Code Generation

Course : 2CS701/IT794 – Compiler Construction

Prof Monika Shah

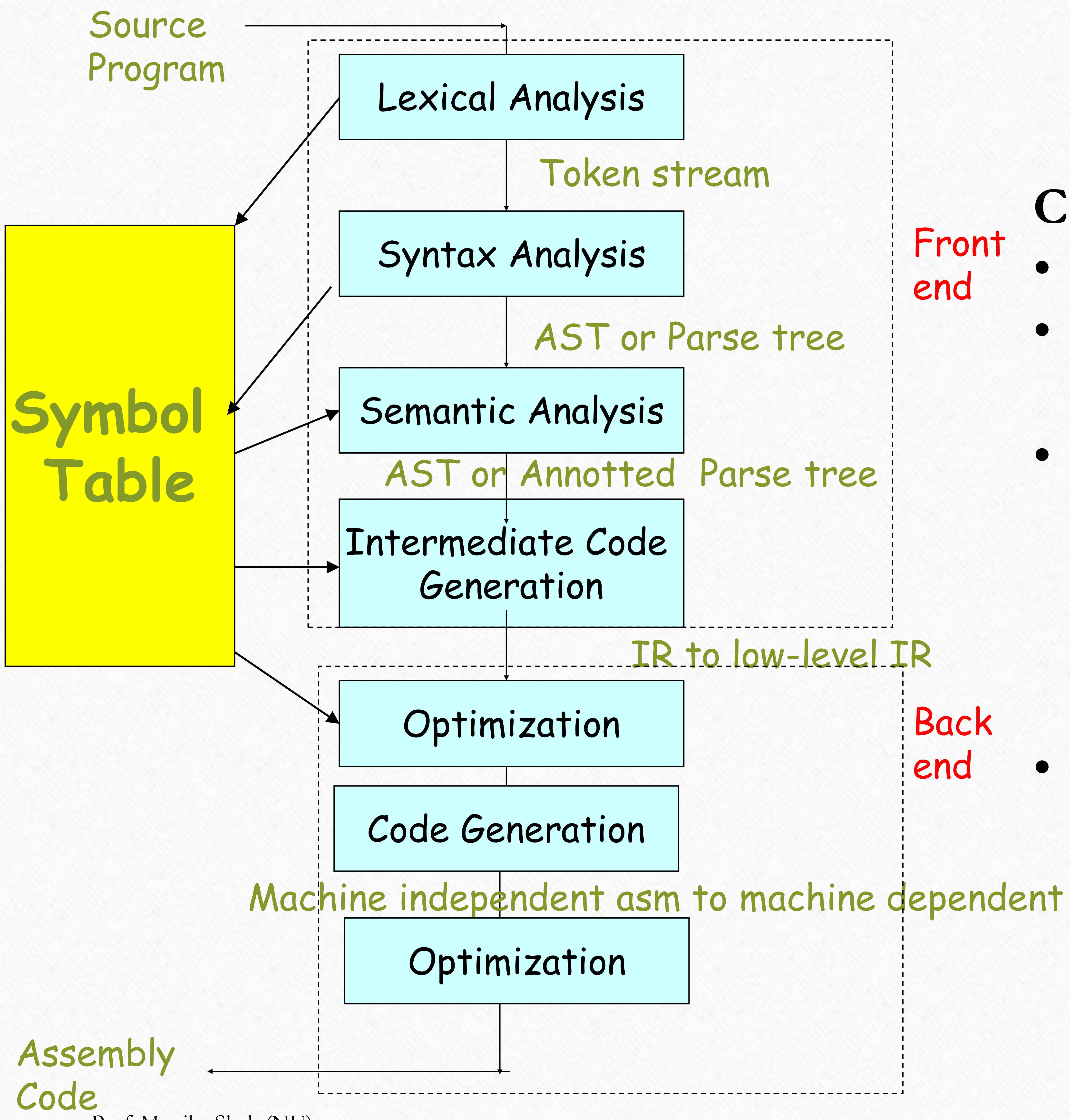
Nirma University

Ref : Ch.9 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

Glimpse

Part-1

- Introduction
- Code Generation Issues



Code Generation

- Final Phase of Compilation
- Produces a semantically equivalent target program
- Main Functionalities :
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

- **Functionalities for QoS**
  - Produce Correct code (semantic preserving)
  - Efficiency
    - Effective use of resources
    - User Heuristics to generate good, but suboptimal code

Major functionalities of Code Generation

- **Instruction selection**

Choose semantically equivalent target machine instructions for the IR statement

Choose cost effective instructions to implement the IR statements

- **Register allocation and assignment**
  - **Register allocation:** selecting the set of variables that will reside in registers at each point in the program
  - **Resister assignment:** selecting specific register that a variable reside in
- **Instruction ordering**

Decide schedule of Instruction (Challenging for Parallel)

Issues in design of Code Generation phase

- Input to Code Generator
- Target Program
- Memory Management
- Instruction Selection
- Register Allocation
- Schedule order of Instructions
- Approaches

Issues in Code Generation design

Input to Code Generation

- Input : IR + Symbol
- IR format choices
  - AST
  - Postfix Notation
  - Three Address Code
- Assumption :
  - Front end phase generate low-level IR
  - Syntactic and semantic errors detected by earlier phases

Issues in Code Generation design

Target Program (Output)

- The back-end code generator of a compiler may generate different forms of code, depending on the requirements:
  - **Absolute machine code (executable code)**

+ve : Can be placed in fixed location in memory. ➔ immediate executed

  - **Relocatable machine code (object files for linker)**

+ve : allows subprograms to be compiled separately

  - **Assembly language**

+ve : facilitates debugging

+ve: Easy Code Generation using Assembler

  - **Byte code forms for interpreters (e.g. JVM)**

+ve : Platform independence

Issues in Code Generation design

Target Program (Output)

The Target Machine

- Implementing code generation requires thorough understanding of the target machine architecture and its instruction set
- Our (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

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction
  $= 1 + \text{cost}(\textit{source}\text{-mode}) + \text{cost}(\textit{destination}\text{-mode})$

Examples

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 value at location *contents*(12+*contents*(`R1`))	3

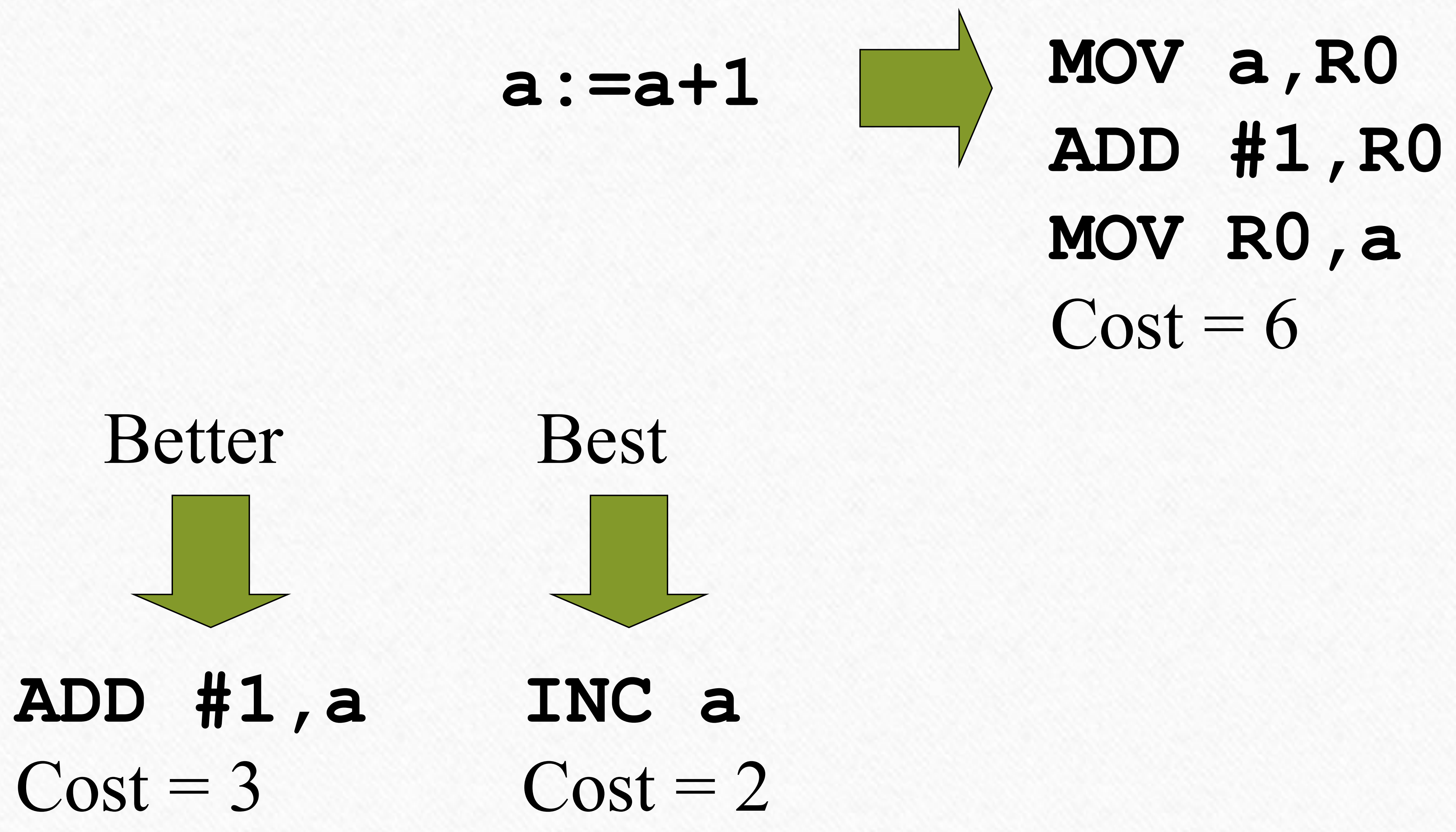
Issues in Code Generation design

Target Program (Output)

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

Issues in Code Generation design

Instruction Selection: Utilizing Addressing Modes

- Suppose we translate **`a:=b+c`** into

```
MOV b,R0
ADD c,R0
MOV R0,a
```

- Assuming addresses of **`a`**, **`b`**, and **`c`** are stored in **`R0`**, **`R1`**, and **`R2`**

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

- Assuming **`R1`** and **`R2`** contain values of **`b`** and **`c`**

```
ADD R2,R1
MOV R1,a
```

Issues in Code Generation design

Need for Global Machine-Specific Code Optimizations

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

Issues in Code Generation design

Register Allocation and Assignment

- 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
- Finding an optimal register assignment in general is NP-complete

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
```

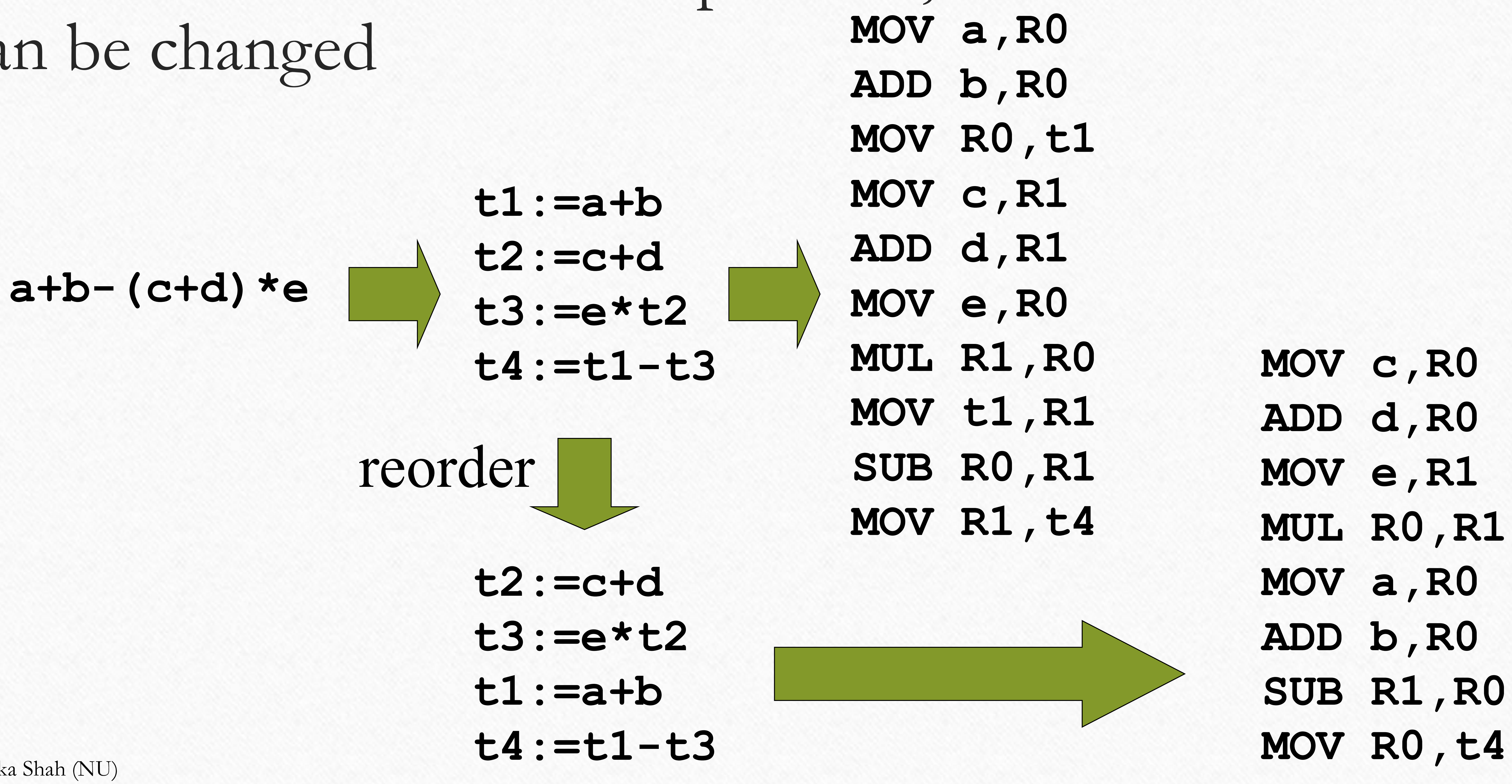
↓ { R0=a, R1=t }

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

Issues in Code Generation design

Choice of Evaluation Order

- 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
```

Issues in Code Generation design

Memory Mapping

- Mapping names in source program to memory address is done by Code Generator phase
- Labels in three address code need to be converted to addresses of Instructions
  - Back patching technique