *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
- Finding an optimal register assignment in general is NP-complete

Register allocation

- Certain machine requires even-odd register pairs for some operands and results.
- For example , consider the division instruction of the form :
- D x, y
  - x – dividend even register in even/odd register pair
  - y – divisor
  - even register holds the remainder
  - odd register holds the quotient

Register allocation - Example

```
t:=a*b
t:=t+a
t:=t/d
```

↓ { R1=t }

```
MOV a,R1
MUL b,R1
ADD a,R1
DIV d,R1
MOV R1,t
```

```
t:=a*b
t:=t+a
t:=t/d
```

↓ { R0=a, R1=t }

```
MOV a,R0
MOV R0,R1
MUL b,R1
ADD R0,R1
DIV d,R1
MOV R1,t
```

Choice of Evaluation Order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.
- When instructions are independent, their evaluation order can be changed

```
a+b-(c+d)*e
```

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

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

reorder

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

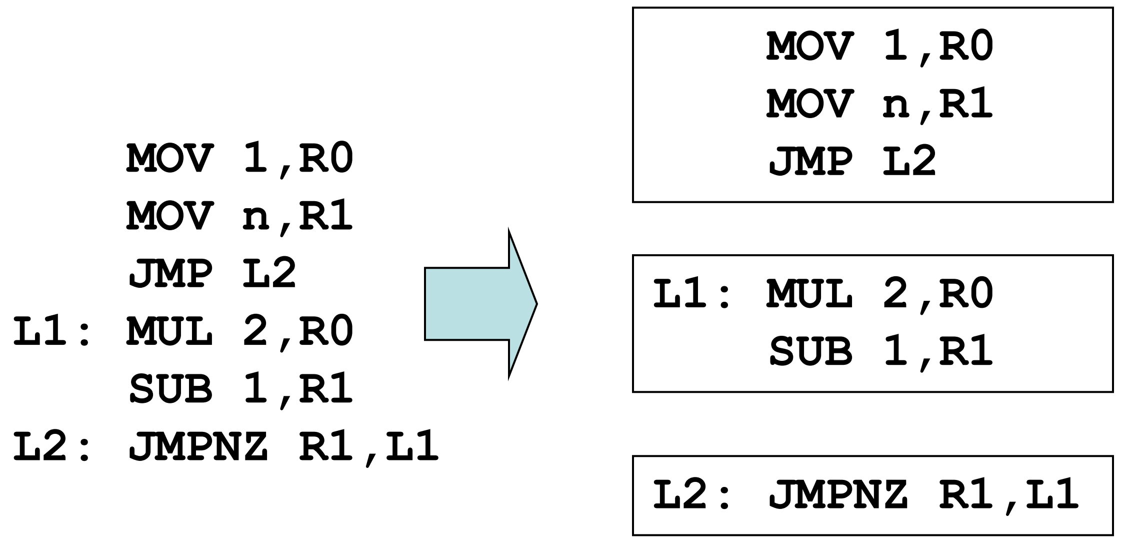
```
MOV c,R0
ADD d,R0
MOV e,R1
MUL R0,R1
MOV a,R0
ADD b,R0
SUB R1,R0
MOV R0,t4
```

Basic Blocks

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- Example:

```
t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5
```

Basic Blocks



```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Partition Algorithm for Basic Blocks

- Input: A sequence of three-address statements
- Output: A list of basic blocks with each three-address statement in exactly one block
- Algorithm:
  1. Determine the set of leaders, the first statements of basic blocks
     - The first statement is the leader
     - Any statement that is the target of a goto is a leader
     - Any statement that immediately follows a goto is a leader
  2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Basic Blocks

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin
prod :=0;
i:=1;
do begin
prod :=prod+ a[i] * b[i];
i :=i+1;
end
while i <= 20
end
```

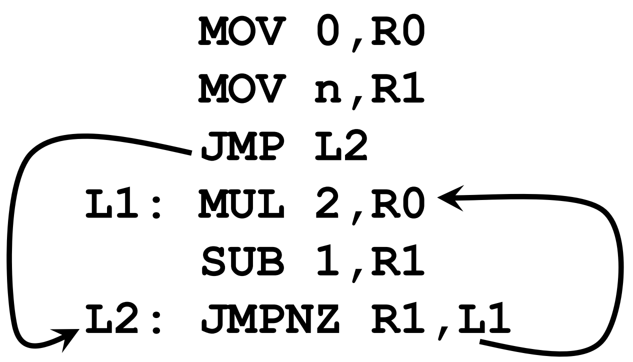
(1) prod := 0
(2) i := 1
(3) t_1 := 4* i
(4) t_2 := a[t_1] /*compute a[i] */
(5) t_3 := 4* i
(6) t_4 := b[t_3] /*compute b[i] */
(7) t_5 := t_2*t_4
(8) t_6 := prod+t_5
(9) prod := t_6
(10) t_7 := i+1
(11) i := t_7
(12) if i<=20 goto (3)

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

```
     MOV   1,R0
     MOV   n,R1
     JMP   L2
L1:  MUL   2,R0
     SUB   1,R1
L2:  JMPNZ R1,L1
```

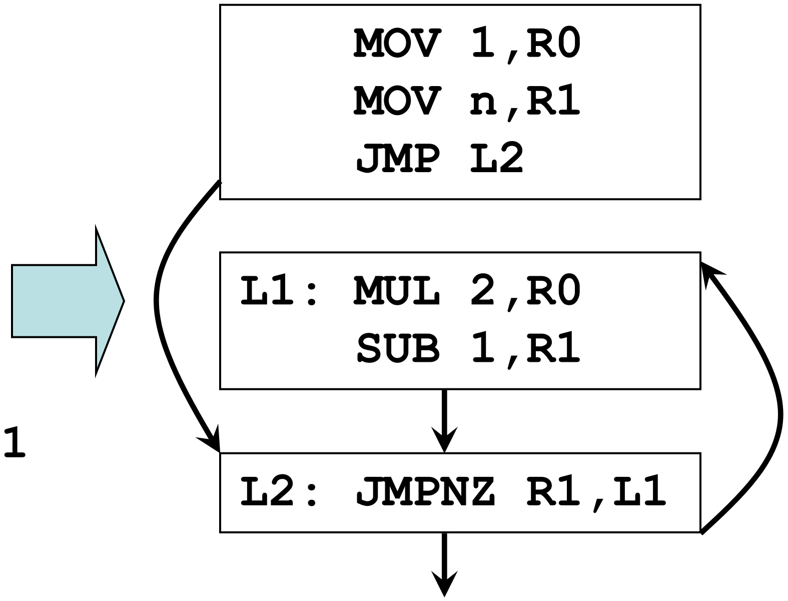


```
     MOV   0,R0
     MOV   n,R1
     JMP   L2
L1:  MUL   2,R0
     SUB   1,R1
L2:  JMPNZ R1,L1
```

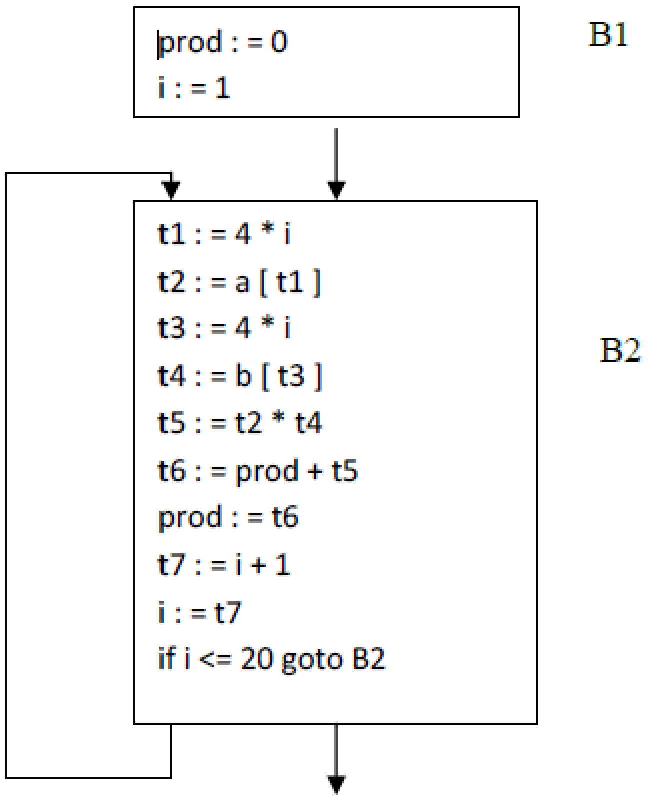
Basic Blocks and Control Flow Graphs

- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```

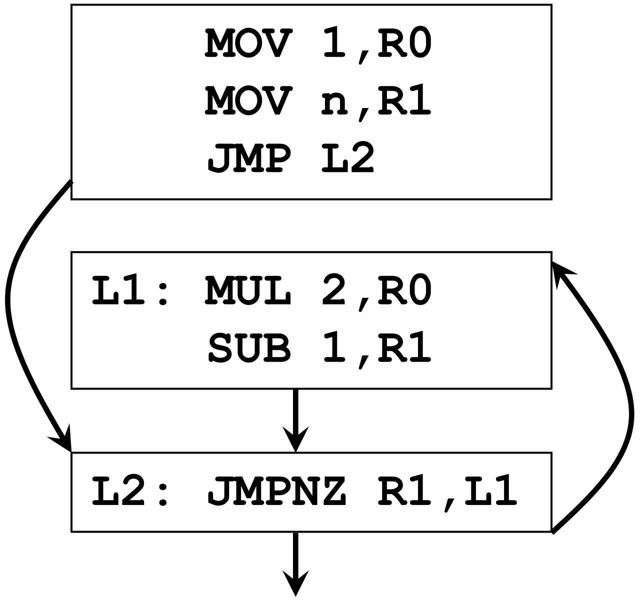


Flow Graphs


prod : = 0
i : = 1
B1
t1 : = 4 * i
t2 : = a [t1]
t3 : = 4 * i
t4 : = b [t3]
t5 : = t2 * t4
t6 : = prod + t5
prod : = t6
t7 : = i + 1
i : = t7
if i <= 20 goto B2
B2

Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



```
      MOV 1,R0
      MOV n,R1
      JMP L2
```

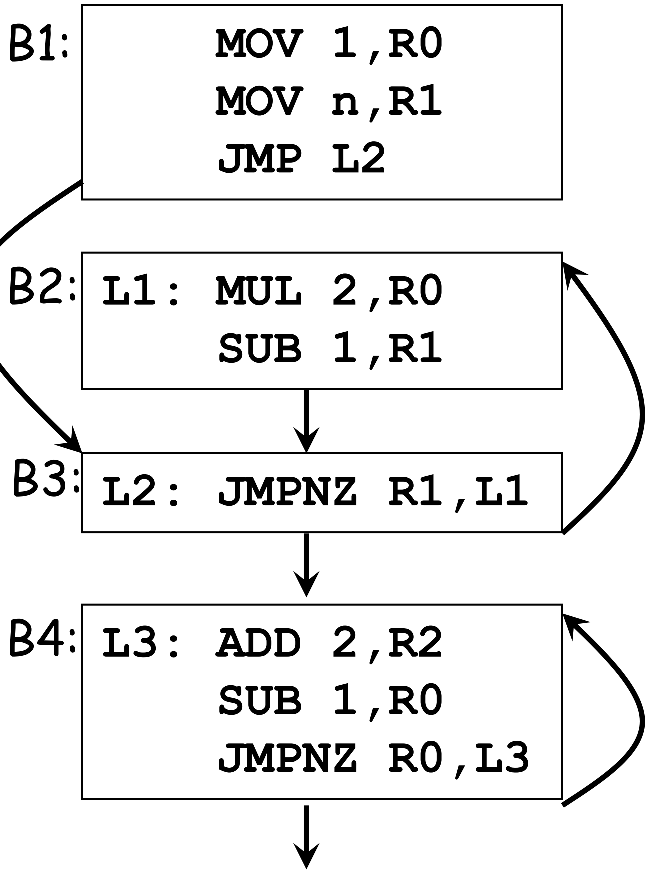
```
L1:   MUL 2,R0
      SUB 1,R1
```

```
L2:   JMPNZ R1,L1
```

Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry
- A loop that contains no other loops is called an inner loop.

Loops (Example)



```
B1:       MOV 1,R0
          MOV n,R1
          JMP L2
B2:  L1:  MUL 2,R0
          SUB 1,R1
B3:  L2:  JMPNZ R1,L1
B4:  L3:  ADD 2,R2
          SUB 1,R0
          JMPNZ R0,L3
```

Strongly connected components:

SCC={ {B2,B3},
{B4} }

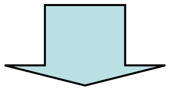
Entries:
B3, B4

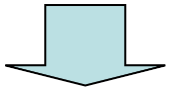
Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b   := 0
t1  := a + b
t2  := c * t1
a   := t2
```

```
a   := c * a
b   := 0
```





```
a  := c*a
b  := 0
```

```
a  := c*a
b  := 0
```

Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

References

- John E Hopcroft and Jeffery D Ullman, Introduction to Automata Theory, Languages and Computations, Narosa Publishing House, 2002.
- Michael Sipser, "Introduction of the Theory of Computation", Second Edition, Thomson Brokecole, 2006.
- J. Martin, "Introduction to Languages and the Theory of Computation", Third Edition, Tata McGraw Hill, 2003.