



# Principles of Compiler Construction

---

**Lecturer: CHANG HUIYOU**

**Note that most of these slides were created by:**

**Prof. Wen-jun LI**

**Dr. Zhong-mei SHU**

**Dr. Han LIN**



# Lecture 11. Code Generation

---

1. Introduction
2. Abstraction of Target Machines
3. Implementation of Procedures
4. Optimization of Basic Blocks
5. A Simple Code Generator
6. Peephole Optimization

# 1. Introduction

---

- Code generators focus on
  1. Instruction selection
    - Choose appropriate target-machine instructions to implement IR statements.
  2. Register allocation and assignment
    - Make full use of registers, the fastest computational unit.
  3. Instruction ordering
    - Decide the order in which the execution of instructions is scheduled.
    - Also called **instruction scheduling**.

# Instruction Selection

---

- Both the speed and the size cost of the generated code should be considered.
  - Trade-off and consequence.
- Example 1: **a = a + 1**
  - LD R0, a
  - ADD R0, R0, #1
  - ST a, R0
  - INC a
- Example 2: set register R0 to 0
  - LD R0, #0
  - XOR R0, R0

# Register Allocation and Assignment

---

- Include two subproblems
  - Register allocation
    - Select the set of variables that will reside in registers at each point in the program.
  - Register assignment
    - Pick the specific register that a variable will reside in.
- A difficult problem in code generation.

# Instruction Scheduling

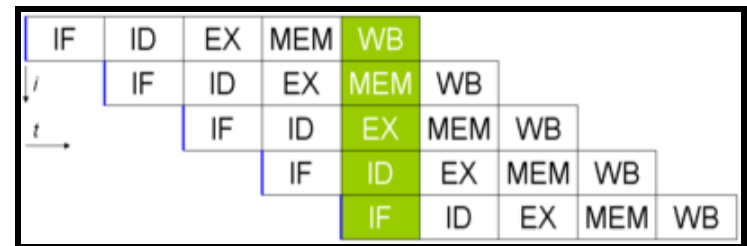
---

- Important for modern pipelined processors
  - RISC: Reduced Instruction Set Computer
    - RISC vs. CISC
  - Data Hazards (**stalls** in a pipeline): attempt to use a datum before it becomes available in register.
    - A value written to a register will not be available in its following cycles.
    - The following cycles should be utilized by instructions that does not depend on the value
      - Or they will be wasted.
    - Optimization: clustering instructions with no dependencies together, while preserving original semantics.

# Instruction Scheduling (cont')

- An example: **(a + b) + c** on SGI's MIPS 2000

- LD R1, a
- LD R2, b
- NOP**
- ADDi R1, R1, R2
- LD R2, c
- NOP**
- ADDi R1, R1, R2



指令流水线:

- Optimal generated code

- LD R1, a
- LD R2, b
- LD R3, c
- ADDi R1, R1, R2
- NOP
- ADDi R1, R1, R3

取指令

译码

执行

访存

写回寄存器堆

## 2. Abstraction of Target Machines

---

- Instruction set
- Addressing modes
- Program and instruction costs



# Instruction Set

---

- Supports the following instructions

- LD        dst, addr        // load
- ST        x,     Ri        // store
- OP        dst, src1, src2   // ADD, SUB, ...  
  OP        dst, src1        // unary operation
- BR        L                // branch
- Bcond    r,        L        // conditional branch

# Addressing Modes

---

- Addressing in instructions

- **#C** // immediate constant, cost = 1
- **x** // absolute, cost = 1
- **\*x** // indirect memory, cost = 1
- **R** // direct register, cost = 0
- **\*R** // indirect register, cost = 0
- **a(Ri)** // direct indexed, cost = 1
  - **a** is a variable or a constant
  - LD R1, a(R2)
    - $R1 = \text{contents}(a + \text{contents}(R2))$
  - LD R1, 100(R2)
    - $R1 = \text{contents}(100 + \text{contents}(R2))$
- **\*a(Ri)** // indirect indexed, cost = 1
  - LD R1, \*100(R2)
    - $R1 = \text{contents}(\text{contents}(100 + \text{contents}(R2)))$

# Program and Instruction Costs

---

- Cost of an instruction
  - = 1 + operand\_addr\_cost
- Cost of a program
  - = total of instruction costs
- Examples

- LD R0, R1 // cost = 1
- LD R0, x // cost = 2
- LD R1, \*100(R2) // cost = 2



## 3. Implementation of Procedures

---

- Static Allocation
- Stack Allocation

# Static Allocation

- Implementation of **call** *callee*

- ...  
ST *callee.staticArea*, *#here + 20* // return address  
BR *callee.codeArea* // jump to Callee  
...

- Implementation of **return**

- ...  
BR *\*callee.staticArea* // return to Caller

Assume that the return address is saved at the **beginning** of the activation record. And  
**20 = 5 words \* 4 bytes/word**

# An Example: Three-Address Code

---

// code for **c**

action<sub>1</sub>

call p

action<sub>2</sub>

halt

// code for **p**

action<sub>3</sub>

return

# An Example: Implementation

```
// code for c
100  action1
120  ST  364,  #140    // return address
132  BR  200          // call p
140  action2
160  HALT              // return to operating system
...

// code for p
200  action3
220  BR  *364          // return to its caller
....

// activation record for c
300                                     // return address to Caller
304                                     // local data for c
....

// activation record for p
364  140                         // return address to Caller
368                                     // local data for p
```

# Stack Allocation

- Implementation of initialization

- `LD SP, #stackStart` // initialize the stack
  - `...` // code for main()
  - `HALT` // terminate

- Implementation of **call** *callee*

- `...`
  - `ADD SP, SP, #caller.recordSize` // push an AR
  - `ST *SP, #here + 16` // return address
  - `BR callee.codeArea` // jump to Callee
  - `SUB SP, SP, #caller.recordSize` // pop the AR
  - `...`

DBv2 must assume that  
addressing **0(SP)** has no  
additional cost

- Implementation of **return**

- `...`
  - `BR *0(SP)` // return to Caller



# An Example: Three-Address Code

// code for **m**

action<sub>1</sub>

call q

action<sub>2</sub>

halt

// code for **p**

action<sub>3</sub>

return

// code for **q**

action<sub>4</sub>

call p

action<sub>5</sub>

call q

action<sub>6</sub>

call q

return

# An Example: Implementation

```
100 LD    SP, #600
108 action1
128 ADD SP, SP, #msize
136 ST    *SP, #152
144 BR    300
152 SUB SP, SP, #msize
160 action2
180 HALT
```

code for p

```
...
200 action3
220 BR    *0(SP)
```

code for q

```
....
300 action4
320 ADD SP, SP, #qsize
328 ST    *SP, #344
```

```
336 BR    200
344 SUB SP, SP, #qsize
352 action5
372 ADD SP, SP, #qsize
380 ST    *SP, #396
388 BR    300
396 SUB SP, SP, #qsize
404 action6
424 ADD SP, SP, #qsize
432 ST    *SP, #448
440 BR    300
448 SUB SP, SP, #qsize
456 BR    *0(SP)
```

blue for call  
sequence

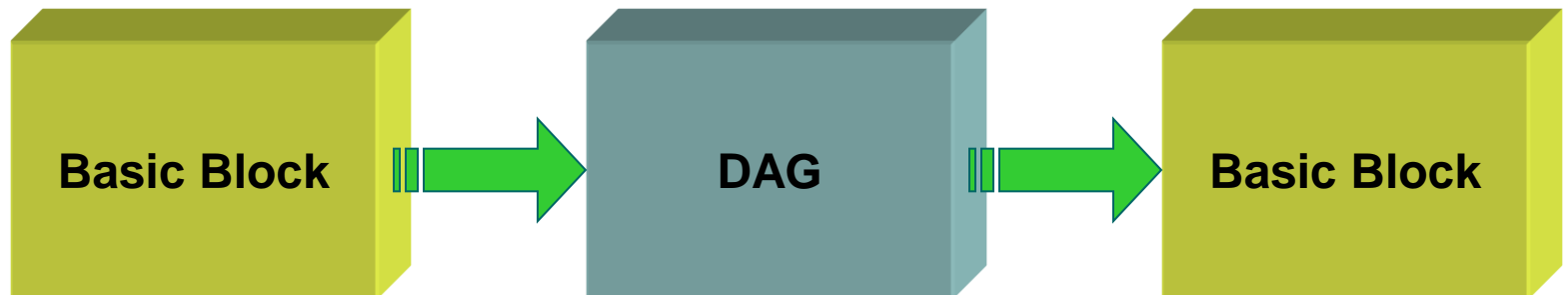
```
...
600
```

stack

## 4. Optimization of Basic Blocks

---

- Approach



Petri Nets in  
Super Compiler

# Basic Blocks

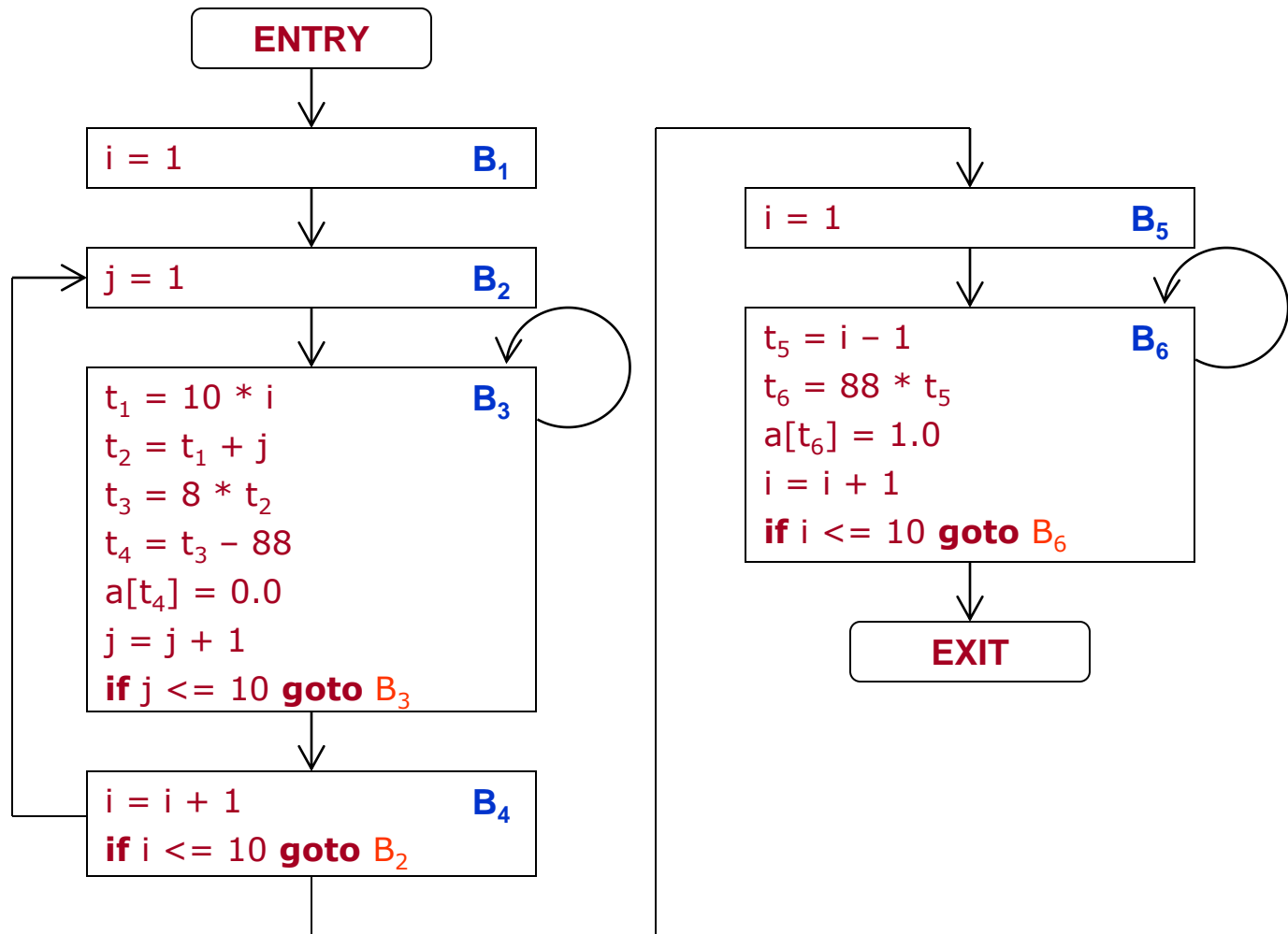
---

- Single entry and single exit
- 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
```

# Flow Graphs



# Construction of Flow Graphs

---

- Partition three-address instructions into basic blocks
  - **Leader:** the 1<sup>st</sup> instruction in a basic block.
  - Algorithm to find all leaders
    1. The 1<sup>st</sup> three-address instruction
    2. Target of a conditional or unconditional jump
    3. Instruction immediately follows a conditional or unconditional jump

# Liveness and Next-Use Information

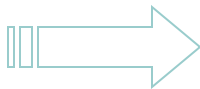
- Calculate liveness and next-use info in a basic block: backward scanning
  - Initialize: for each variable  $v$ ,
    - $v.\text{nextUse} = \text{none}$ ;
    - $v.\text{liveness} = v \text{ is temporary ? false : true}$ ;
  - For each  $i: x = y + z$ 
    - Attach information of  $x$ ,  $y$  and  $z$  to instruction  $i$ ;
    - $x.\text{liveness} = \text{false}$ ;
    - $x.\text{nextUse} = \text{none}$ ;
    - $y.\text{liveness} = z.\text{liveness} = \text{true}$ ;
    - $y.\text{nextUse} = z.\text{nextUse} = i$ ;

Conservative.  
Global data-flow  
analysis in practice

# An Example

- Information are stored at the entry of each variables in the symbol table

(1)  $t = a - b$   
 (2)  $u = a - c$   
 (3)  $v = t + u$   
 (4)  $d = v + u$



(1)  $t^{(3),T} = a^{(2),T} - b^{-},T$   
 (2)  $u^{(3),T} = a^{-},T - c^{-},T$   
 (3)  $v^{(4),T} = t^{-},F + u^{(4),T}$   
 (4)  $d^{-},T = v^{-},F + u^{-},F$

| Var.     | Next-Use |     |     |     |     | Liveness |     |     |     |     |
|----------|----------|-----|-----|-----|-----|----------|-----|-----|-----|-----|
|          | Init     | (4) | (3) | (2) | (1) | Init     | (4) | (3) | (2) | (1) |
| <b>a</b> | –        |     |     | (2) | (1) | T        |     |     | T   | T   |
| <b>b</b> | –        |     |     |     | (1) | T        |     |     |     | T   |
| <b>c</b> | –        |     |     | (2) |     | T        |     |     | T   |     |
| <b>d</b> | –        | –   |     |     |     | T        | F   |     |     |     |
| <b>t</b> | –        |     | (3) |     | –   | F        |     | T   |     | F   |
| <b>u</b> | –        | (4) | (3) | –   |     | F        | T   | T   | F   |     |
| <b>v</b> | –        | (4) | –   |     |     | F        | T   | F   |     |     |

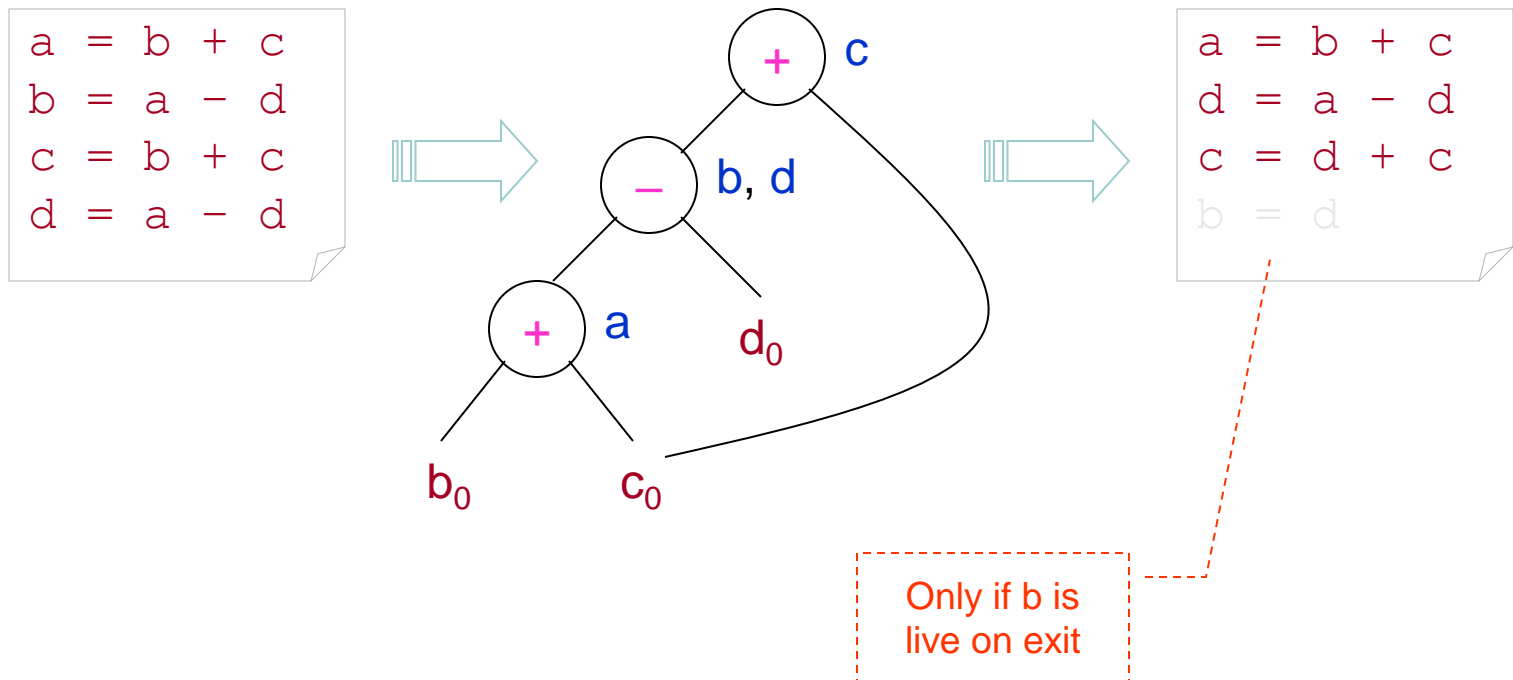


# Optimization Based on DAGs

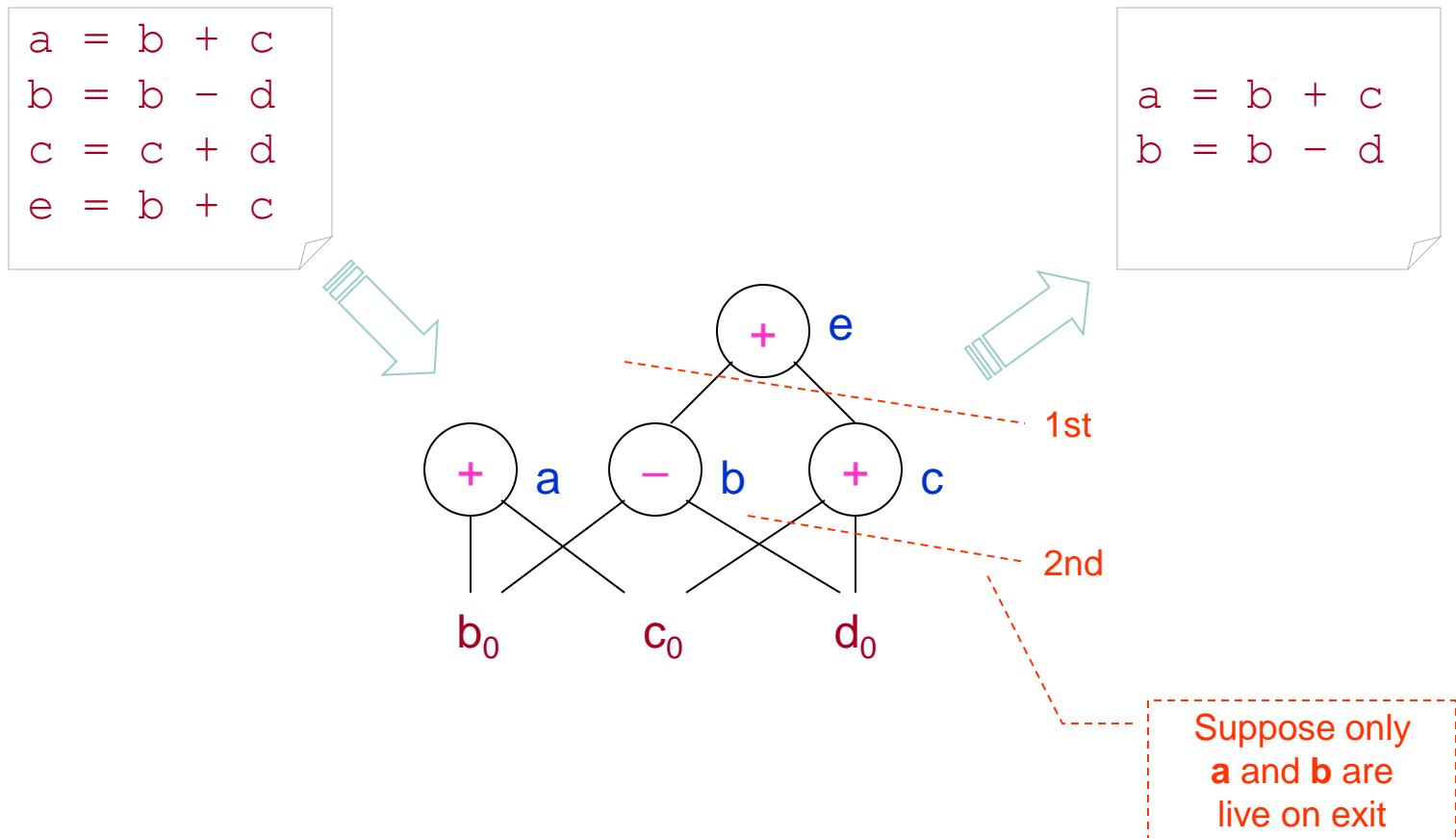
---

- Perform several code-improving transformations
  - Elimination of local common subexpressions
  - Elimination of dead code
  - Reordering of independent statements
  - Application of algebraic laws

# Local Common Subexpressions



# Dead Code



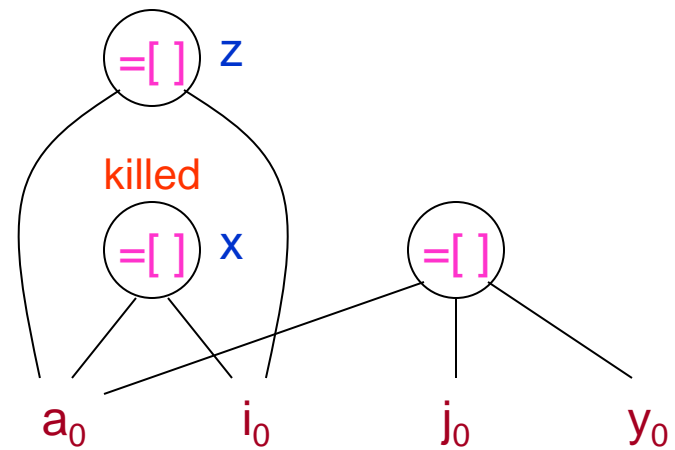
# Algebraic Identities

---

- Arithmetic identities
  - $x + 0 = 0 + x = x$
  - $x * 1 = 1 * x = x$
  - $x - 0 = x$
  - $x / 1 = x$
  - ...
- Local reduction in strength
  - $2 * x \Rightarrow x + x$
  - $x / 2 \Rightarrow x * 0.5$
  - ...
- Constant folding
  - $2 * 3.14 \Rightarrow 6.28$
  - ...

# Array References

```
x = a[i]  
a[j] = y  
z = a[i]
```

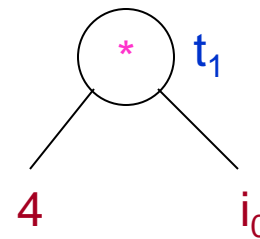


# An Example:

## Construction a DAG from a BB (1)

---

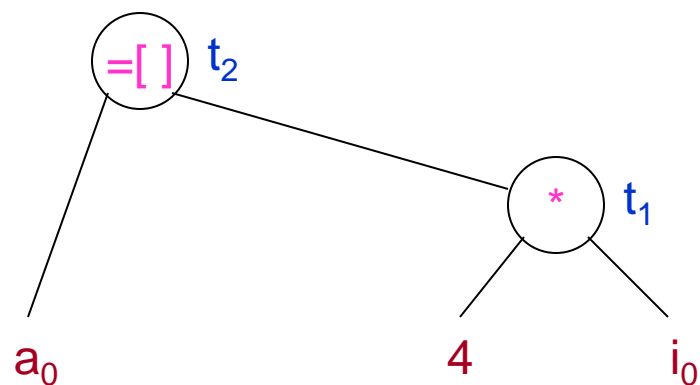
```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



# An Example:

## Construction a DAG from a BB (2)

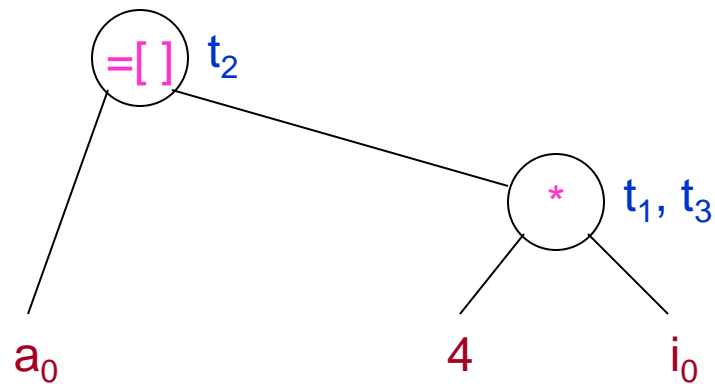
```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



# An Example:

## Construction a DAG from a BB (3)

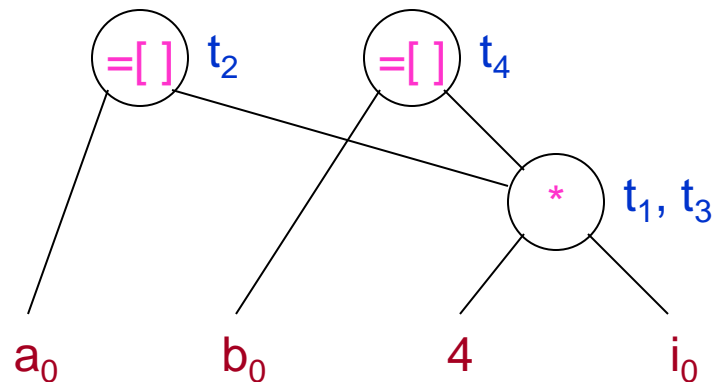
```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```





# An Example: Construction a DAG from a BB (4)

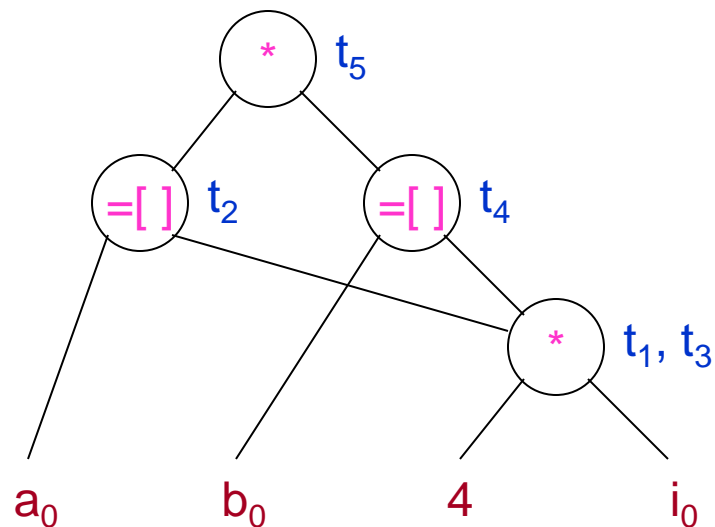
```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



# An Example:

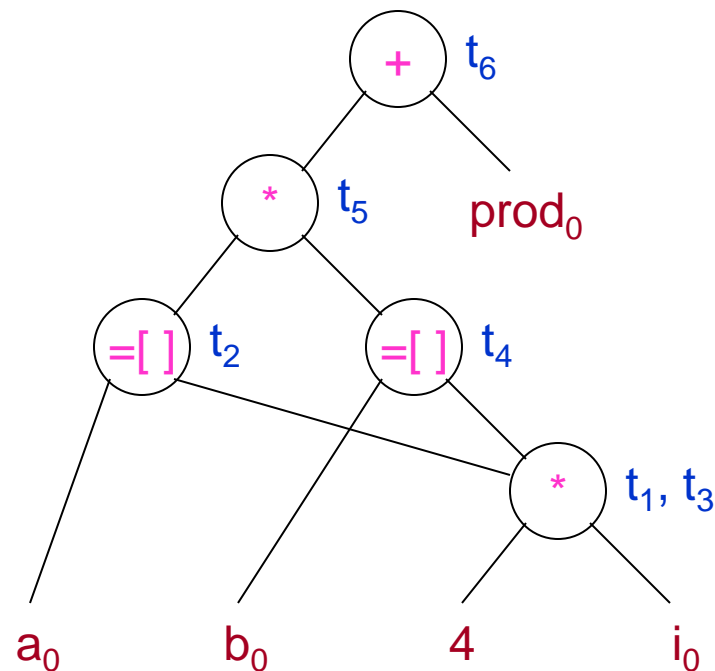
## Construction a DAG from a BB (5)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



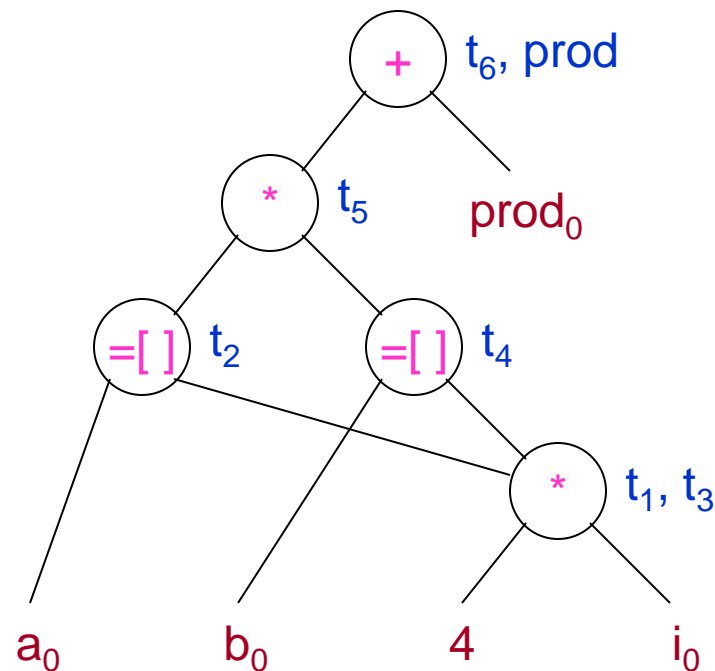
# An Example: Construction a DAG from a BB (6)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



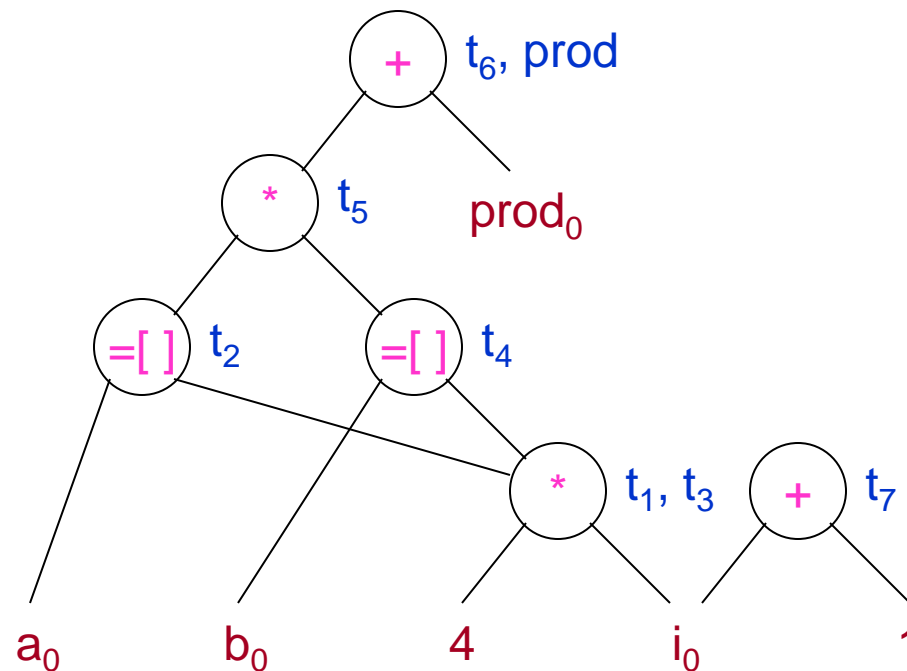
# An Example: Construction a DAG from a BB (7)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



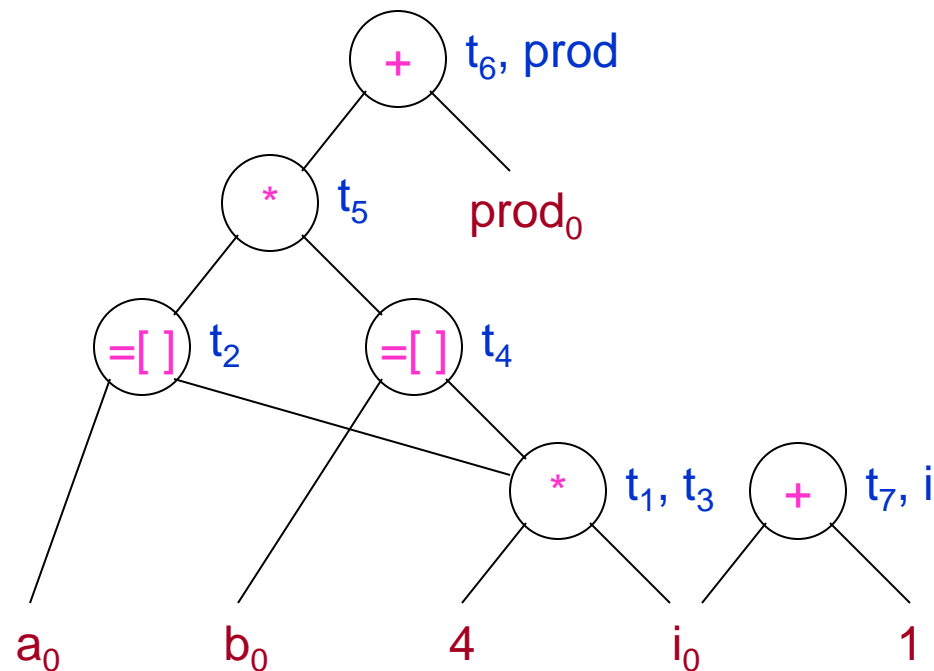
# An Example: Construction a DAG from a BB (8)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



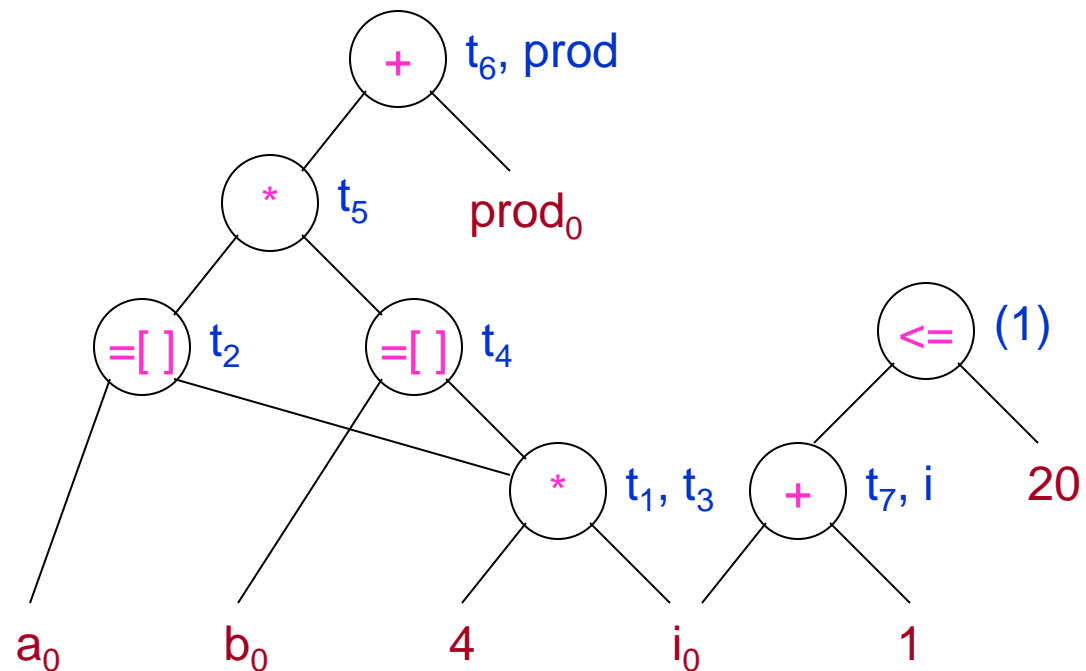
# An Example: Construction a DAG from a BB (9)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



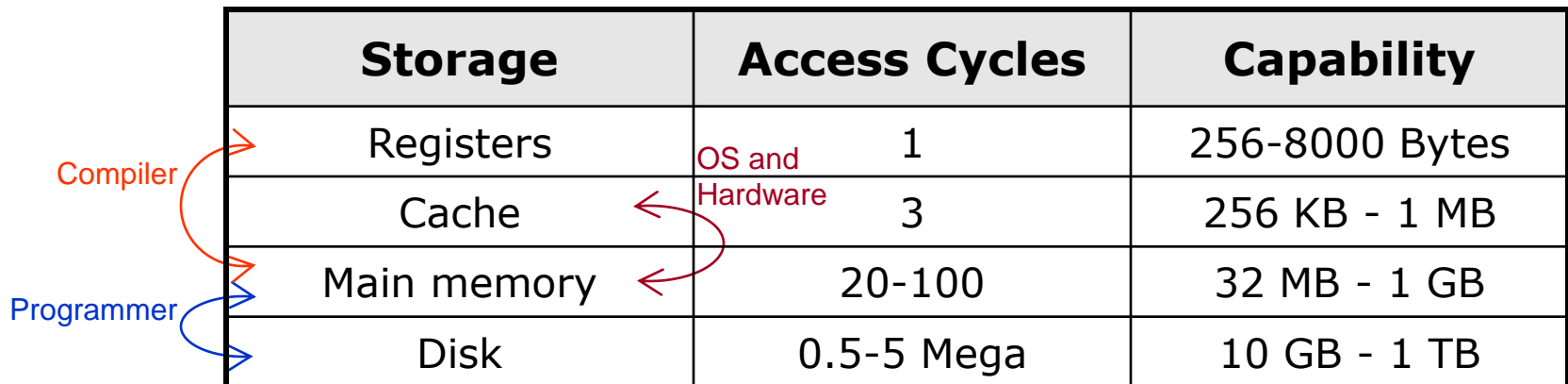
# An Example: Construction a DAG from a BB (10)

```
t1 = 4 * i  
t2 = a[t1]  
t3 = 4 * i  
t4 = b[t3]  
t5 = t2 * t4  
t6 = t5 + prod  
prod = t6  
t7 = i + 1  
i = t7  
if i <= 20 goto (1)
```



# 5. A Simple Code Generator

- What does "simple" mean ?
  - Generate code for a single basic block.
- Motivation
  - Make full use of registers to the best advantage.



| Storage     | Access Cycles | Capability     |
|-------------|---------------|----------------|
| Registers   | 1             | 256-8000 Bytes |
| Cache       | 3             | 256 KB - 1 MB  |
| Main memory | 20-100        | 32 MB - 1 GB   |
| Disk        | 0.5-5 Mega    | 10 GB - 1 TB   |



# Register and Address Descriptors

---

- Register descriptor
  - **RegDesc: Register**  $\rightarrow 2^{\text{Variable}}$
- Address descriptor
  - **AddrDesc: Variable**  $\rightarrow 2^{\text{Location}}$
  - **Location = Register  $\cup$  Memory**

# Code-Generation Algorithm

---

- Based on function `getReg(x = y op z)`
  - Select registers for variables in three-address instruction `x = y op z`.
    - That is  $R_x$ ,  $R_y$  and  $R_z$  for  $x$ ,  $y$  and  $z$  respectively.
  - Make decisions based on the register descriptor and the address descriptor.

# Code-Generation Algorithm (cont')

---

- Machine instructions for  $x = y + z$ 
  - Call  $\text{getReg}(x = y + z)$  to select registers  $R_x$ ,  $R_y$  and  $R_z$  for  $x$ ,  $y$  and  $z$ .
  - **If**  $y \notin \text{regDesc}(R_y)$ , get some  $y' \in \text{addrDesc}(y)$  and issue an instruction:  $\text{LD } R_y, y'$ .
  - **If**  $z \notin \text{regDesc}(R_z)$ , get some  $z' \in \text{addrDesc}(z)$  and issue an instruction:  $\text{LD } R_z, z'$ .
  - Issue an instruction:  $\text{ADD } R_x, R_y, R_z$ .
  - Adjust register and address descriptors.

# Code-Generation Algorithm (cont')

---

- Machine instructions for  $x = y$ 
  - Call  $\text{getReg}(x = y)$  to select registers  $R_x$  and  $R_y$ .
    - $\text{getReg}()$  will always choose the same register for both  $x$  and  $y$ .
  - **If**  $y \notin \text{regDesc}(R_y)$ , get some  $y' \in \text{addrDesc}(y)$  and issue an instruction:  $\text{LD } R_y, y'$ .
  - Adjust register descriptor:  $\text{regDesc}(R_y) \cup = \{x\}$

# Design of `getReg(...)`

---

- Input:  $x = y + z$
- Output:  $R_x$ ,  $R_y$  and  $R_z$
- Algorithm (use  $R_y$  as an example)
  - **if**  $y$  in some registers, pick  $R_y$  in them.
  - **elseif** there is empty registers, pick one as  $R_y$ .
  - **elseif** { let  $R$  be a candidate and  $R$  holds  $v$ ,  
**foreach**  $v$  check:
    - **if** `addrDesc(v)` has other location,  $R$  is OK.
    - **elseif**  $v == x$  **and**  $x != z$ ,  $R$  is OK.
    - **elseif**  $v$  is not used later,  $R$  is OK.
    - **else** **spill**, i.e. issue [ST v, R](#).
  - choose the  $R$  with minimal spills }.

## 6. Peephole Optimization

---

- A simple but effective technique for locally code improvement
  - Examine a sliding window (peephole) of target instructions.
  - Replace instruction sequence within the peephole by a shorter or faster sequence.

# Eliminating Redundant Loads and Stores

---

## ○ Examples

- LD R0, a  
ST a, R0  
// eliminated

# Eliminating Unreachable Code

---

## ○ Examples

- if debug == 1 goto L1  
goto L2  
L1: print debugging information  
L2:  
if debug != 1 goto L2  
print debugging information  
L2:



# Flow-of-Control Optimizations

## ○ Examples

- goto L1  
...  
L1: goto L2  
goto L2  
...  
L1: goto L2
- if a < b goto L1  
...  
L1: goto L2  
if a < b goto L2  
...  
L1: goto L2

Recall the translation scheme of flow-of-control statements !

# Algebraic Simplification and Reduction in Strength

---

## ○ Examples

- $x = x + 0$   
// eliminated
- $x = x * 1$   
// eliminated
- $y = x * 2$   
 $y = x \ll 1$
- $y = x * 4$   
 $y = x \ll 2$

# Enjoy the Course!

---

