© Manuel Cargaleiro

Code Generation

Compilers course

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Email: jmpc@fe.up.pt

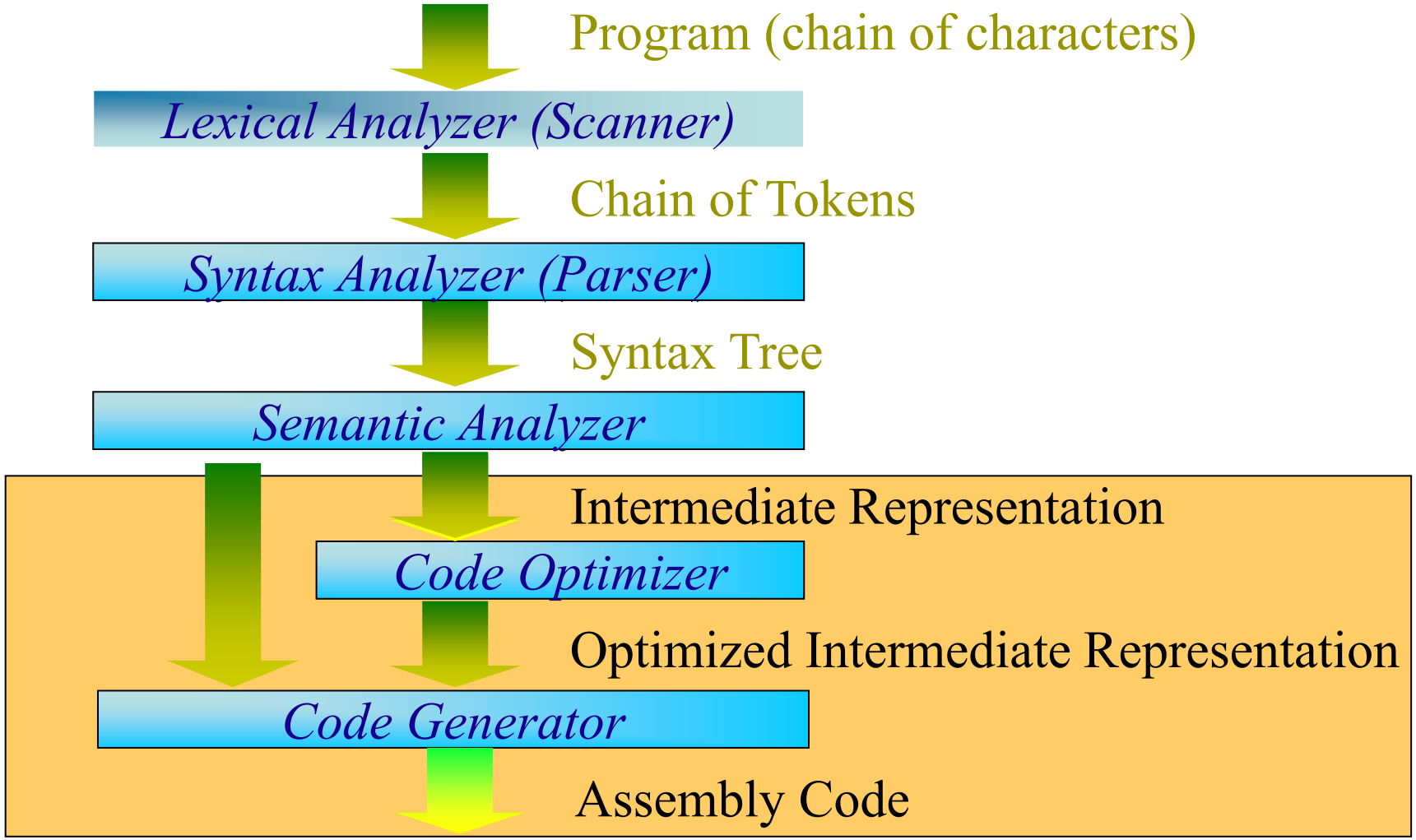
U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Problem

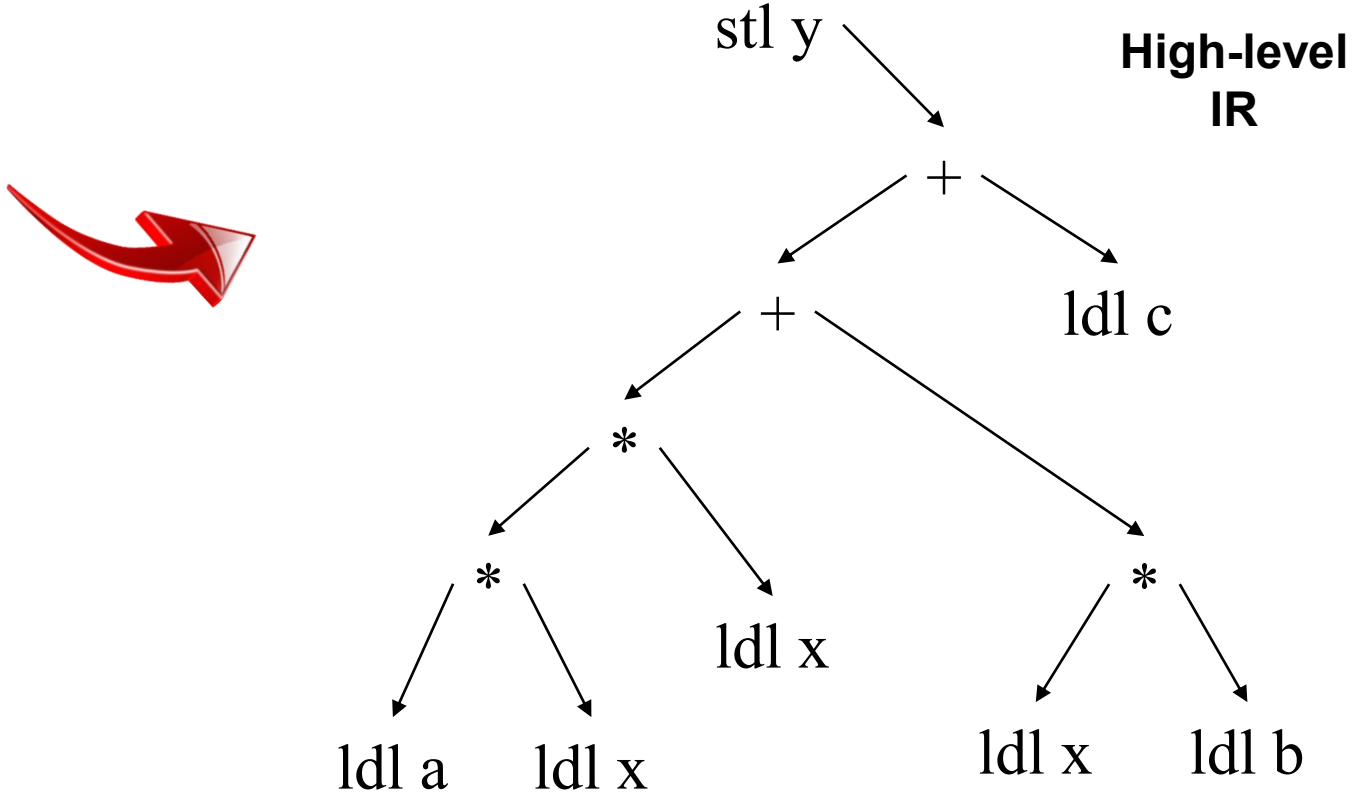
- How to generate assembly code given a low-level intermediate representation?
- Not optimized:
  - Local variables and function parameters all assigned to distinct stack positions
- Optimized:
  - Sharing of relative stack positions by two or more local variables
  - Utilization of registers from the register file of the target microprocessor to accommodate local variables
  - ...

Final Code Generation



Code Generation

y=a*x*x+b*x+c;

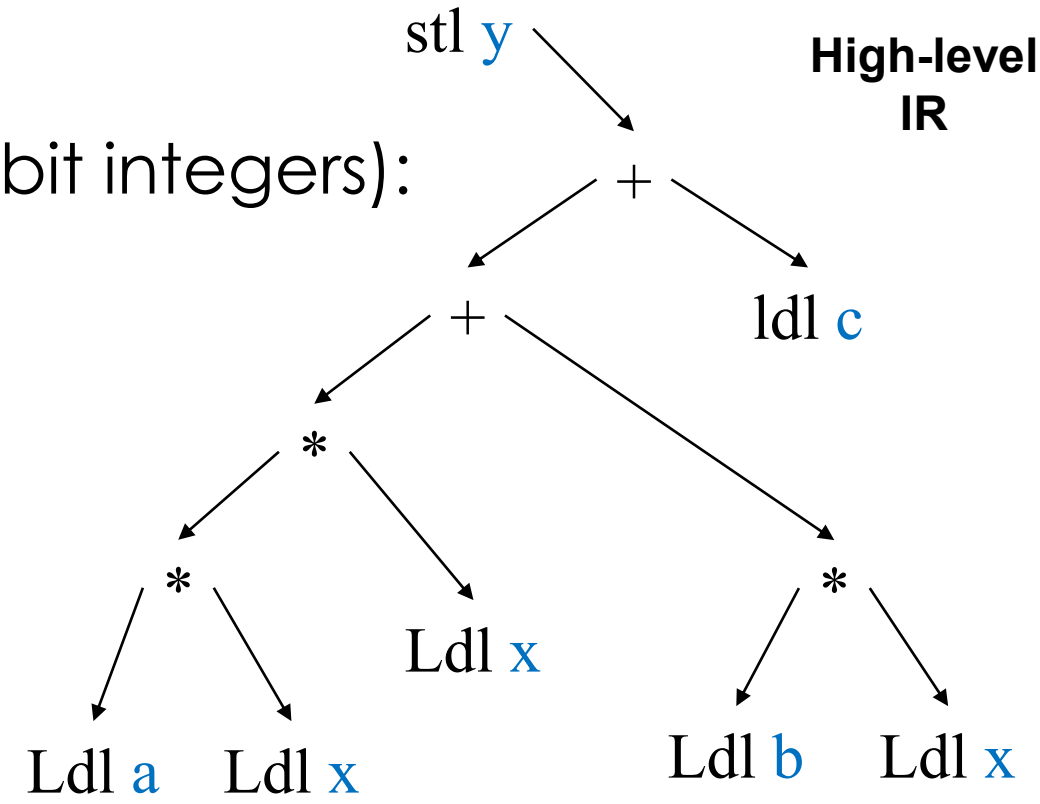


Code Generation

y=a*x*x+b*x+c;

Variables (assume 32-bit integers):

- a
- b
- x
- c
- y

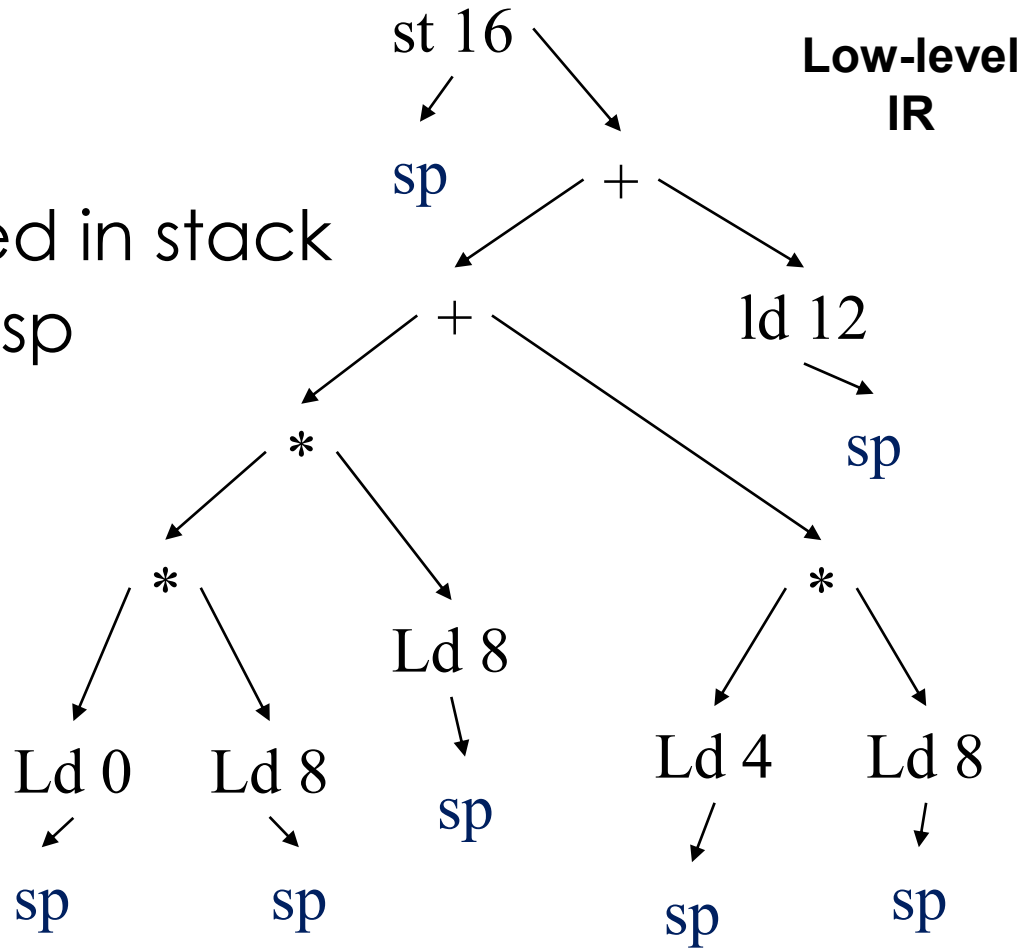


Code Generation

y=a*x*x+b*x+c;

Variables:

- let's assume all stored in stack
- relative position to $sp
  - a: 0
  - b: 4
  - x: 8
  - c: 12
  - y: 16



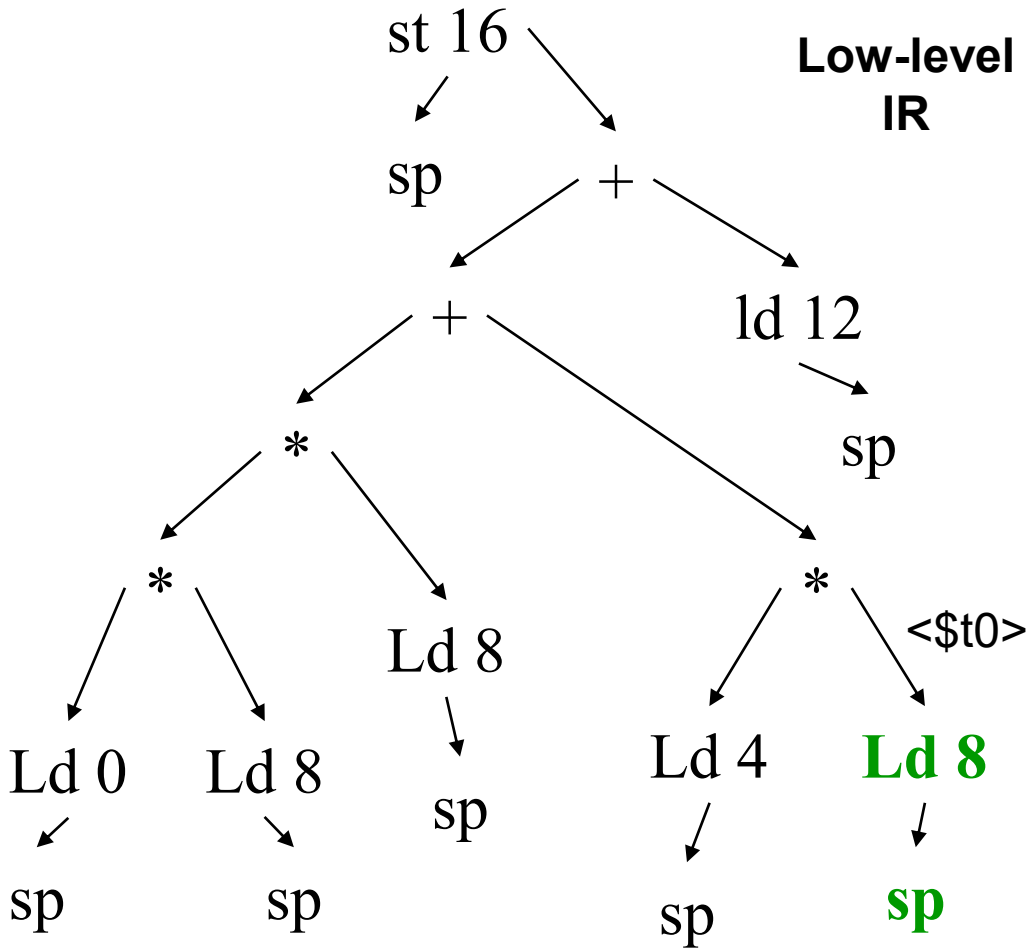
Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

Begin by leaves:

lw $t0, 8($sp)



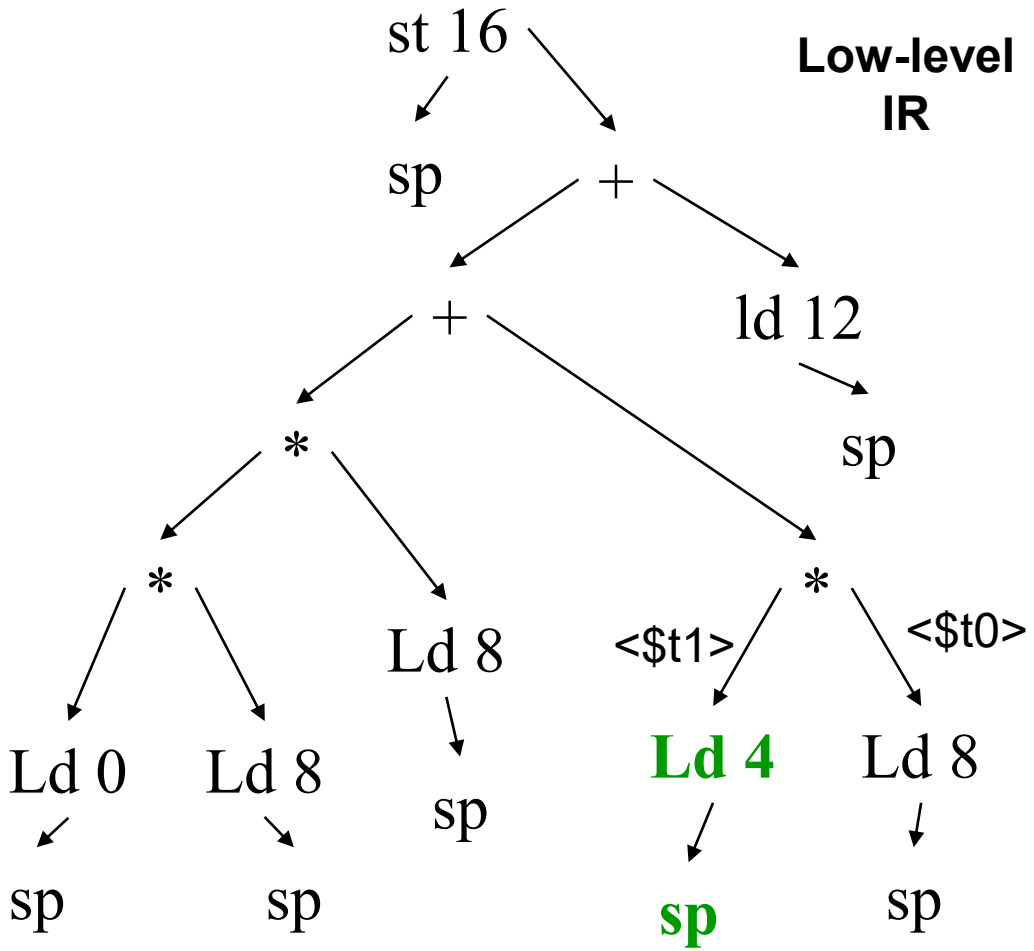
Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)

lw $t1, 4($sp)

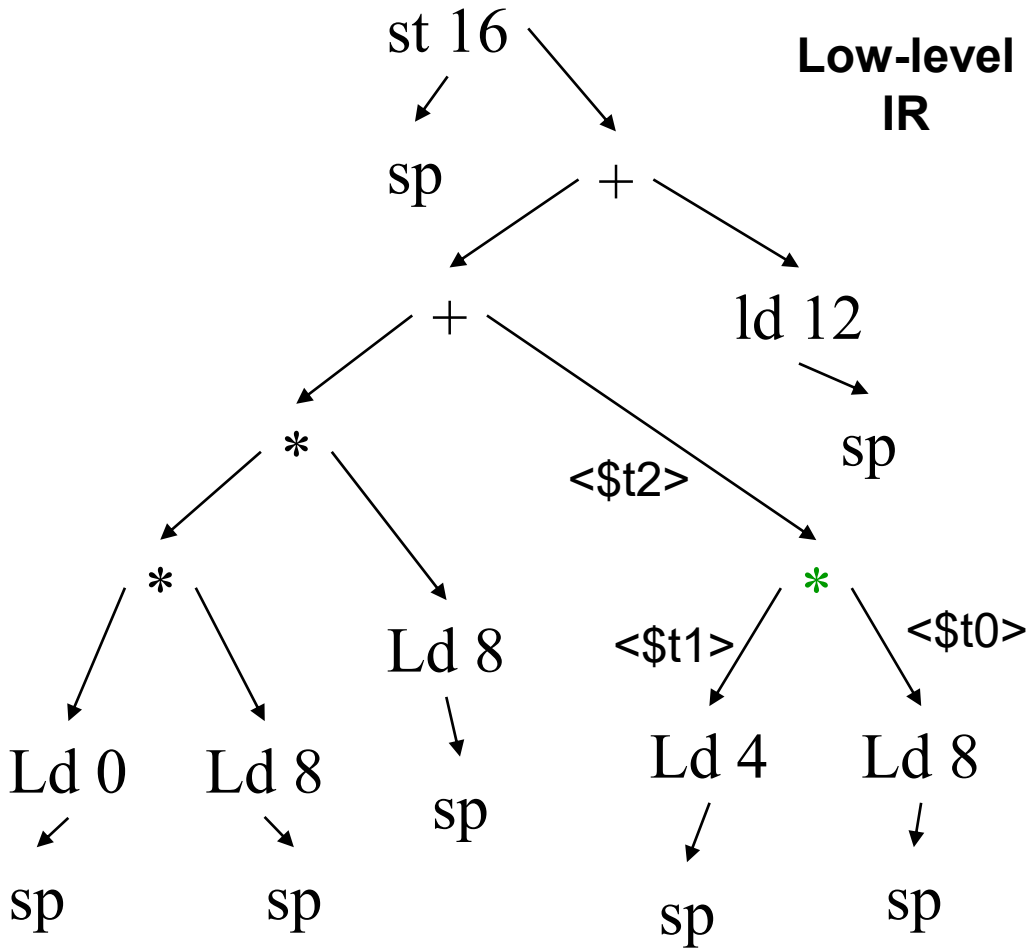


Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0



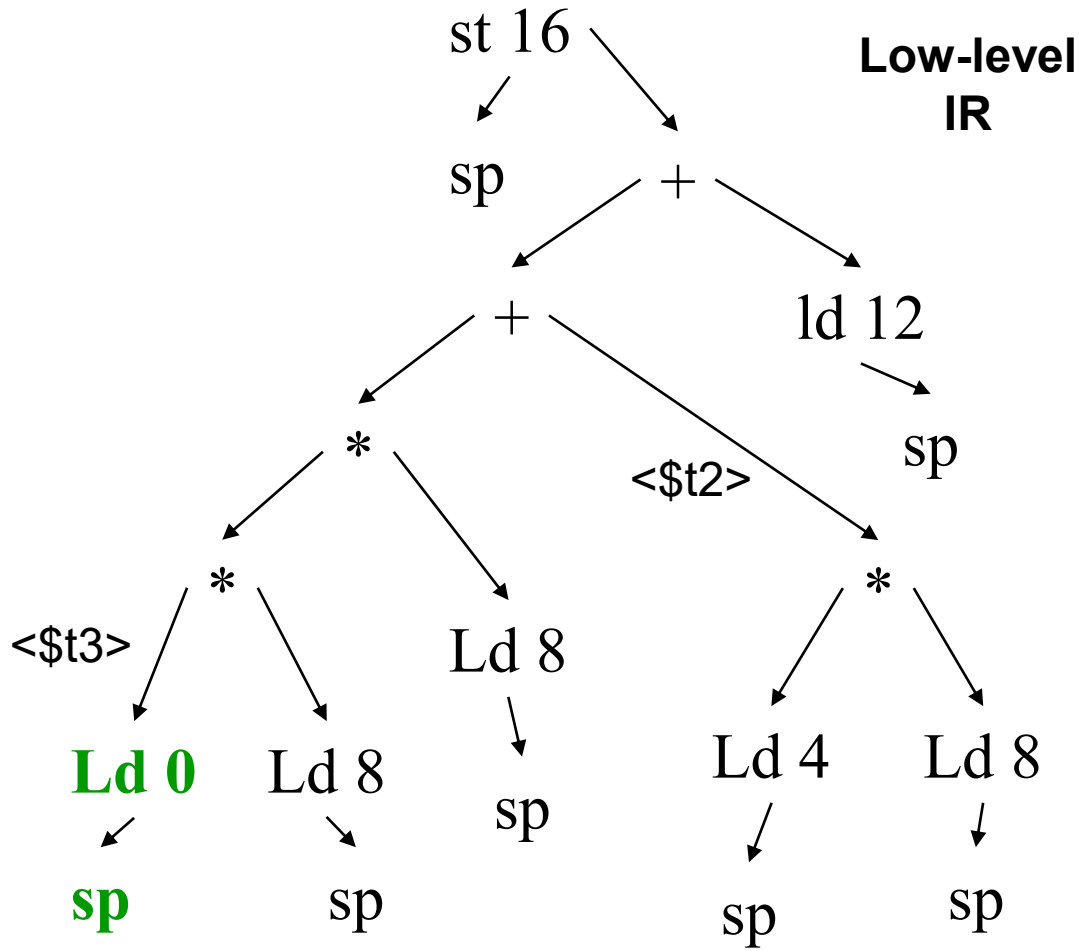
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)



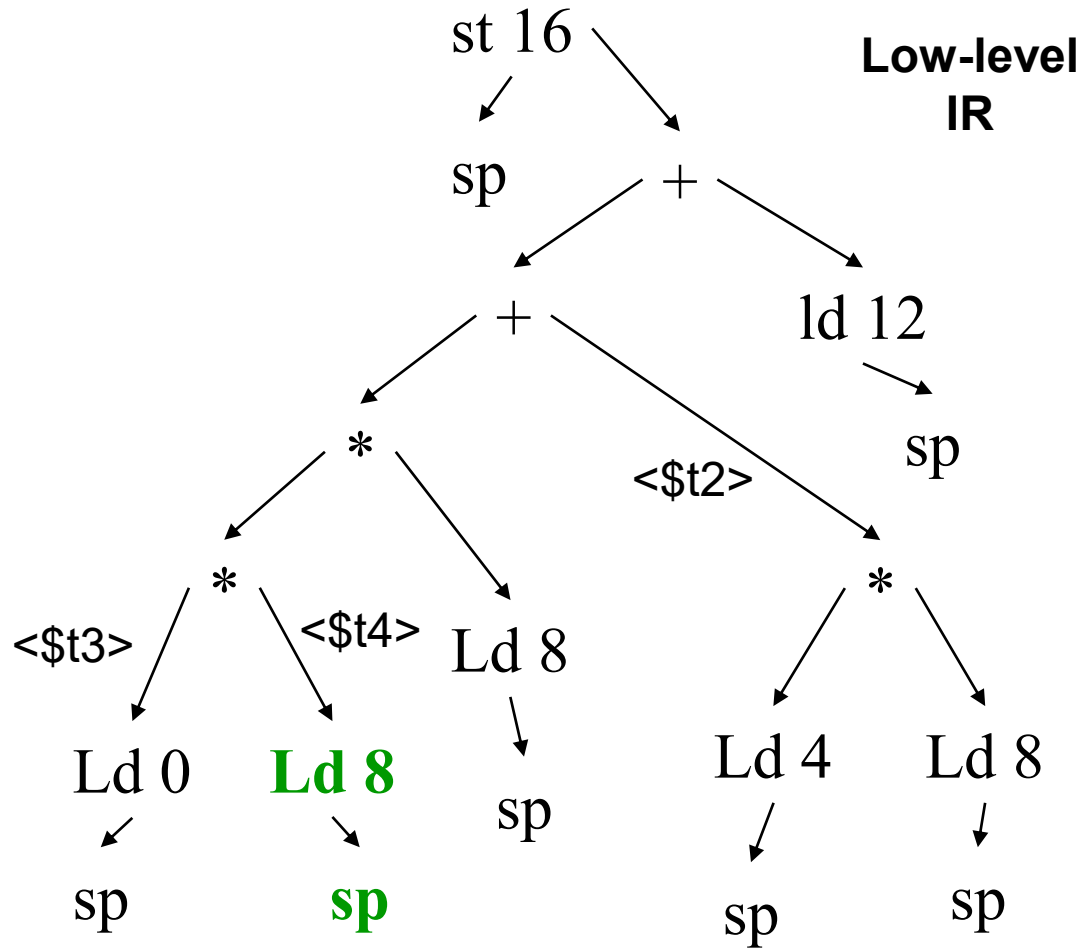
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)



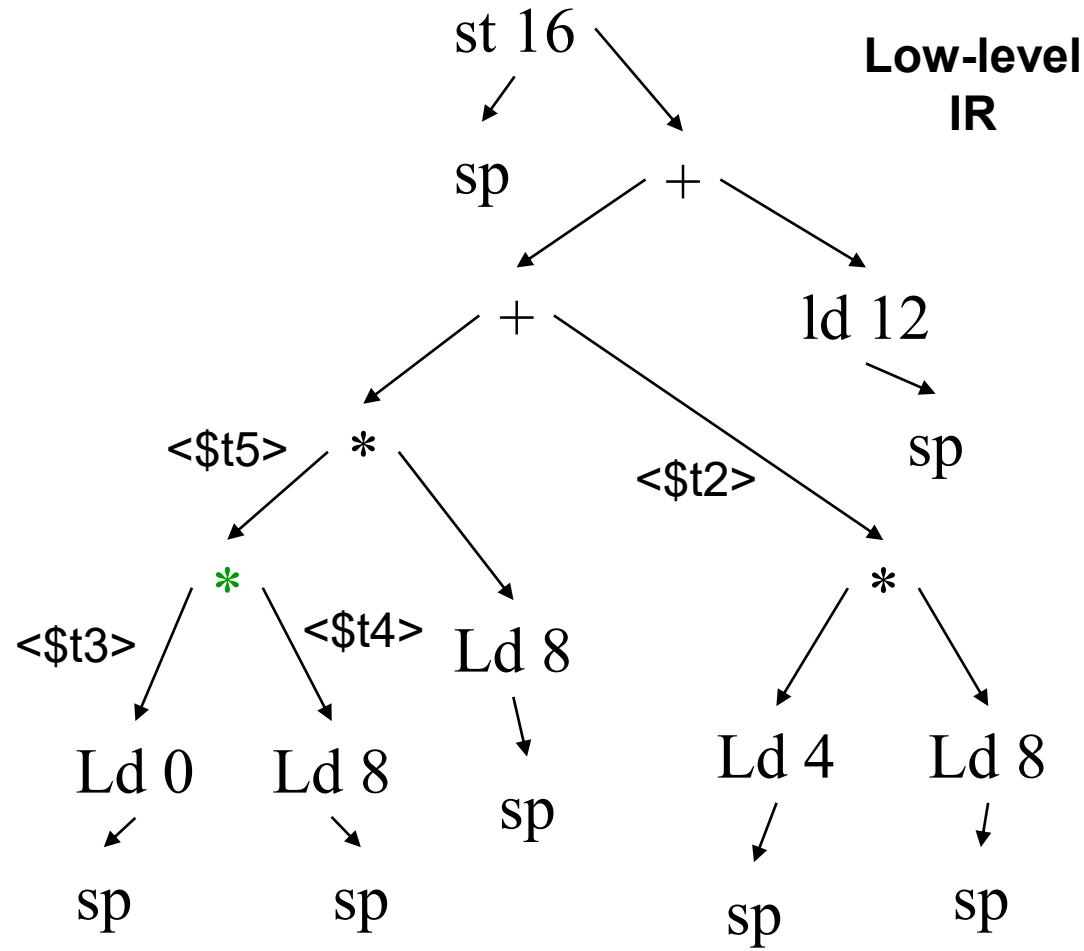
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4



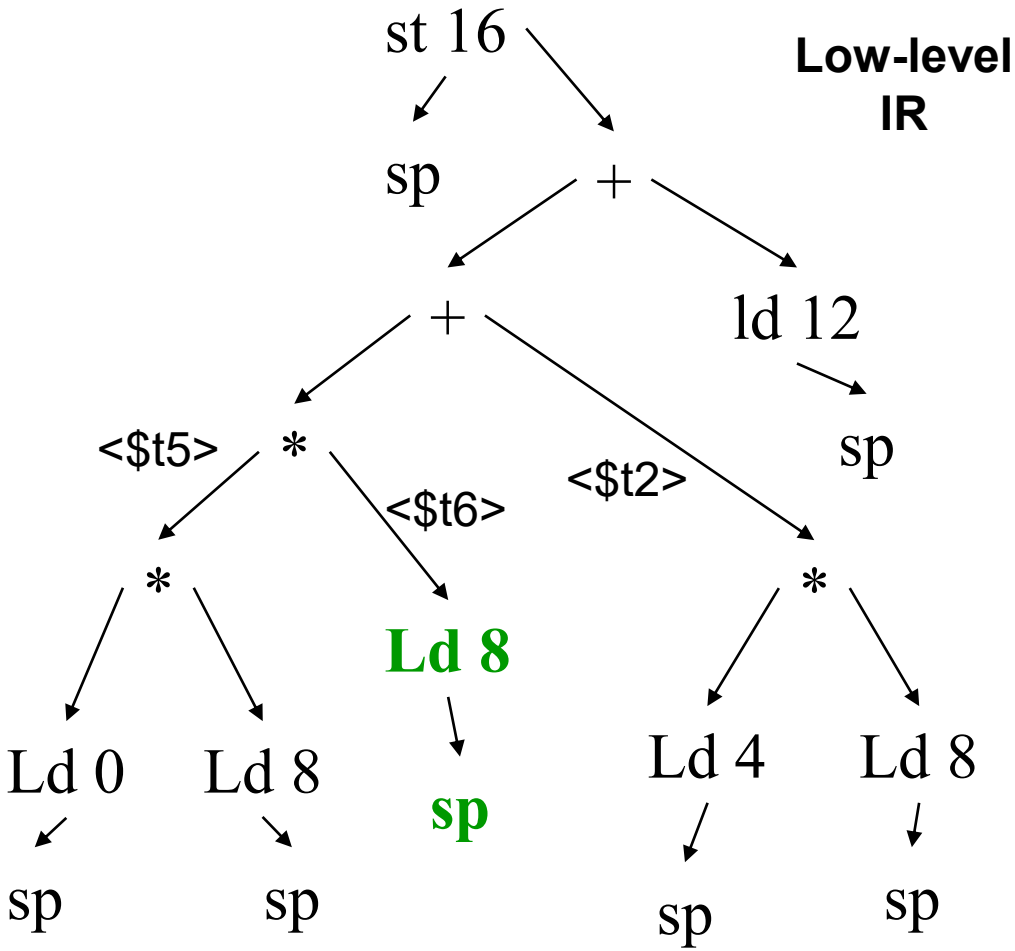
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)

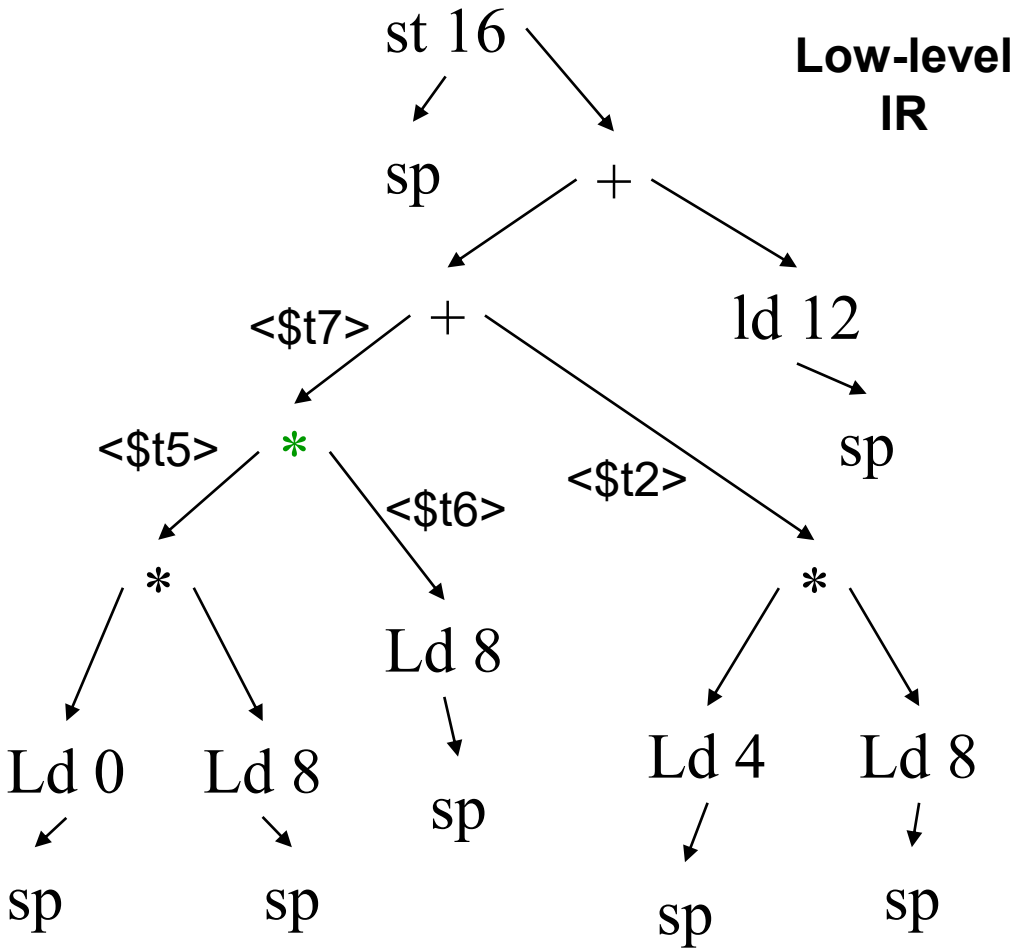


Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
```

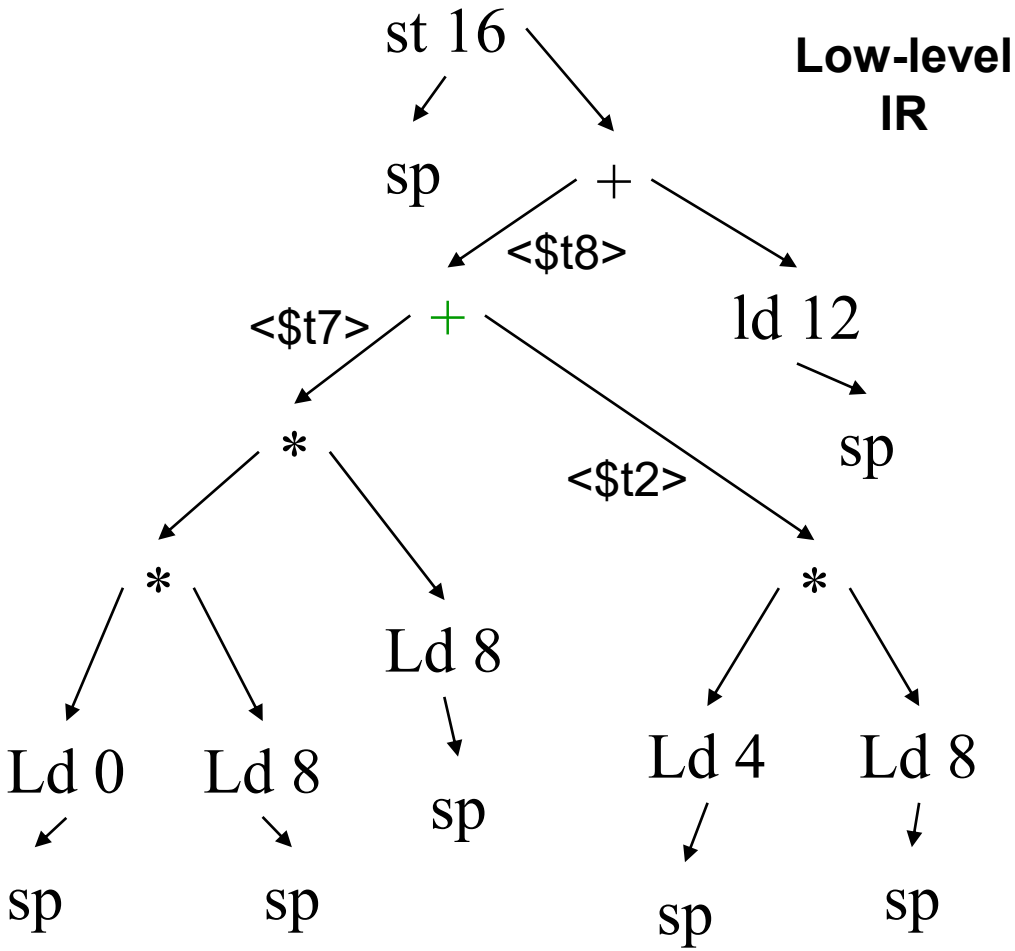


Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2



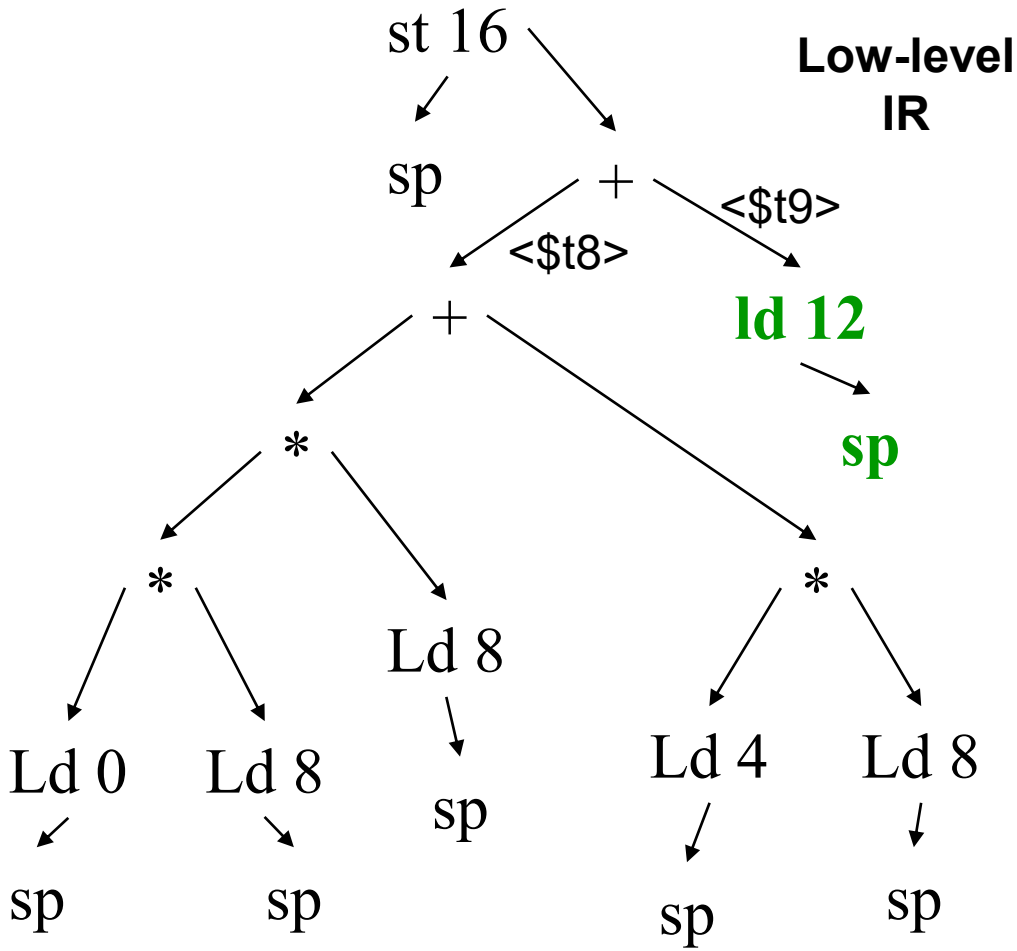
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
```



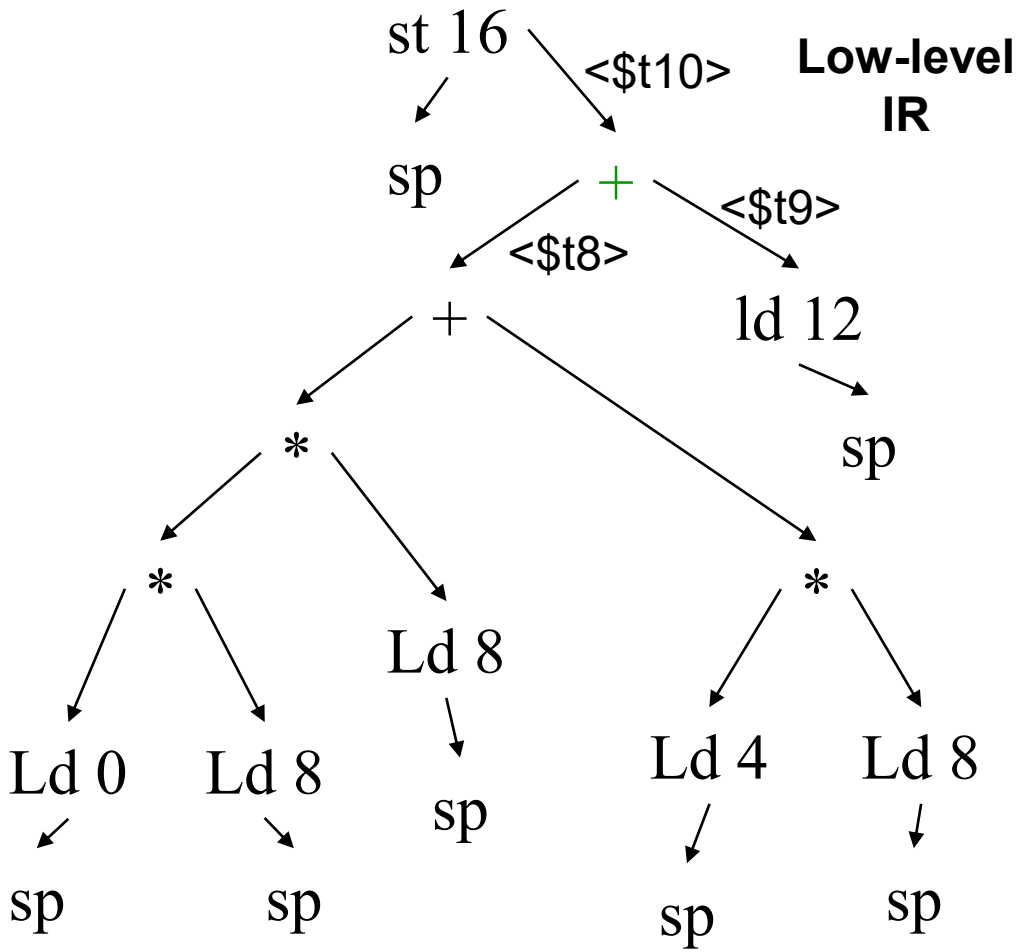
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
Add $t10, $t8, $t9
```

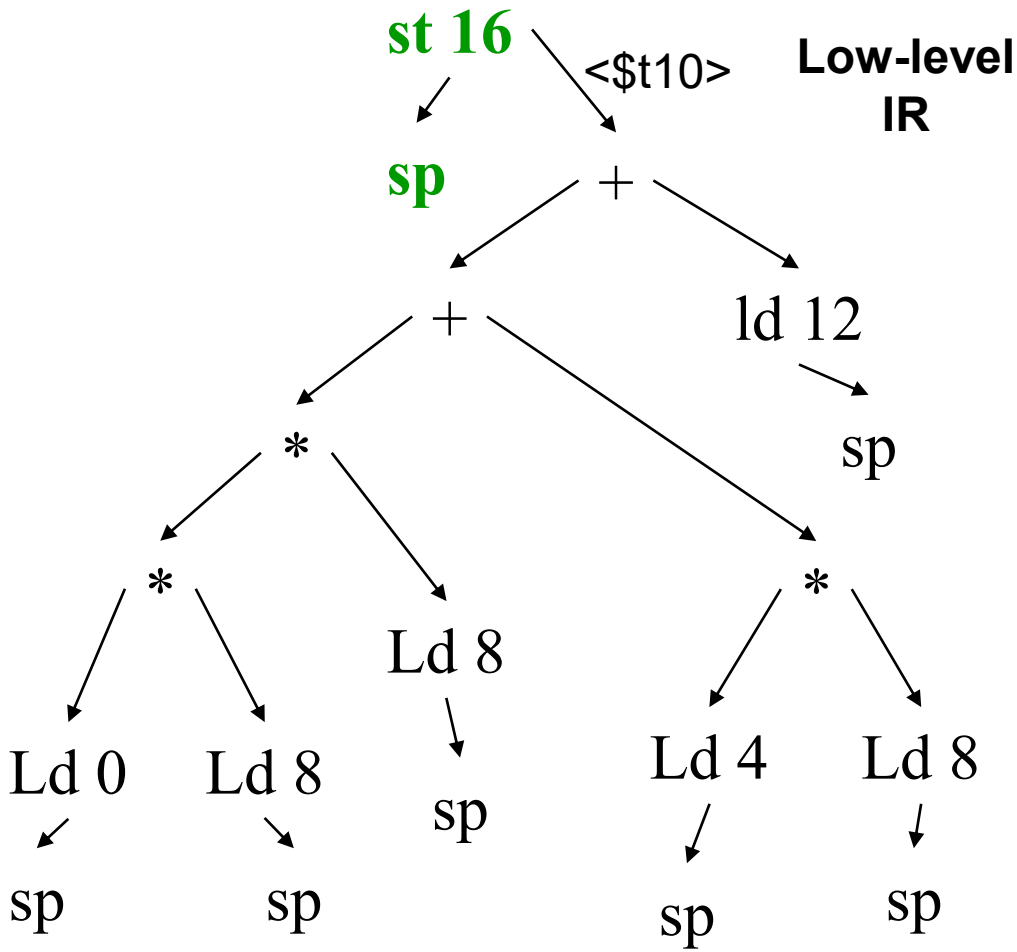


Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
Add $t8, $t7, $t2
lw $t9, 12($sp)
Add $t10, $t8, $t9
Sw  $t10, 16($sp)
```



Code Generation

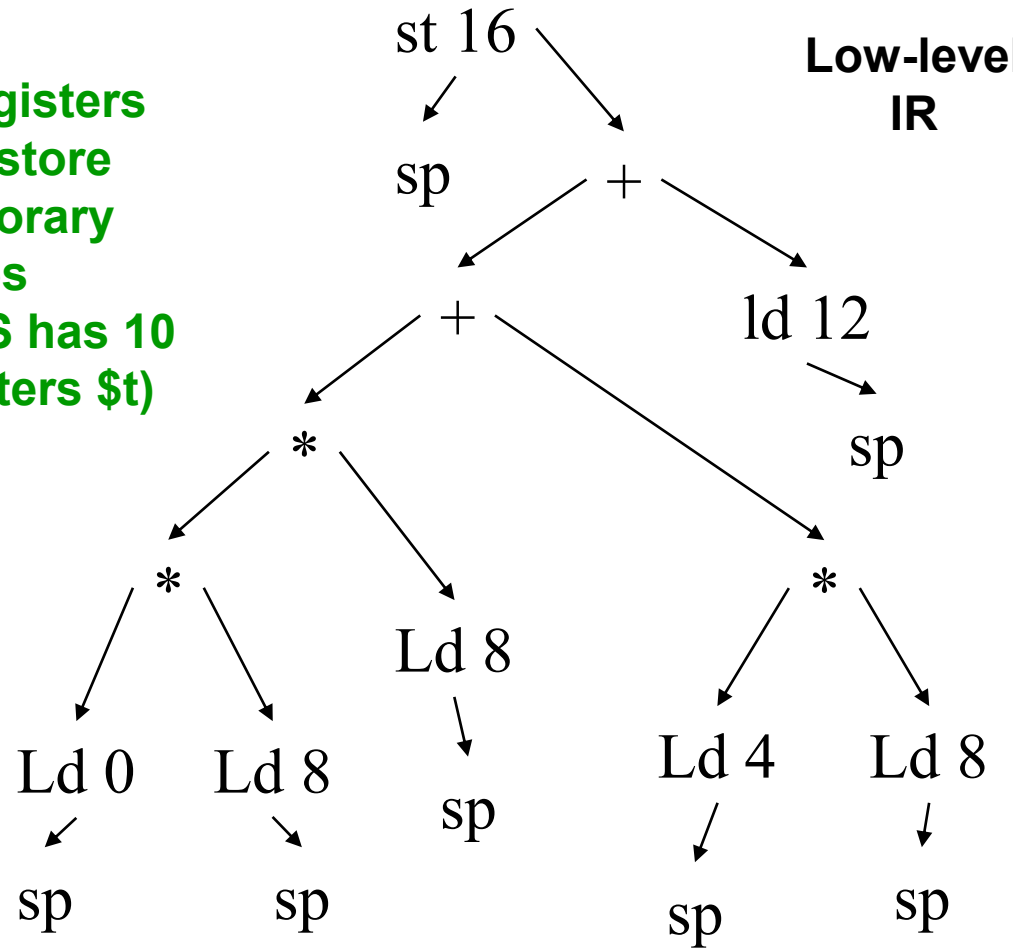
Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t2, $t1, $t0
lw $t3, 0($sp)
lw $t4, 8($sp)
mult $t5, $t3, $t4
lw $t6, 8($sp)
mult $t7, $t5, $t6
add $t8, $t7, $t2
lw $t9, 12($sp)
add $t10, $t8, $t9
sw  $t10, 16($sp)
```

11 registers $t to store temporary values (MIPS has 10 registers $t)

Low-level IR

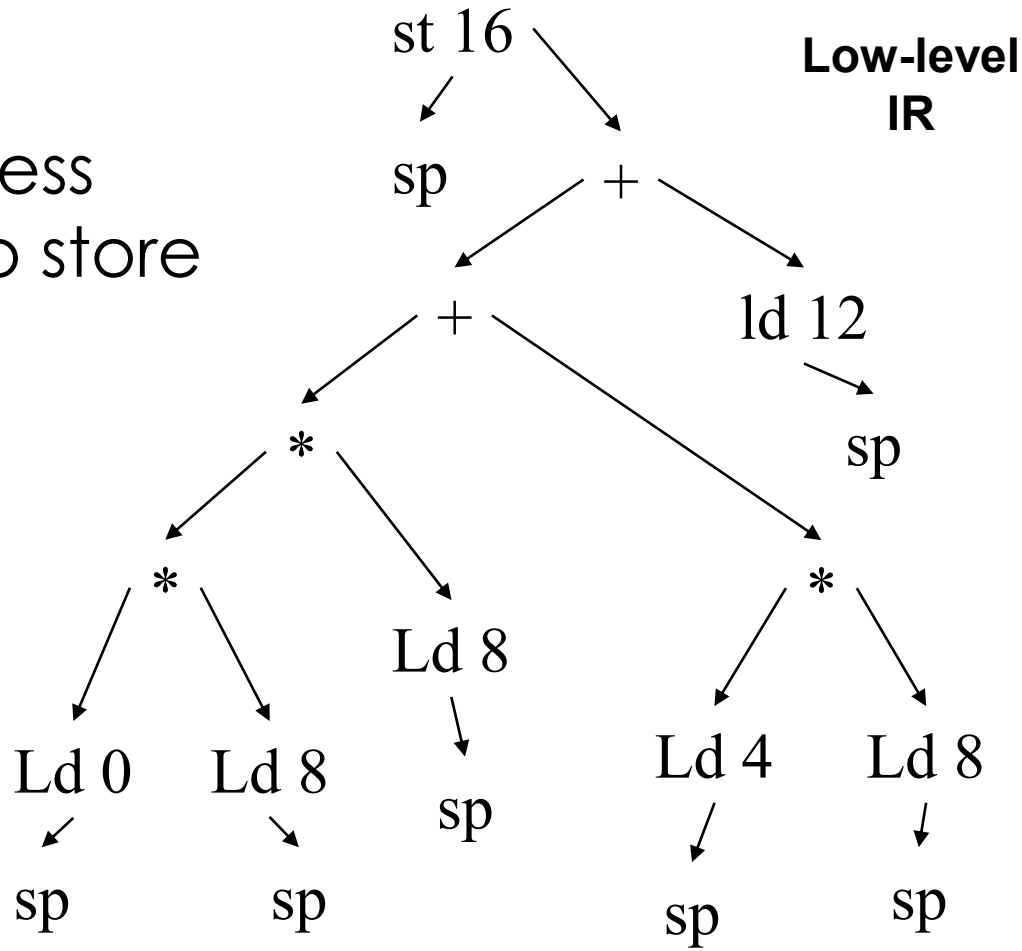


Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

y=a*x*x+b*x+c;

Solution using less registers $t to store temporary values?



Low-level IR

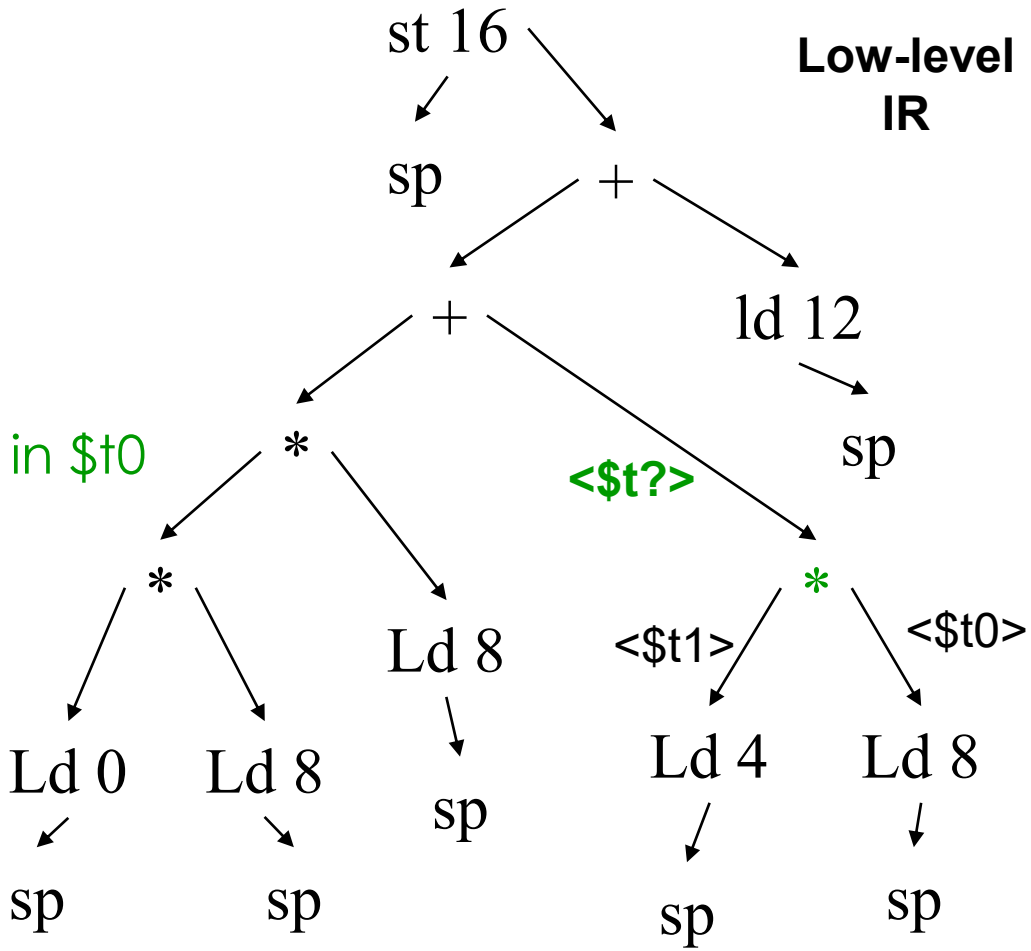
Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

- **lw $t0, 8($sp)**
- **lw $t1, 4($sp)**
- **mult $t?, $t1, $t0**

Result can be stored in $t1 or in $t0



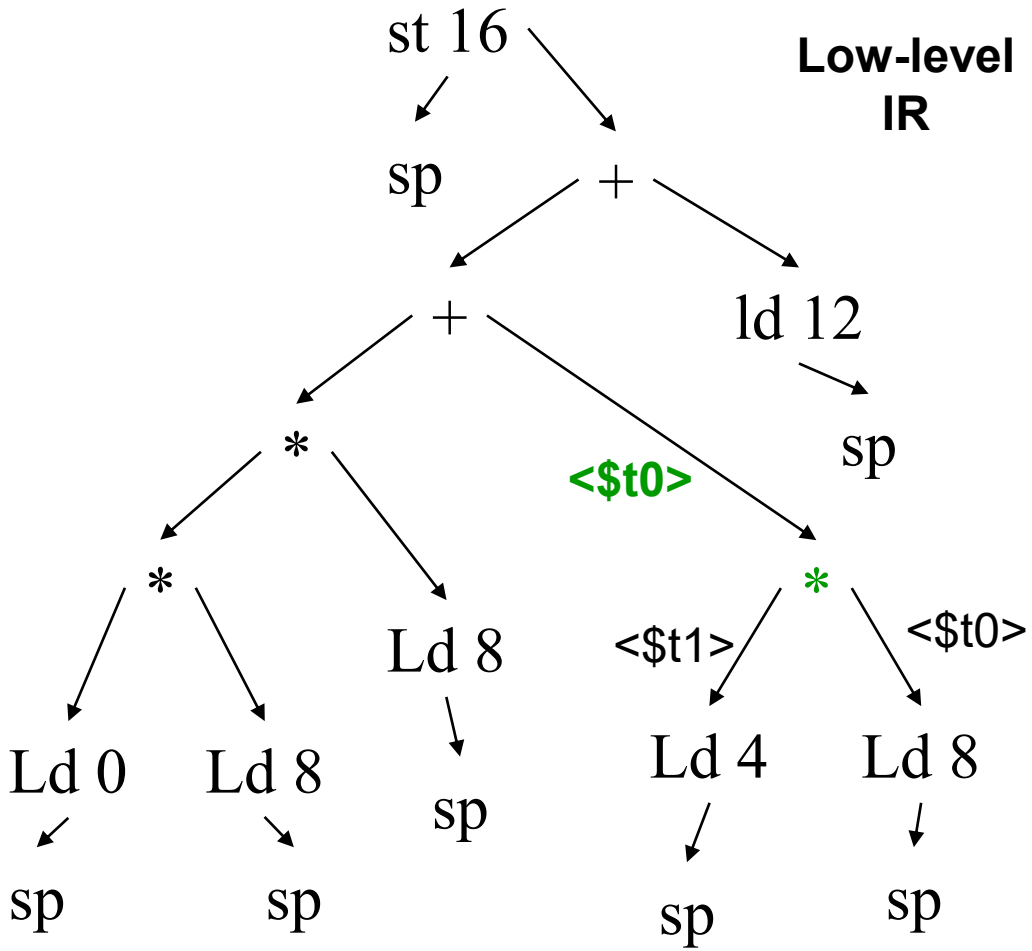
Low-level IR

Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

- **lw $t0, 8($sp)**
- **lw $t1, 4($sp)**
- **mult $t0, $t1, $t0**
- **…**



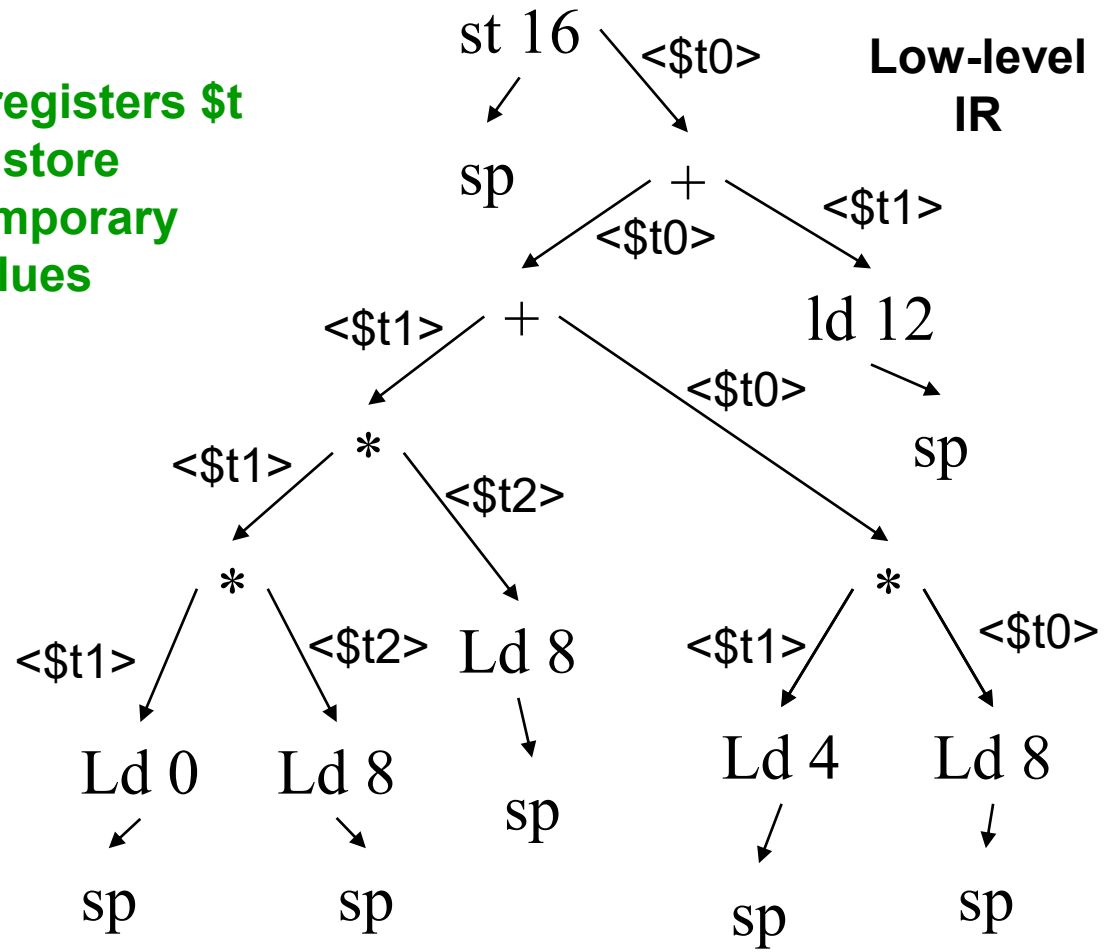
Code Generation

Relative position to $sp
a: 0
b: 4
x: 8
c: 12
y: 16

- y=a*x*x+b*x+c;

```
lw $t0, 8($sp)
lw $t1, 4($sp)
mult $t0, $t1, $t0
lw $t1, 0($sp)
lw $t2, 8($sp)
mult $t1, $t1, $t2
lw $t2, 8($sp)
mult $t1, $t1, $t2
add $t0, $t1, $t0
lw $t1, 12($sp)
add $t0, $t0, $t1
sw  $t0, 16($sp)
```

3 registers $t to store temporary values



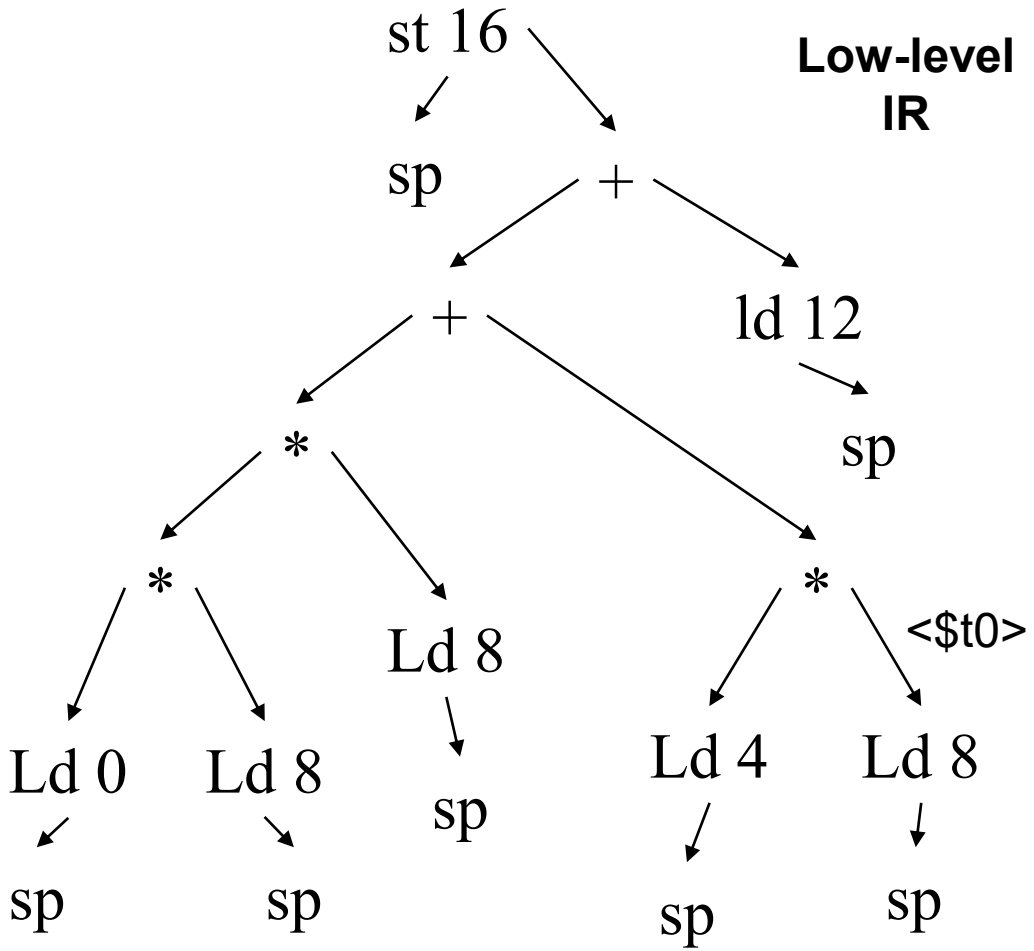
Sethi-Ullman Algorithm for minimum number of registers in arithmetic expressions

CODE GENERATION FROM EXPRESSION TREES

Code Generation: Sethi-Ullman Algorithm

y=a*x*x+b*x+c;

1. Label nodes bottom-up according to the register needs
   - Needed registers in each child node *i* is = → reg_i+1
   - Needed registers in child nodes are ≠ → max(regs)



Relative position to $sp

a: 0

b: 4

x: 8

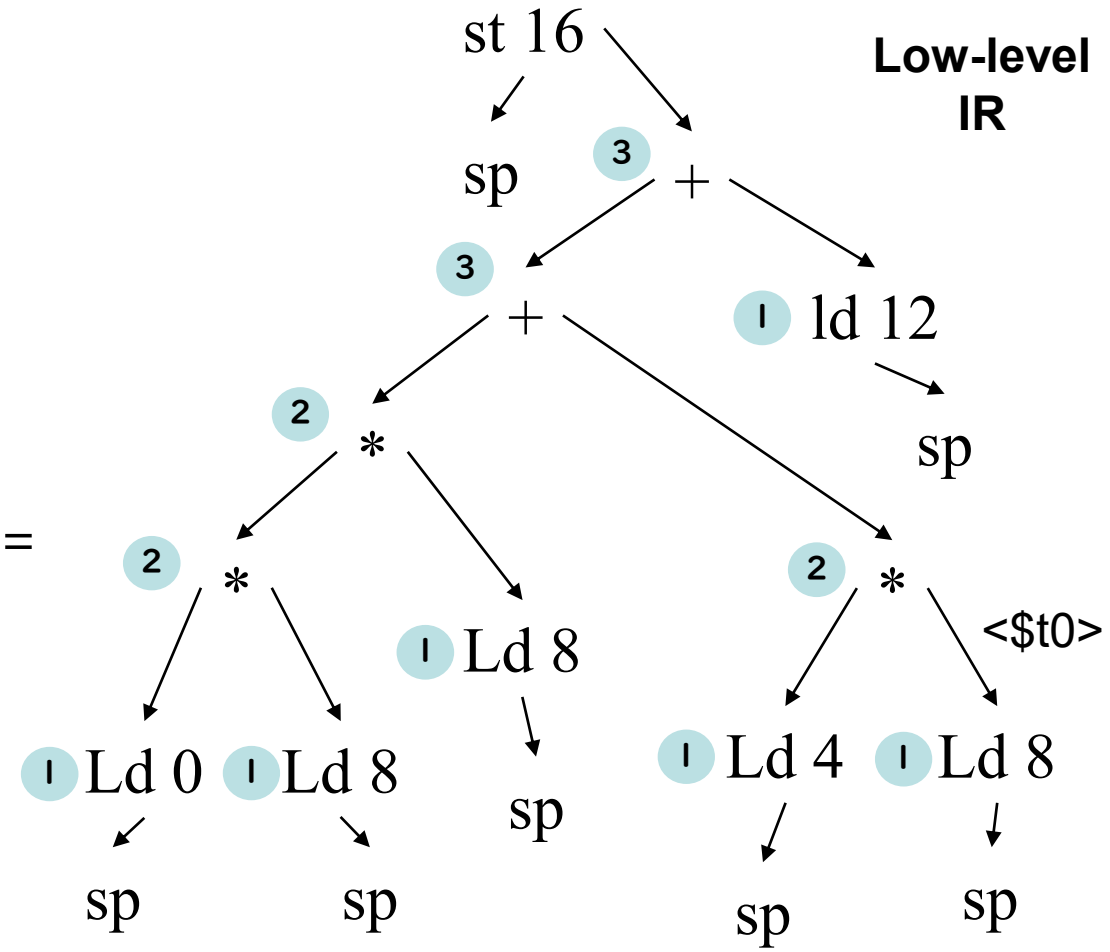
c: 12

y: 16

Code Generation: Sethi-Ullman Algorithm

y=a*x*x+b*x+c;

1. Label nodes bottom-up according to the register needs
   - Needed registers in each child node *i* is = → reg_i+1
   - Needed registers in child nodes are ≠ → max(regs)



Relative position to $sp

a: 0

b: 4

x: 8

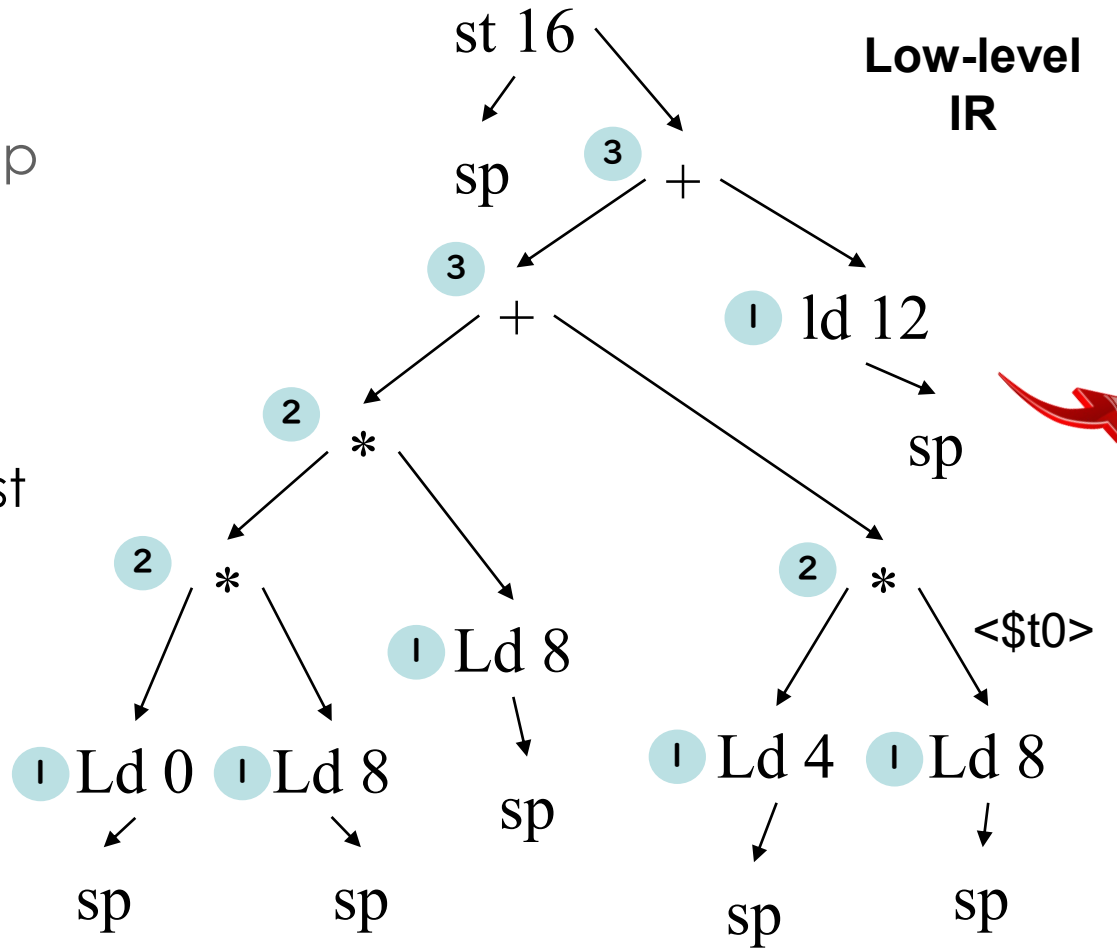
c: 12

y: 16

Code Generation: Sethi-Ullman Algorithm

y=a*x*x+b*x+c;

1. Label nodes bottom-up according to the register needs
2. Traverse recursively tree top-down: first child that requires most registers (left child in the case of equal no. of registers) and emit instruction per node



Relative position to $sp

a: 0

b: 4

x: 8

c: 12

y: 16

```
lw $t1, 0($sp)
lw $t2, 8($sp)
mul $t1, $t2, $t1
lw $t2, 8($sp)
mult $t1, $t1, $t2
lw $t2, 4($sp)
lw $t3, 8($sp)
mult $t2, $t2, $t3
add $t1, $t1, $t2
lw $t2, 12($sp)
add $t1, $t1, $t2
sw  $t1, 16($sp)
```

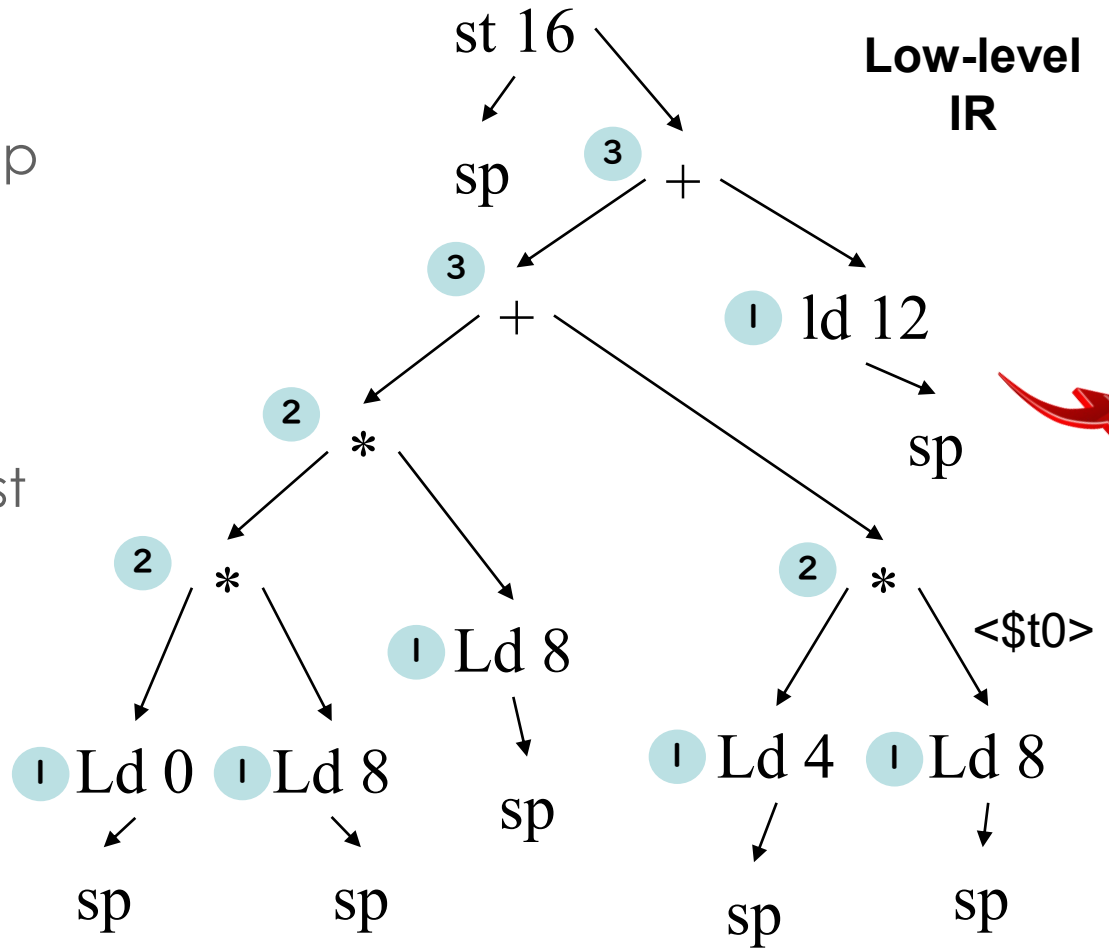
Code Generation: Sethi-Ullman Algorithm

y=a*x*x+b*x+c;

1. Label nodes bottom-up according to the register needs
2. Traverse recursively tree top-down: first child that requires most registers (left child in the case of equal no. of registers) and emit instruction per node

See the algorithm in next slide

Sequence of instructions ok, and 3 registers $t to store temporary values!



Relative position to $sp

a: 0
b: 4
x: 8
c: 12
y: 16

```
lw $t1, 0($sp)
lw $t2, 8($sp)
mul $t1, $t2, $t1
lw $t2, 8($sp)
mult $t1, $t1, $t2
lw $t2, 4($sp)
lw $t3, 8($sp)
mult $t2, $t2, $t3
add $t1, $t1, $t2
lw $t2, 12($sp)
add $t1, $t1, $t2
sw  $t1, 16($sp)
```

Code Generation: Sethi-Ullman Algorithm

Algorithm for code generation:

- Start with T as root and stack with the registers (number given by the labeling)

```
generate(T) {
  if T is a leaf emit("load top(), T");
  elsif T is an internal node with children l and r {
    if regs(r) == 0 { generate(l); emit("op top(), r");}
    if regs(l) >= regs(r) {
       generate(l); R = pop(); generate(r); emit("op R, top()"); push(R);
    } else {  // regs(l) < regs(r)
       swap the top 2 stack elements;
       generate(r); R = pop(); generate(l); emit("op top(), R"); push(R);
       swap the top 2 stack elements;
  }
}
```

Code Generation: Sethi-Ullman Algorithm

- What if the number of registers needed is higher than the actual number of registers?
  - Some results need to be stored in stack!
  - We need to include *Spill* in the generation algorithm (see the Dragon book)

sp

Code Generation for Expressions

- Tree-based IR
  - Sethi-Ullman Algorithm for minimum number of registers in arithmetic expressions (see Dragon, Tiger, Books)
  - *Sethi, Ravi; Ullman, Jeffrey D. (1970), "The Generation of Optimal Code for Arithmetic Expressions", Journal of the Association for Computing Machinery, **17** (4): 715–728, doi:10.1145/321607.321620.*
- DAG-based IR
  - *A. V. Aho and S. C. Johnson. 1976. Optimal Code Generation for Expression Trees. J. ACM 23, 3 (July 1976), 488–501. DOI: https://doi.org/10.1145/321958.321970*

Local Variables Stored in Stack

- Not optimized:
  - Stack accesses require more clock cycles than accesses to internal microprocessor registers
  - Utilization of the stack for all the local variables requires more instructions

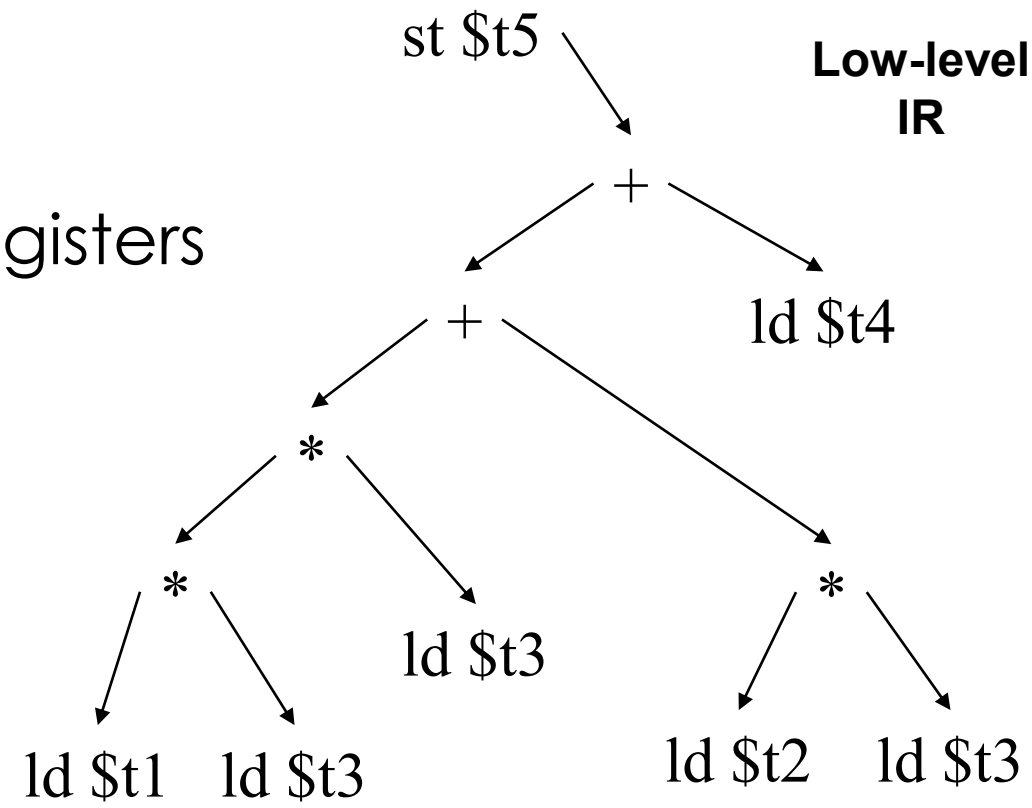
- Thus, allocate registers to as many as possible local variables!

Code Generation: exercise 1

y=a*x*x+b*x+c;

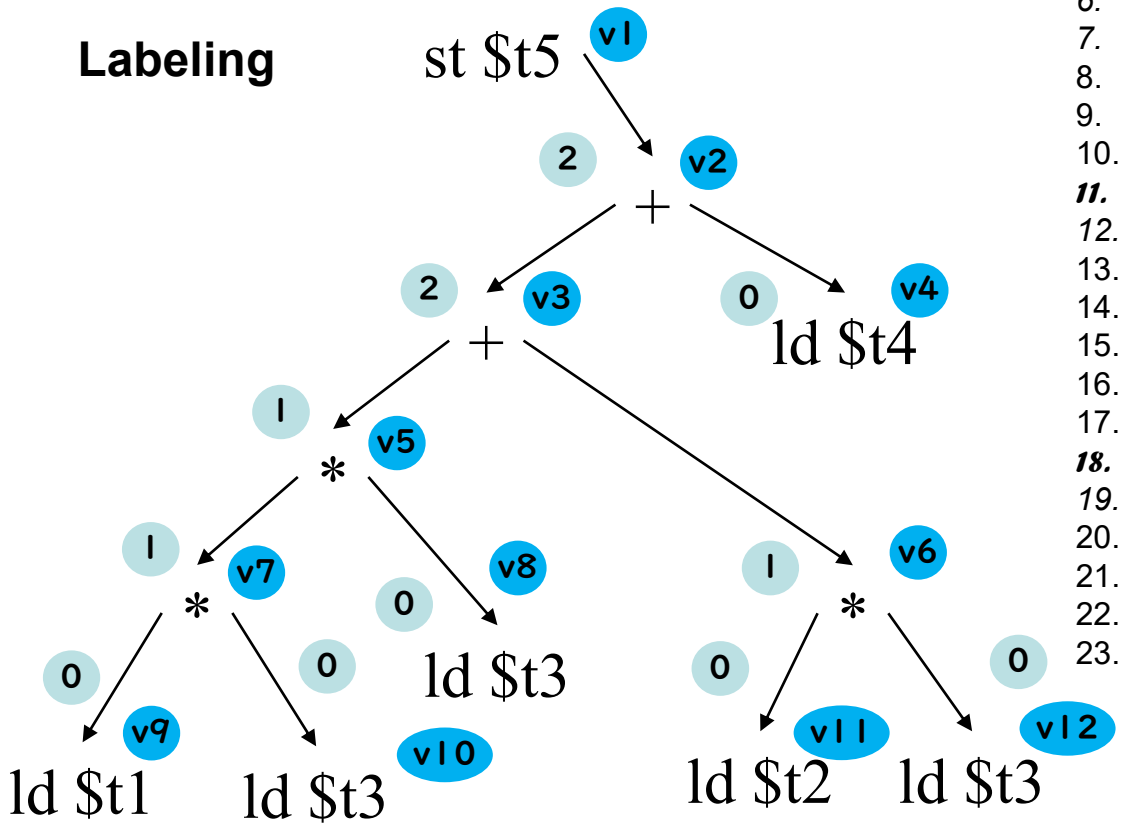
Variables:

- let's assume all in registers
  - a: $t1
  - b: $t2
  - x: $t3
  - c: $t4
  - y: $t5
- Generate the code



Code Generation: exercise 1

y=a*x*x+b*x+c;




```
1.  generate(T, R') {
2.    if T is an end_node emit("op top(), L, R");
3.    elsif T is an internal node with children l and r {
4.      if regs(r) == 0 {
5.        generate(l);
6.        emit("op top(), top(), r");
7.      }
8.     if regs(l) >= regs(r) {
9.        generate(l);
10.       R = pop();
11.       generate(r);
12.       emit("op R, R, top()");
13.       push(R);
14.    } else {  // regs(l) < regs(r)
15.       swap the top 2 stack elements;
16.       generate(r);
17.       R = pop();
18.       generate(l);
19.       emit("op top(), top(), R");
20.       push(R);
21.       swap the top 2 stack elements;
22.    }
23.}
```

1. generate(v2 , $t5)
   5. generate(v3);
      9. generate(v5);
         5. generate(v7);
            2. emit("mul $t6, $t1, $t3");
         6. emit("mul $t6, $t6, $t3");
      10. R = pop();
      11. generate(v6);
            2. emit("mul $t7, $t2, $t3);
         12. emit("add $t7, $t7, $t6); // R is $t6
         13. push $t6;
   6. emit("add $t5, $t6, $t4");

Stack:

$t6	$t7

$t7

$t6	$t7

⇒ 2 registers for auxiliary values: we assume $t6 and $t7

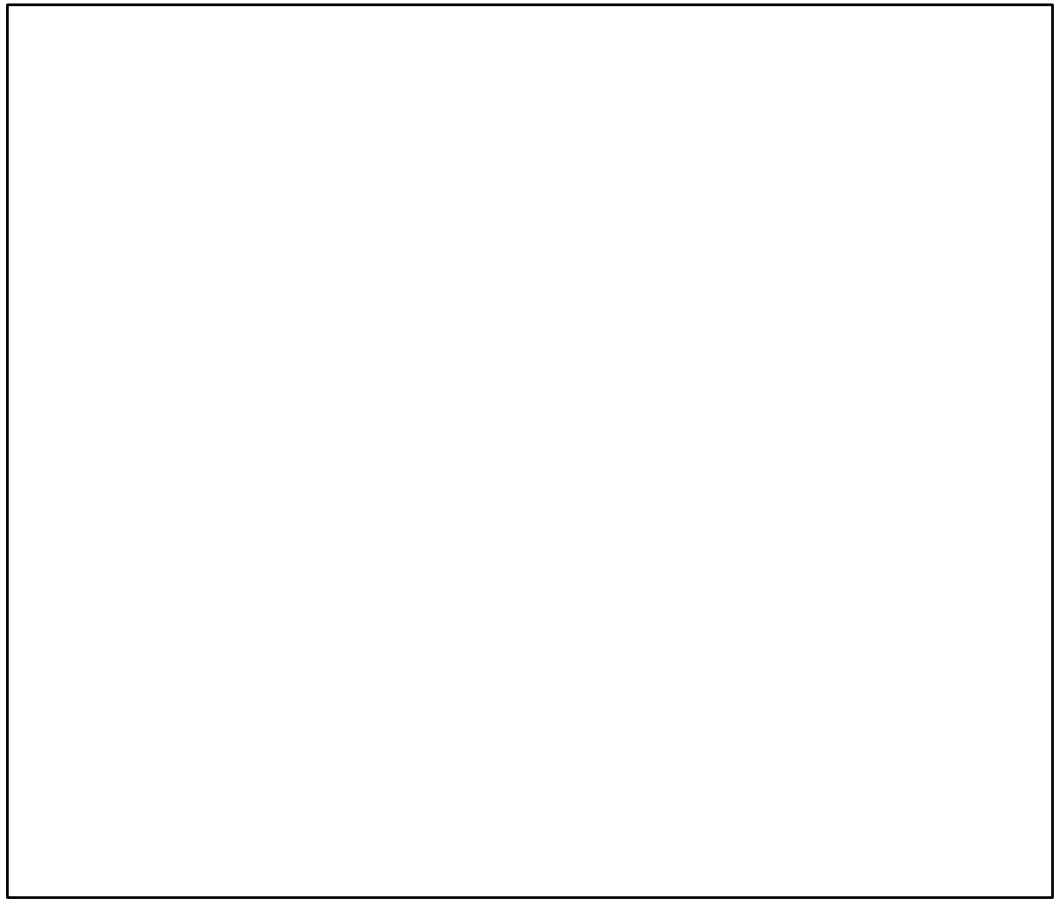
Code Generation: exercise 2

y=(a + b) – (e – (c + d));

Variables:

- let's assume all in registers
  - a: $t1
  - b: $t2
  - c: $t3
  - d: $t4
  - e: $t5
  - y: $t6
- Generate the code

Low-level IR



WHAT ABOUT LOOPS AND CONDITIONAL CONSTRUCTS?

Code Generation

- Use of *templates* to generate assembly code for:
  - If-then and if-then-else constructs
  - Loops

Code Generation

- Templates for If-then and if-then-else constructs

if (test)
** true_body**
else
** false_body**

```
        <do the test>
        boper …, lab_true
        <false_body>
        jump  lab_end
lab_true:
        <true_body>
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
            <stmt1>
lab_init:   <test>
            boper …, lab_true
            jump lab_end
lab_true:
            <body>
            <stmt2>
            jump lab_init
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
                <stmt1>
lab_init:       <test>
                bnoper …, lab_end
                <body>
                <stmt2>
                jump lab_init
lab_end:
```

Code Generation

- Templates for loops

for(stmt1;test;stmt2)
body

```
            <stmt1>
            jump lab_init
lab_true:   <body>
            <stmt2>
lab_init:   <test>
            boper …, lab_true
lab_end:
```

Code Generation

- Depending on the target machine, code generation may need:
  - Algorithms for **instruction selection**
    - Previously, we assumed that each tree node results in one machine instruction!
    - What does happen when there are instructions that cover more than one tree node?
- Sequence of instructions
  - It has impact on registers needed, stack depth, etc.
  - Dealing with pipelining and/or with multiple units need **instruction scheduling**
- Next steps: **Instruction Selection**, **Scheduling** and **Register Allocation**

CODE GENERATION HINTS

Hints for Code Generator

- Go down slowly and step-by-step in the abstraction level
- Use the number of stages you need:
  - Even if each stage does only one thing!
  - Easier to debug, easier to handle the problems
- Mantain the abstraction level consistent
  - IR must mantain the semantic correct everytime!
  - One may need to do optimizations between stages
- Use *assertions*
  - An *assertion* to verify some condition that should apply
  - They help to find bugs

Hints for Code Generator

- Start doing simple code generation, even if naïf
  - Ok to generate code for: 0 + 1*x + 0*y
- The *runtime* library is our friend!
  - Do not try to generate assembly code when there exist library routines with the same functionality
  - Example: malloc

Hints for Code Generator

- Remember that the optimizations come later
  - It is the role of the optimizer to perform the optimizations
  - Think that the optimizer needs to restructure the code according to the portfolio of optimizations it integrates
  - Examples: register allocation, algebraic simplifications, constant propagations
- Use a good test infrastructure
  - Regressive Test
    - If a program originates a bug then add it to the test suite
  - Use makefiles
    - to execute automatically the compiler under development over the test suite and to verify if all of the examples in the test suite pass in the tests (it can imply the use of a simulator of the target architecture)
  - Use the best software engineering practices

Code Generation Example

TARGETING THE JVM

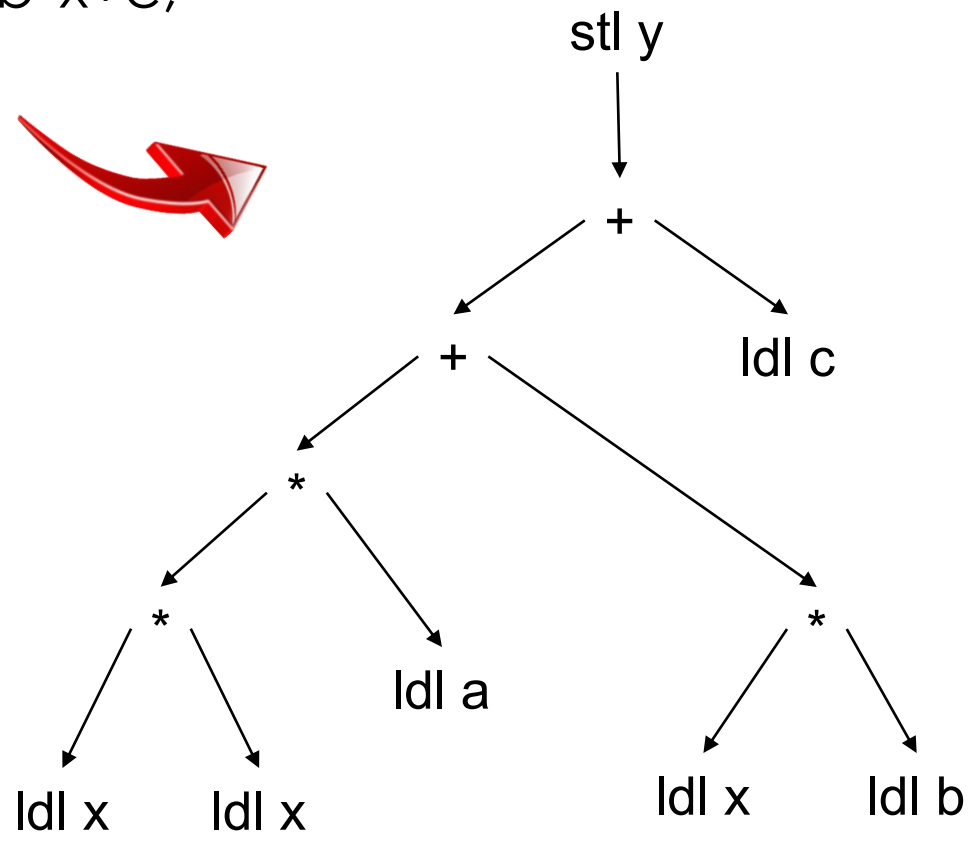
Targeting the JVM

- Instruction Selection does not require complex algorithms
- JVM instructions have very few tree patterns with overlapping
- We can use a Naive/Canonical Generation prioritizing the use of JVM instructions that cover the largest number of tree nodes
  - Example, considering that variable "i" is assigned to JVM local variable "3"

i++;	iload 3 iconst 1 iadd istore 3	iinc 3, 1

Targeting the JVM

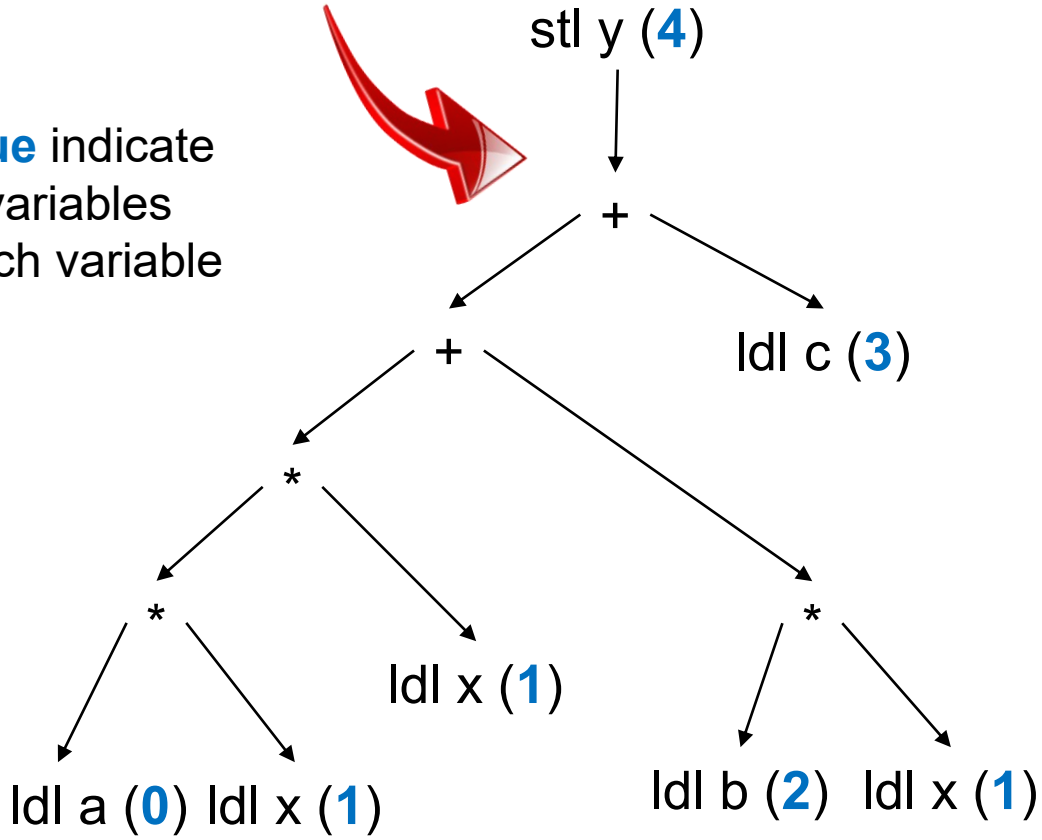
- Example
  - y=a*x*x+b*x+c;



Targeting the JVM

- Example
  - y=a*x*x+b*x+c;

Numbers in **blue** indicate the JVM local variables assigned to each variable

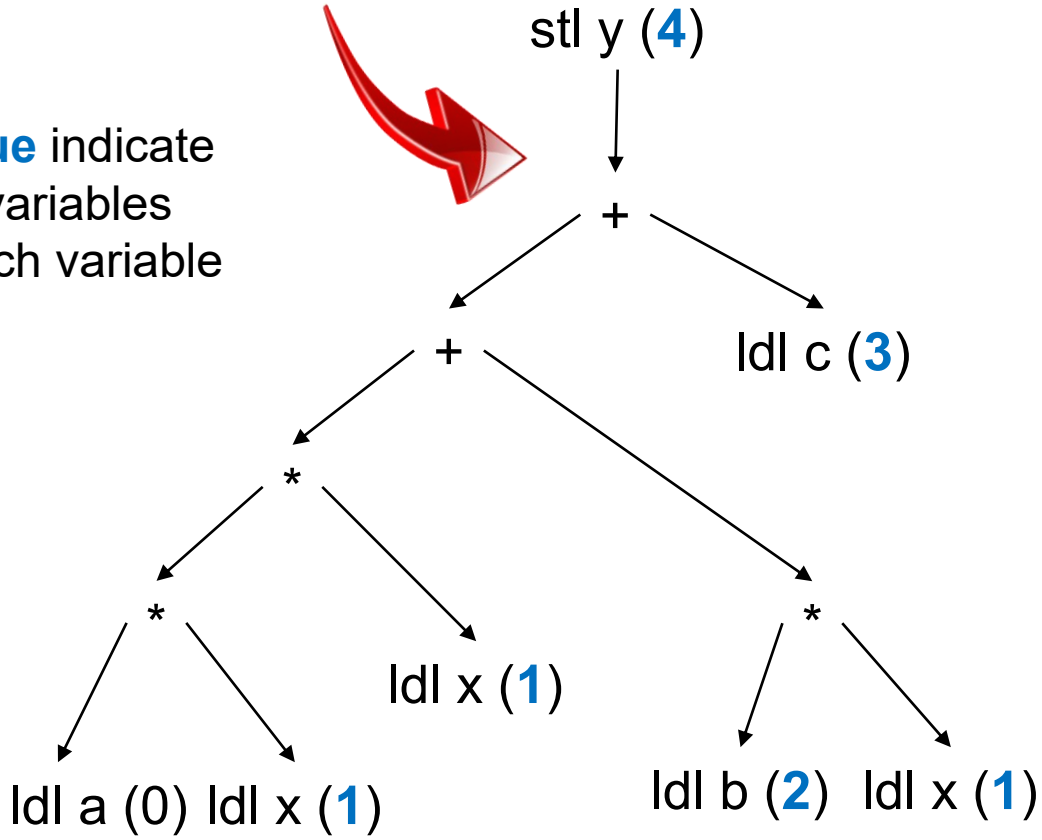


JVM code generated:

Targeting the JVM

- Example
  - y=a*x*x+b*x+c;

Numbers in **blue** indicate the JVM local variables assigned to each variable



Javac generates (stack depth=3):

```
iload_0
iload_1
imul
iload_1
imul
iload_2
iload_1
imul
iadd
iload_3
iadd
istore  4
```

Targeting the JVM

- Uses an operand stack and a table of local variables
- Content on top positions of operand stack must be according to the needed operands foreach operation
  - E.g., iadd requires the top two operands on the stack