CS 4240: Compilers

Lecture 13: Instruction Scheduling

Instructor: Vivek Sarkar
(vsarkar@gatech.edu)

February 25, 2019

Worksheet-11 Solution

(From Lecture given on 02/18/2019)

- Consider the following sequence of IR instructions:

```
1. t1 := i × 4
2. t2 := a + t1
3. t3 := *t2          // Load instruction
4. t4 := b + t1
5. *t4 := t3          // Store instruction
```

- Show the output of instruction selection with the minimal total cost, using any approach you like (need not be an algorithm presented in class). The output should be a sequence of machine instructions. You can generate instructions that refer to virtual IR registers such as i, a, b, and t1…t4.

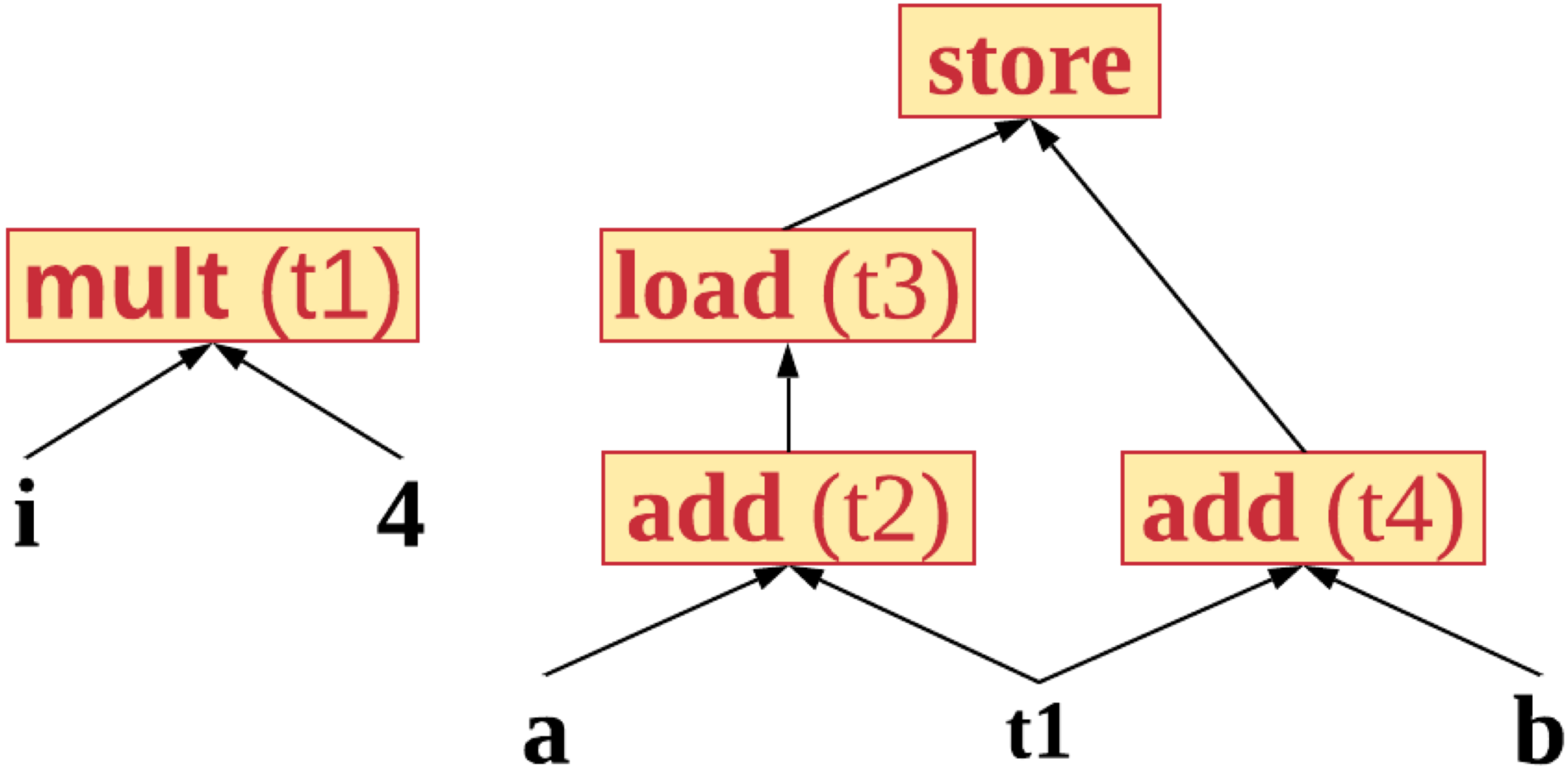
The ISA that we are targeting includes the following instructions, with costs:

- **add r2, r1** r1 ← r1 + r2 Cost = 2
- **addi c, r1** r1 ← r1 + c Cost = 2
- **muli c, r1** r1 ← r1 × c Cost = 4
- **lshi c, r1** r1← r1 << c // Left shift r1's content by c bits.
  // Each shift equals multiplication by 2 Cost = 2
- **lw r2, r1** r1 ← *r2 // Load a word from address contained in r2 to r1 Cost = 6
- **sw r2, r1** *r1 ← r2 // Store a word in r2 into address contained in r1 Cost = 4
- **movmw r2, r1** *r1 ← *r2 // Copy a word from address in r2 to that in r1 Cost = 8

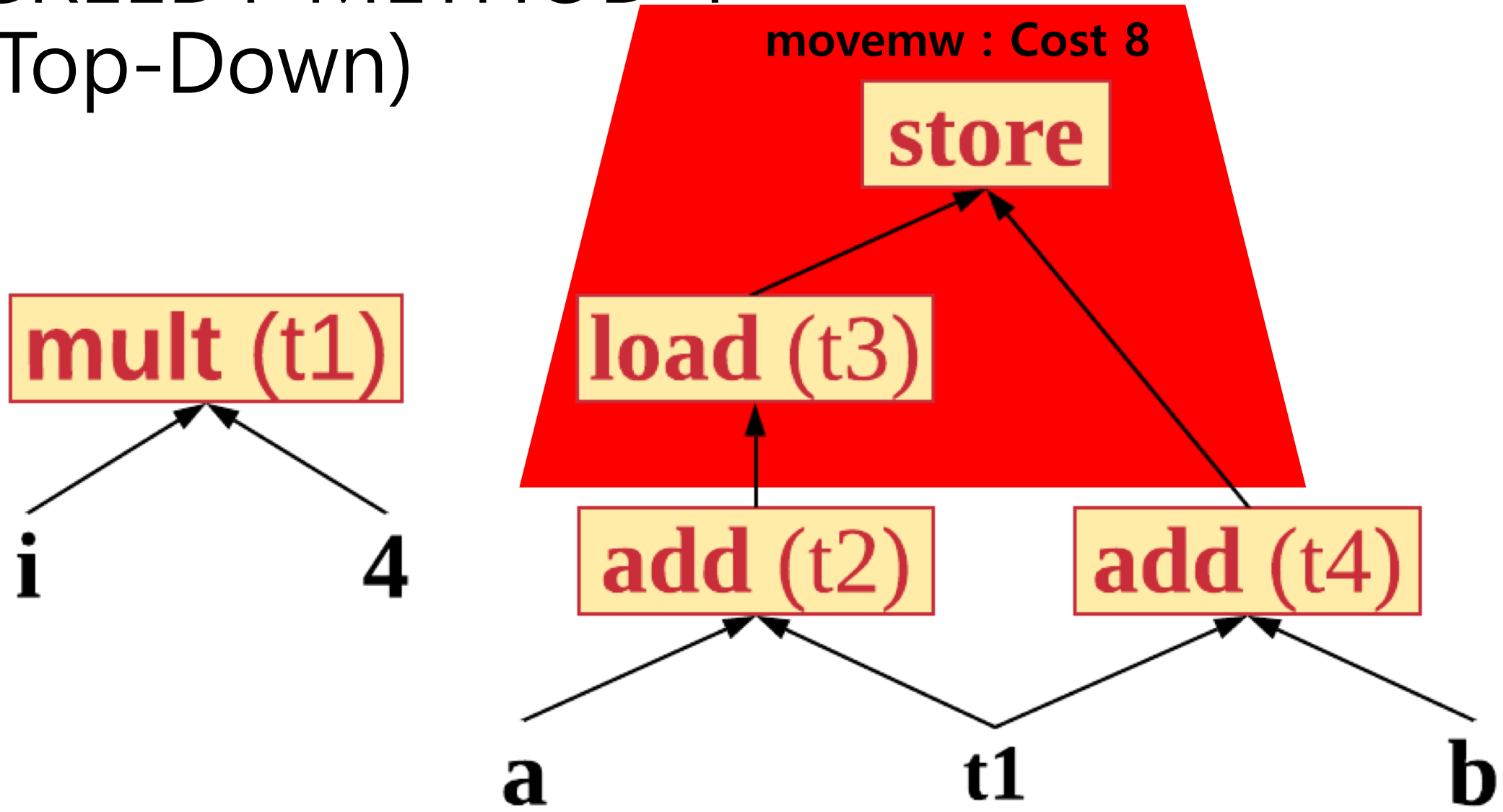
Most naïve solution : Generate machine instructions for each IR instruction

TOTAL COST : 16

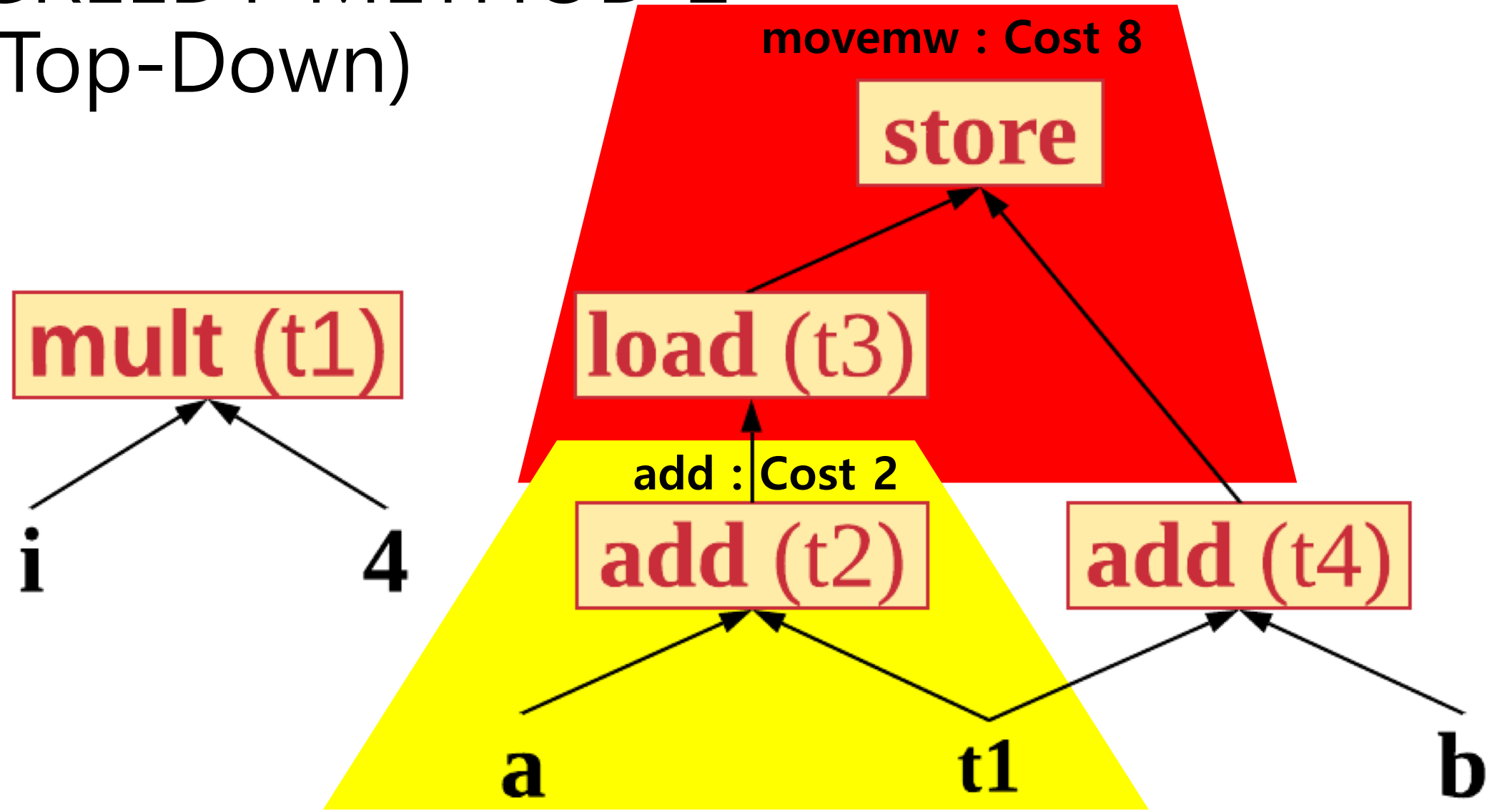
Dependence Tree for the given IR instructions



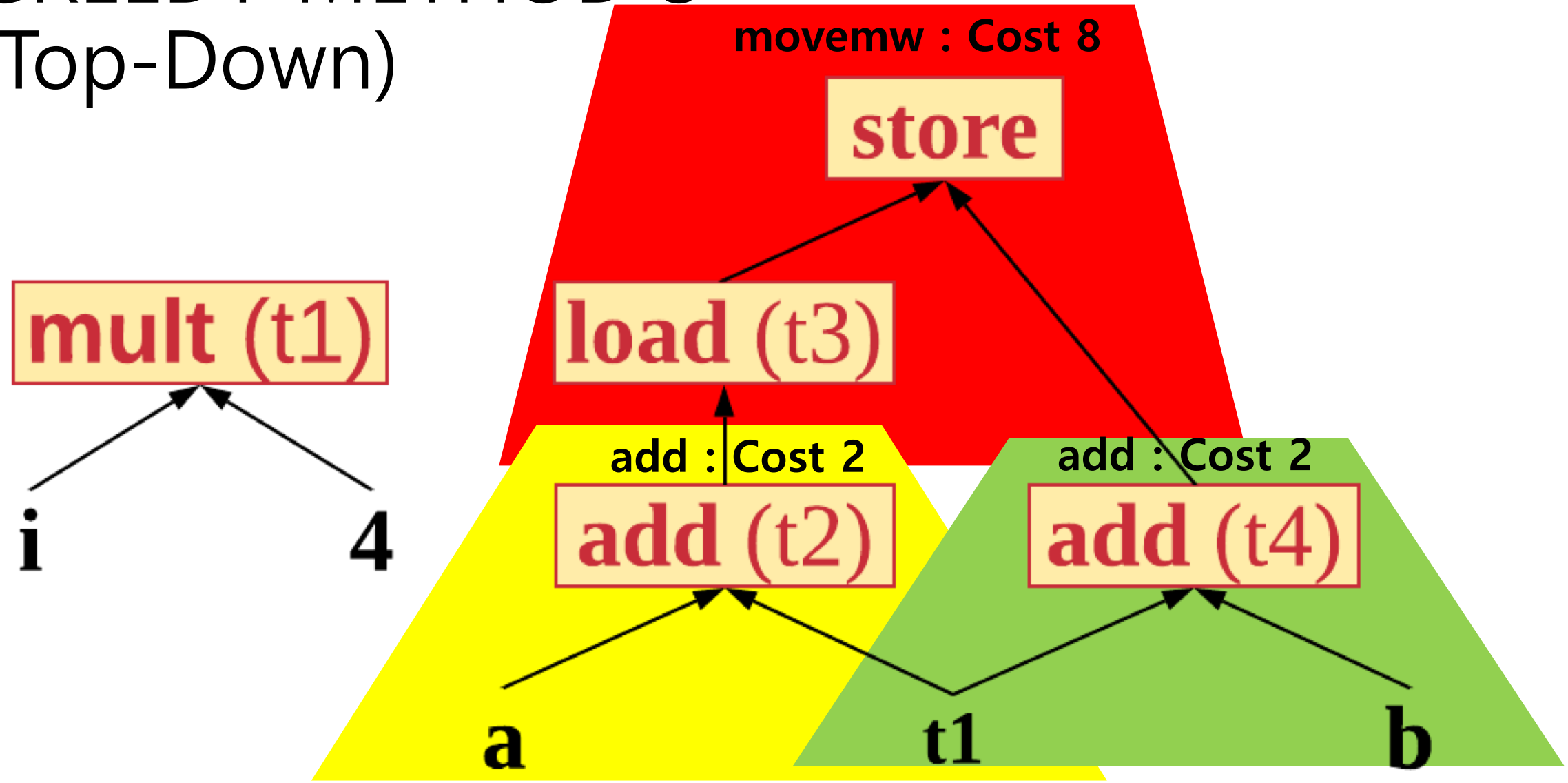
GREEDY METHOD 1 (Top-Down)



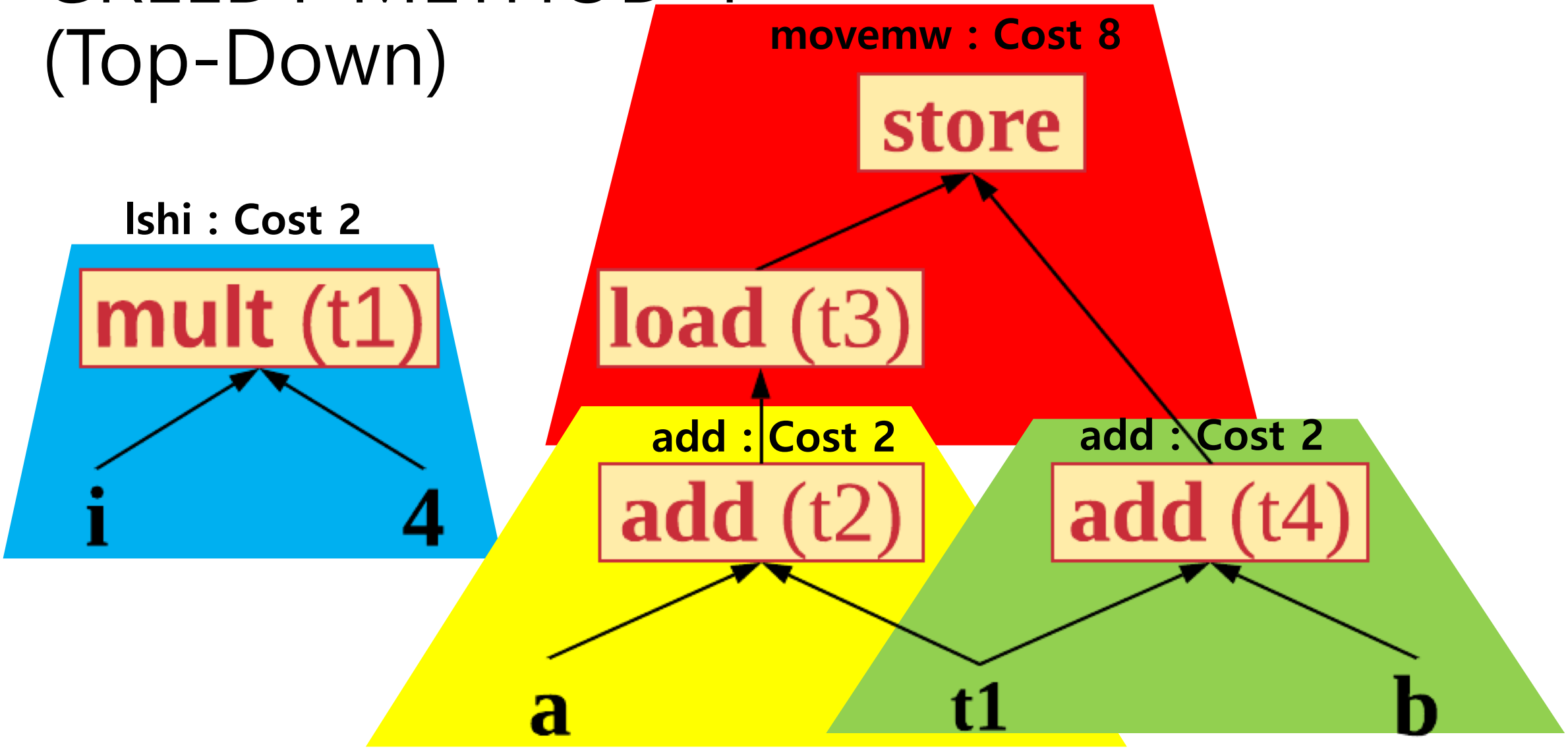
GREEDY METHOD 2 (Top-Down)



GREEDY METHOD 3 (Top-Down)



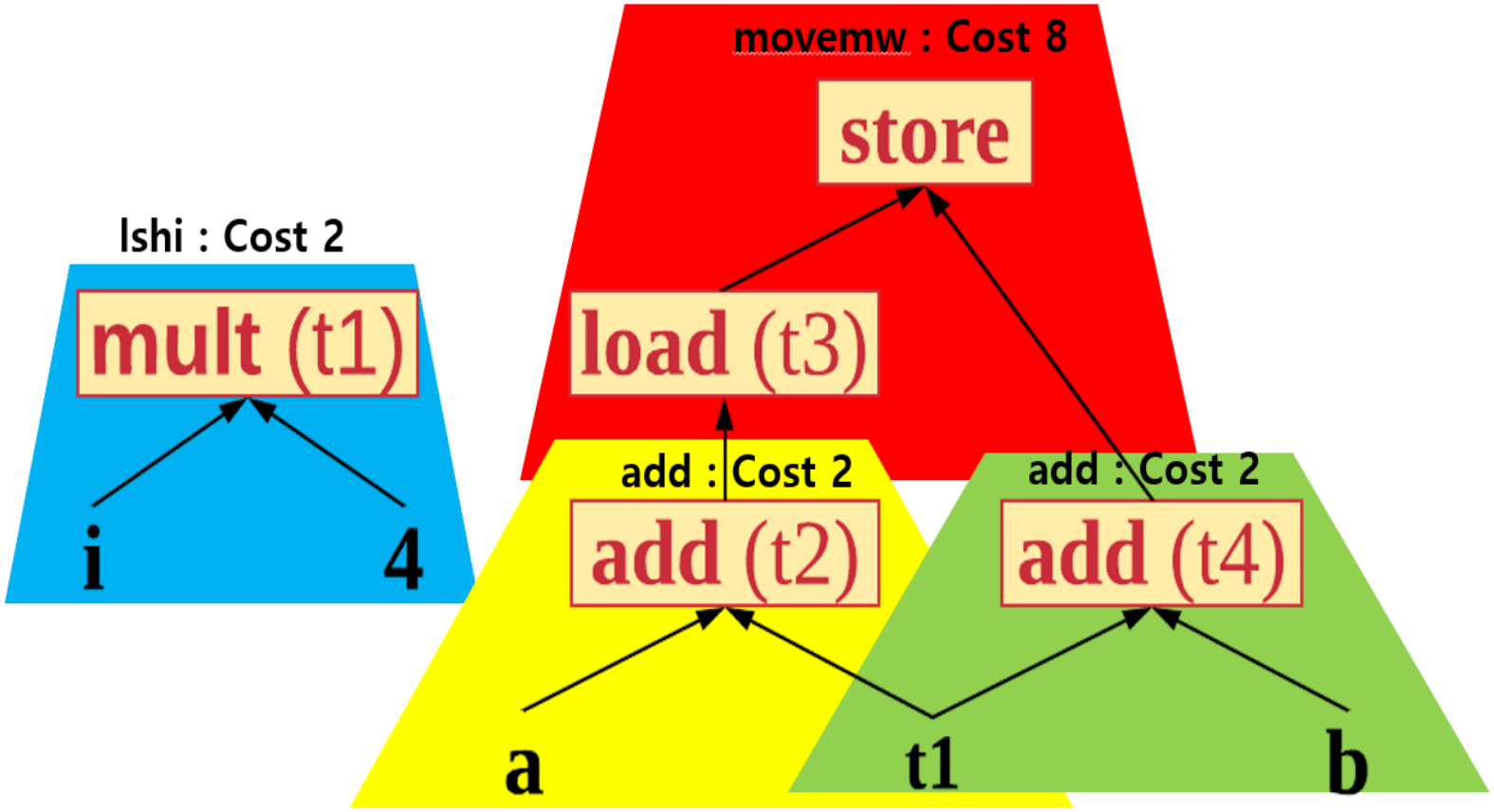
GREEDY METHOD 4 (Top-Down)



GREEDY METHOD 5 (Top-Down)

- All nodes are covered with tiles.

1. lshi 2, i cost 2
2. add i, a cost 2
3. add i, b cost 2
4. movmw a, b cost 8

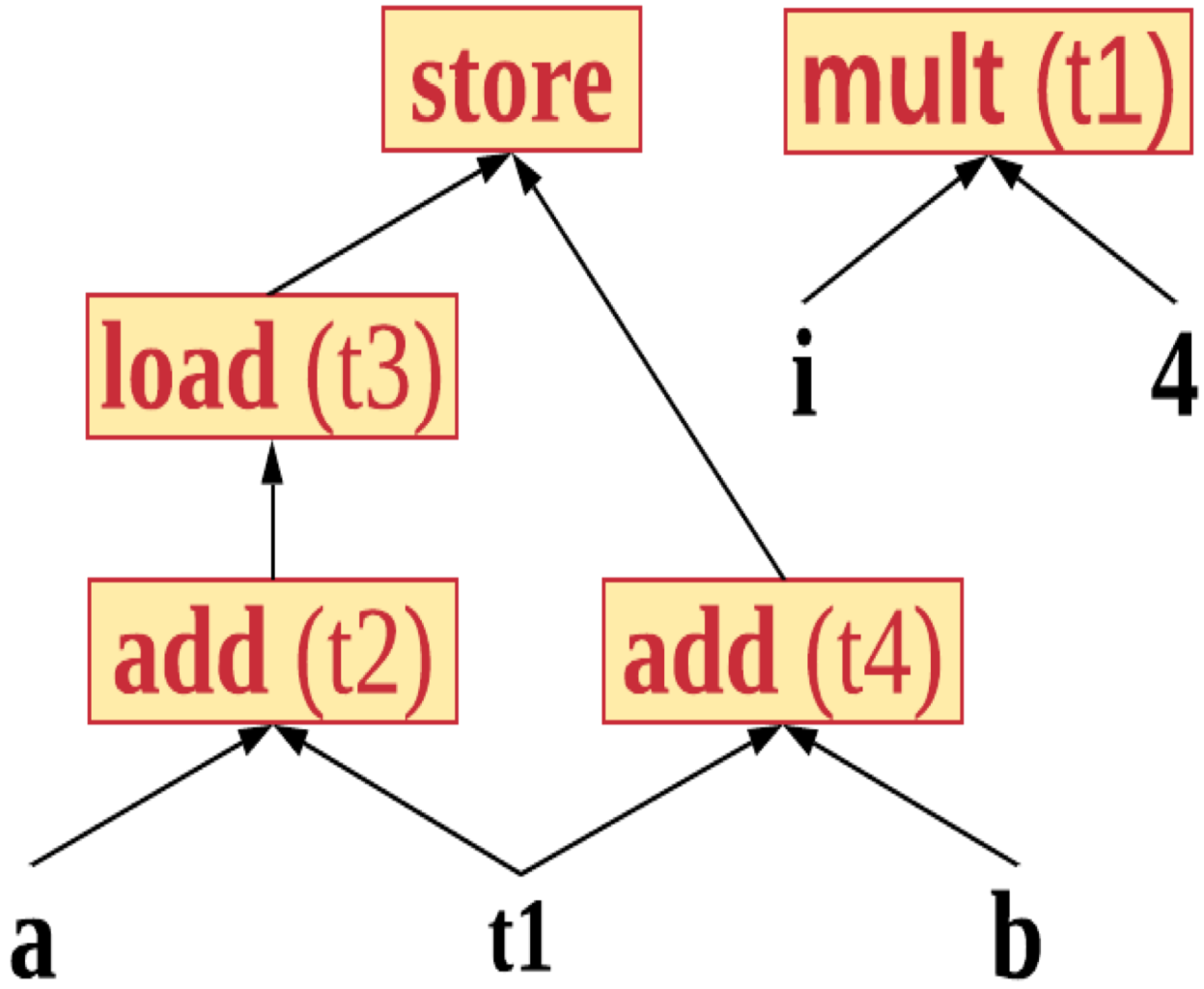


- **Total COST : 14**
  Cheaper than naïve solution!

Dynamic Programming 1 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

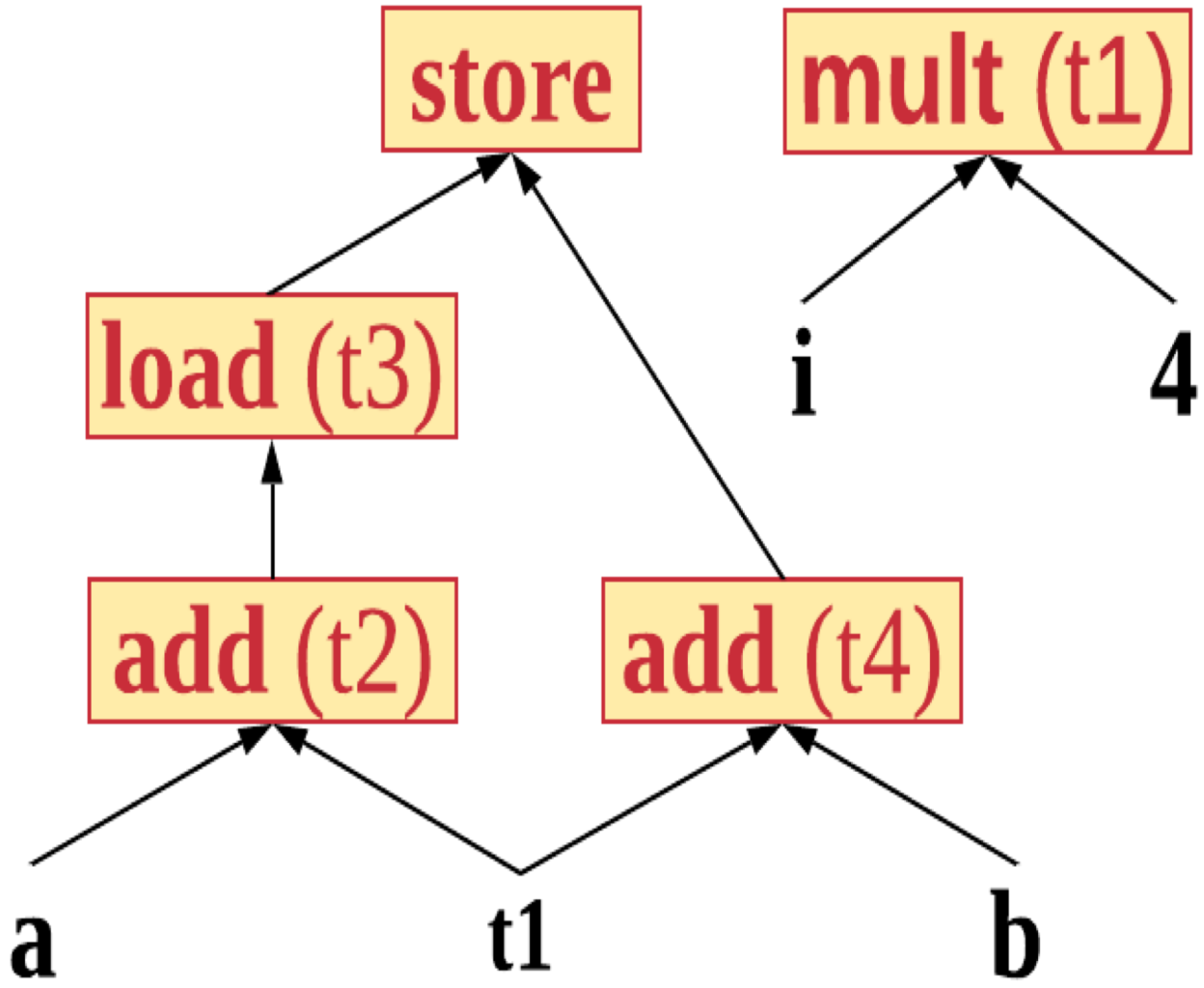
Node	Best Tile	BestCost



Dynamic Programming 2 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

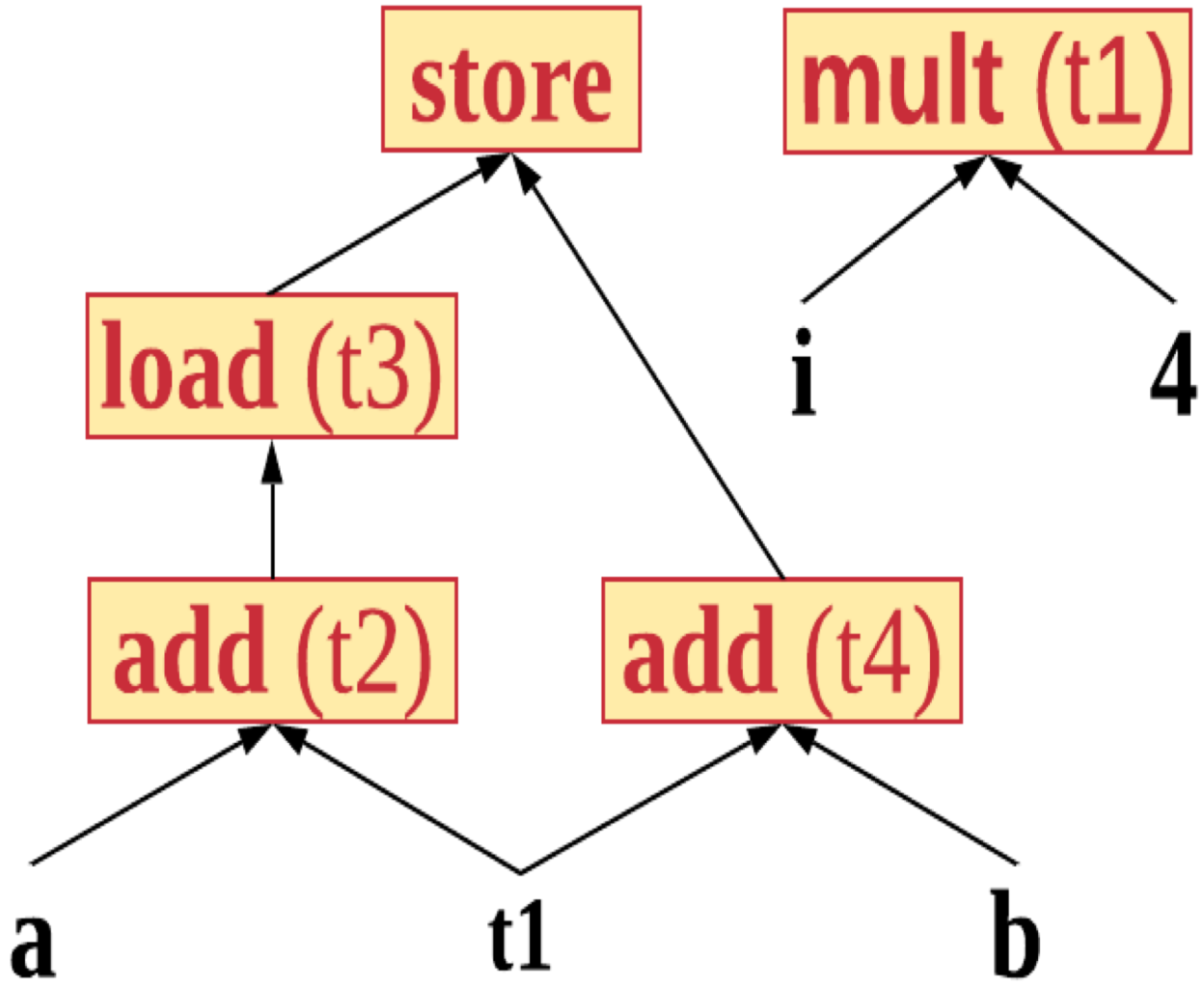
Node	Best Tile	BestCost
Add(t2)	add	2 (add)



Dynamic Programming 3 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

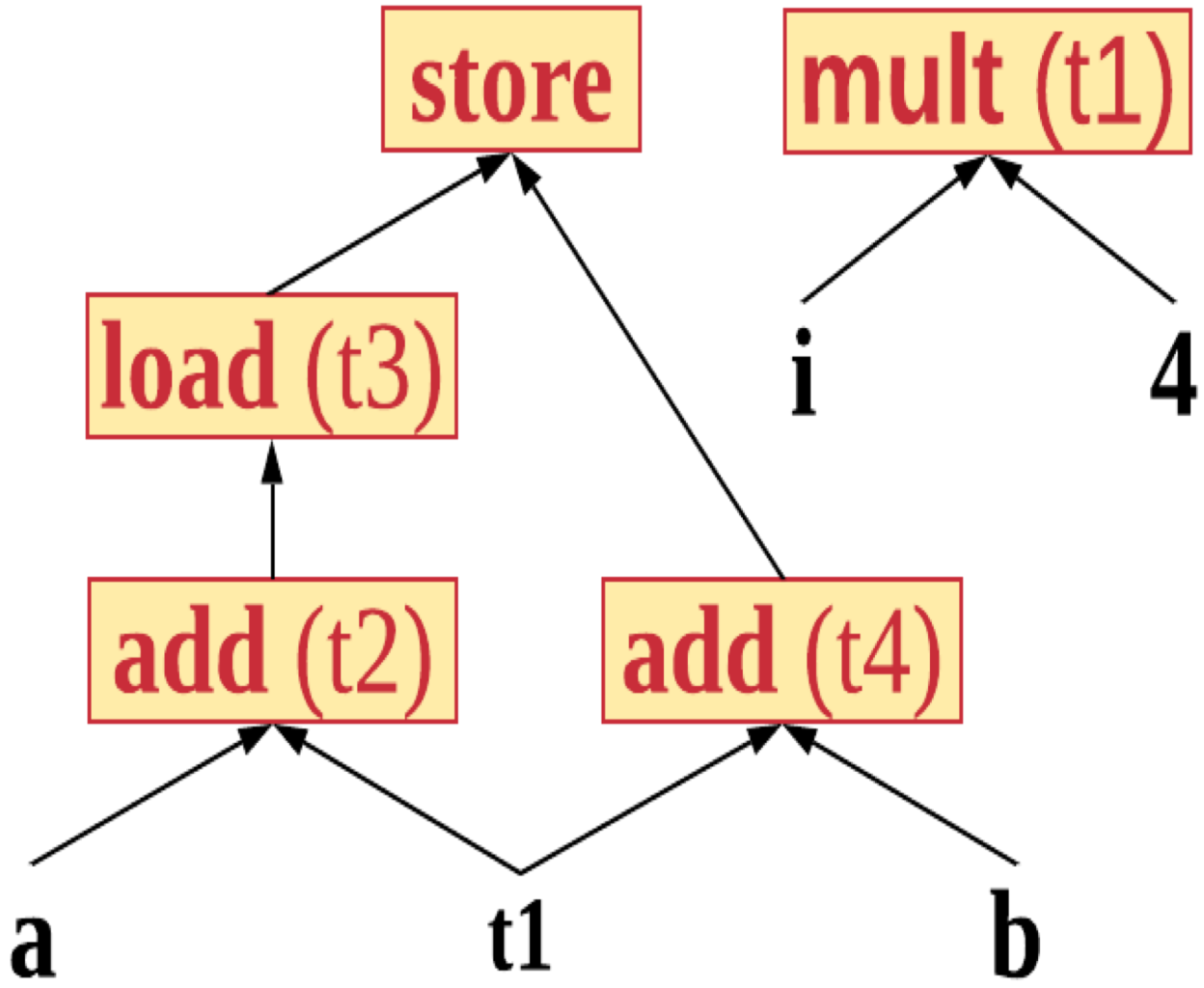
Node	Best Tile	BestCost
Add(t2)	add	2 (add)
Add(t4)	add	2 (add)



Dynamic Programming 4 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

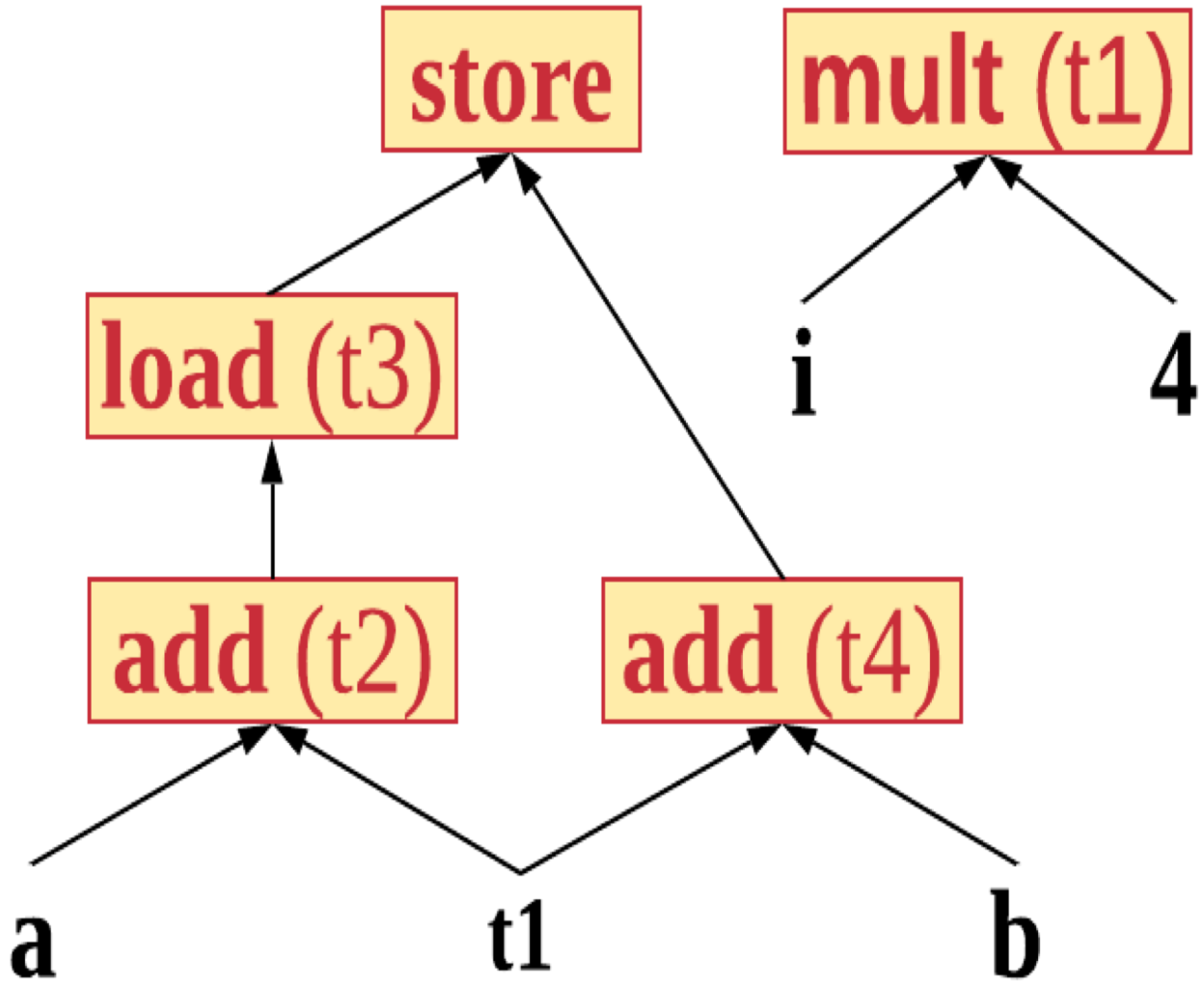
Node	Best Tile	BestCost
Add(t2)	add	2 (add)
Add(t4)	add	2 (add)
Load(t3)	lw	6 (lw) + 2 (t2)



Dynamic Programming 5 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

Node	Best Tile	BestCost
Add(t2)	add	2 (add)
Add(t4)	add	2 (add)
Load(t3)	lw	6 (lw) + 2 (t2)
Store (root)	**movmw**	**8 (movmw)** **+ 2 (t2)** **+ 2 (t4)**



Dynamic Programming 6 (Bottom-Up)

Node	Best Tile	BestCost
mult(t1)	lshi	2 (lshi)

Node	Best Tile	BestCost
Add(t2)	add	2 (add)
Add(t4)	add	2 (add)
Load(t3)	lw	6 (lw) + 2 (t2)
Store (root)	**movmw**	**8 (movmw)** **+ 2 (t2)** **+ 2 (t4)**

Resulting cost of instruction selection : **14**

Instruction Selection Result

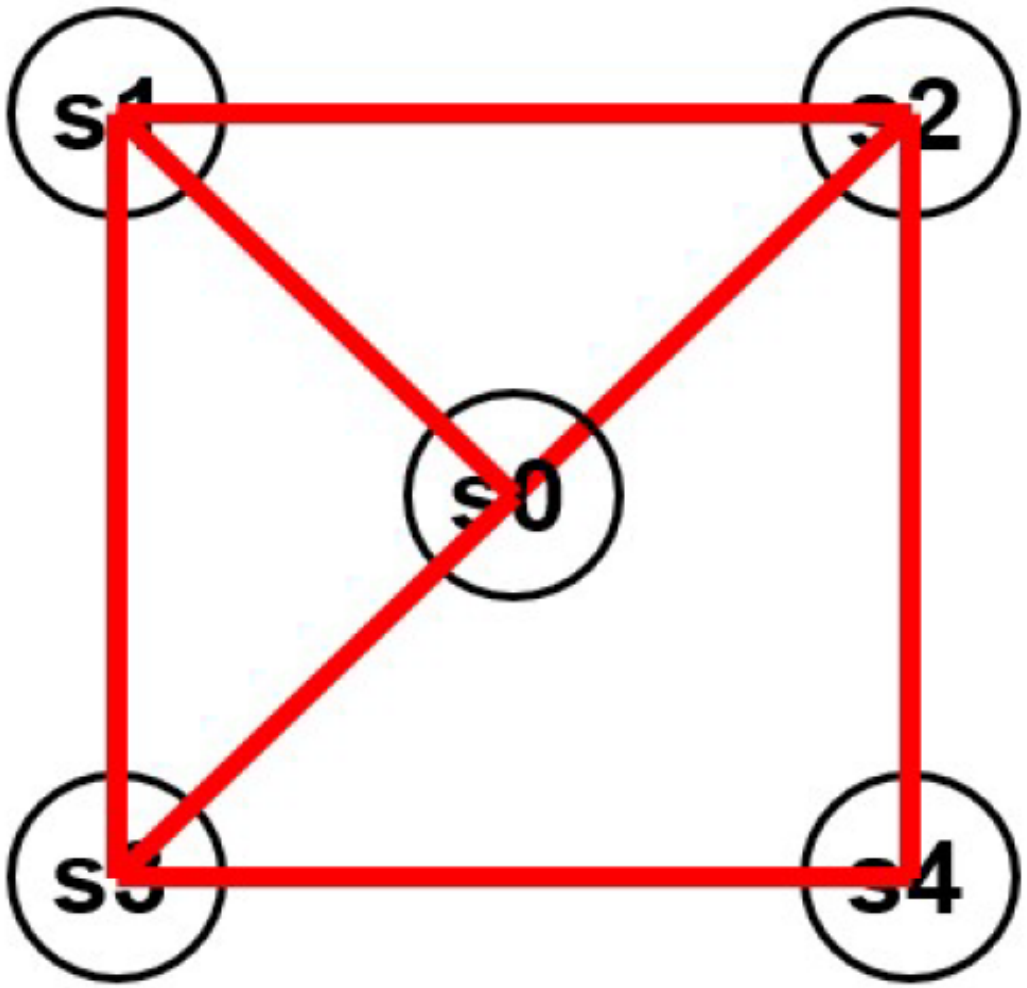
	Cost
4. Movmw a, b	**8**
3. Add i, b	**2**
2. Add i, a	**2**
1. Lshi 2, i	**2**

- In the worksheet's example,
  Greedy method gives the same result as the optimal Dynamic Programming method.
- However,
  greedy method might not be optimal in other cases.

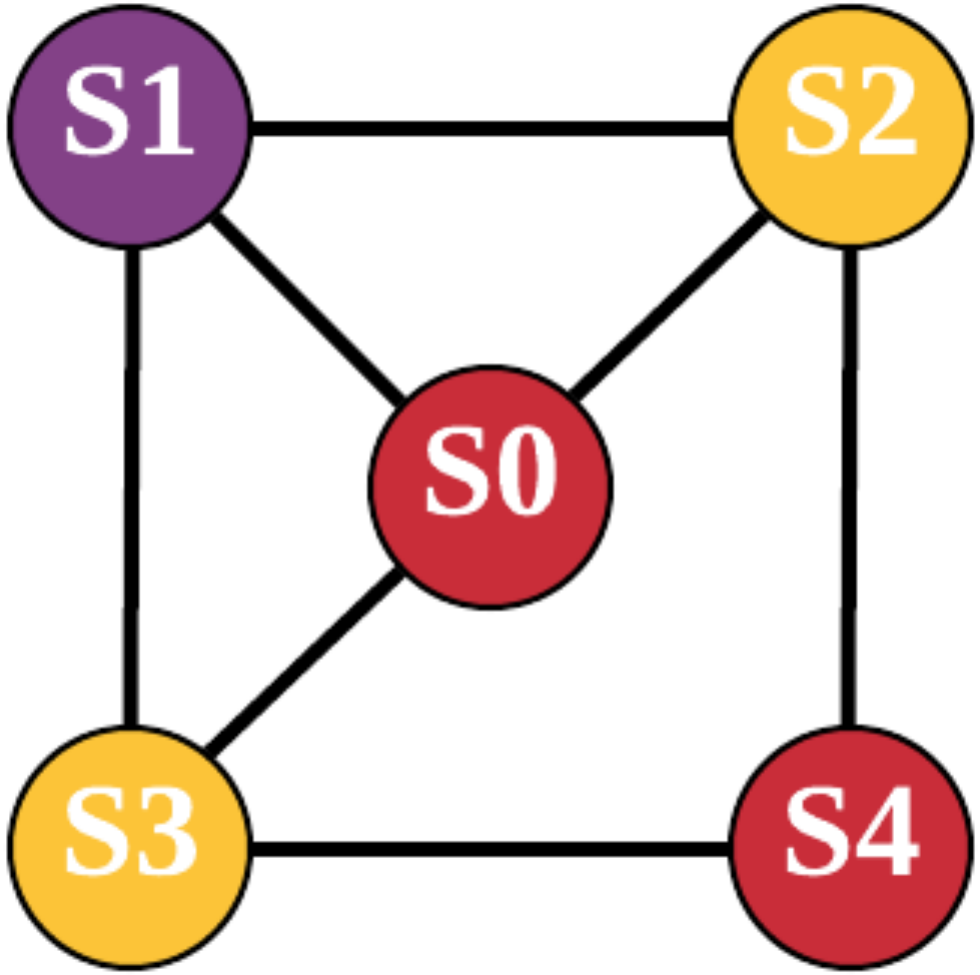
Worksheet-12 Solution

(From Lecture given on 02/20/2019)

Color the interference graph below with the minimum number of colors. Indicate if this coloring can be obtained using the Chaitin or Chaitin-Briggs algorithms studied in class.

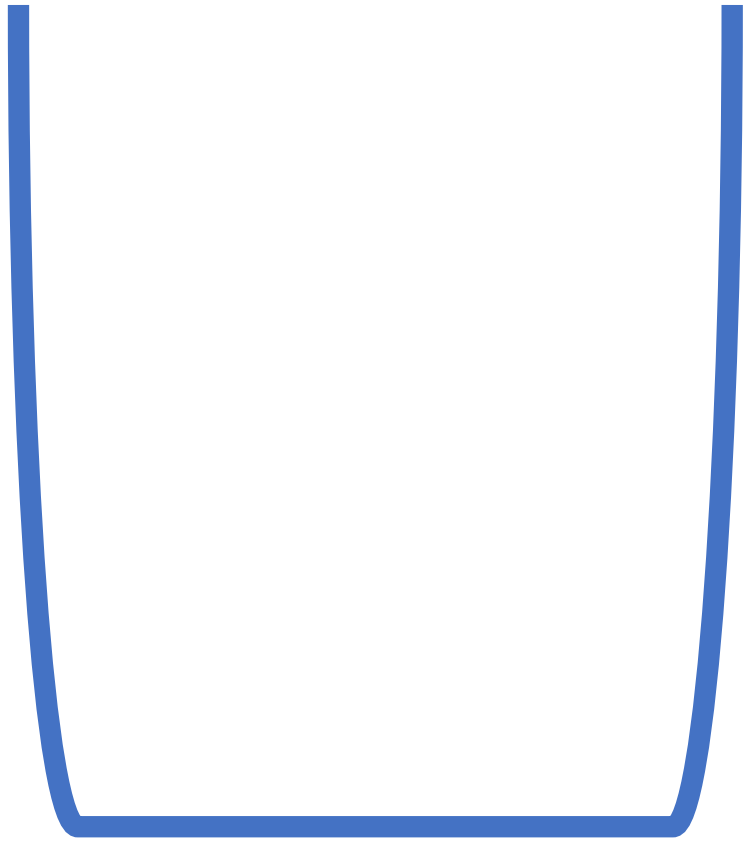


Minimum number of colors needed = **3**



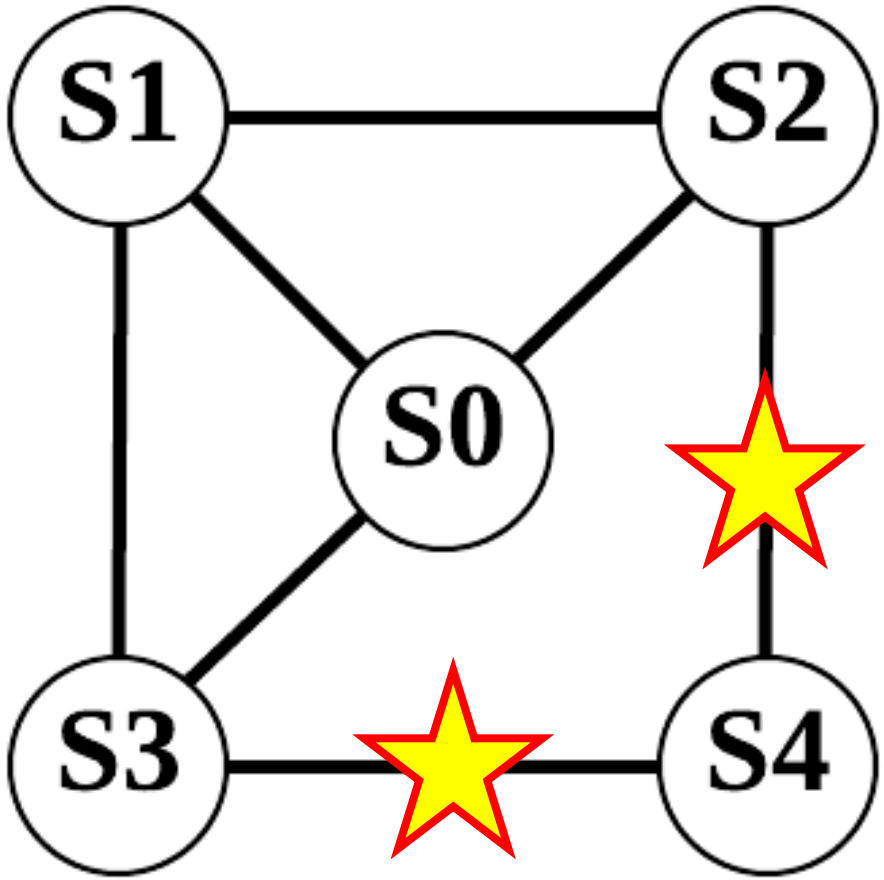
Now,
let's set **K = 3** and see if Chaitin's algorithm or Chaitin-Briggs algorithm can color the graph without any spills.

Chaitin's algorithm – 1 (pushing stage)

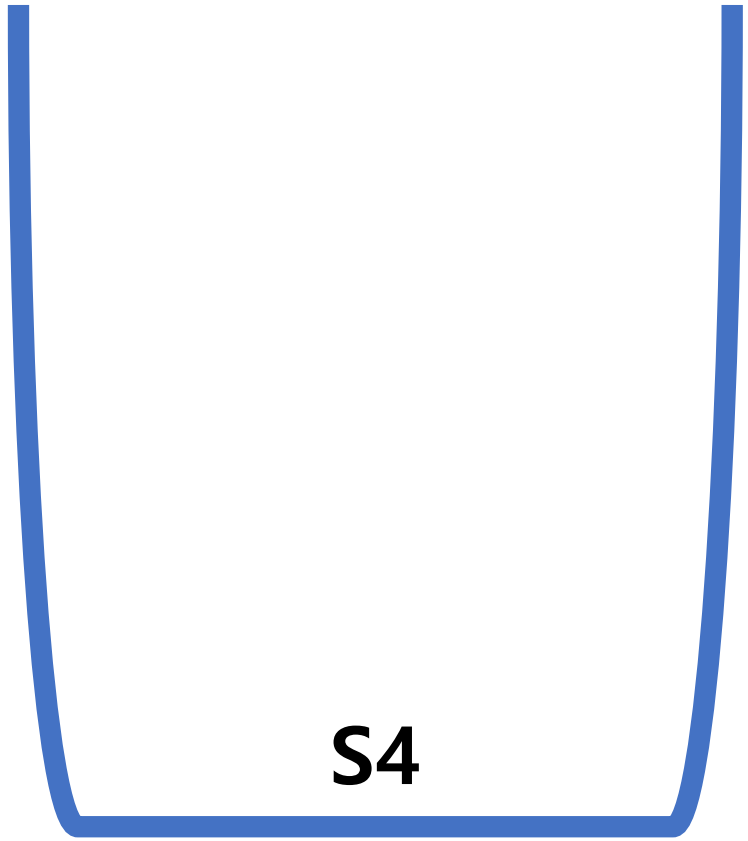


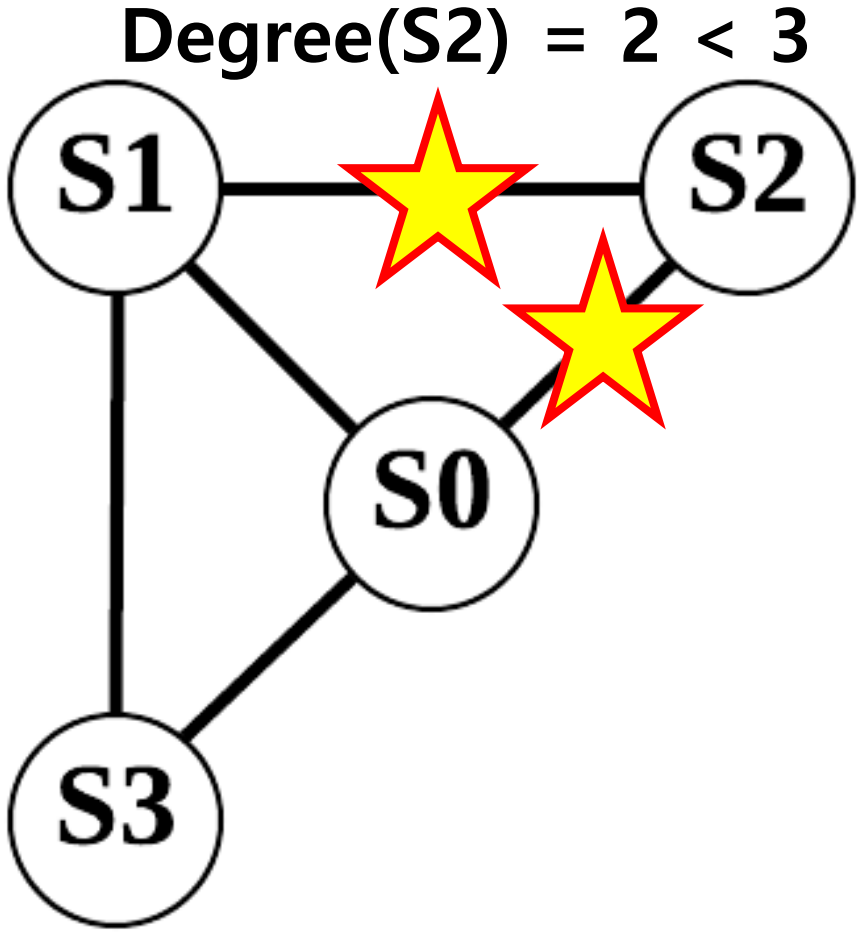
STACK

Degree(S4) = 2 < 3

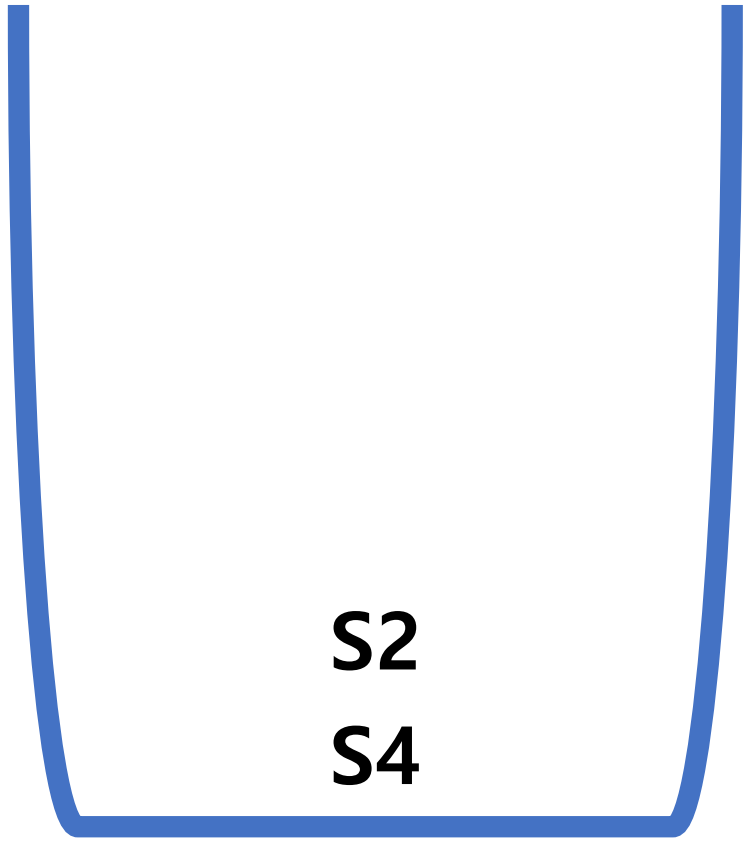


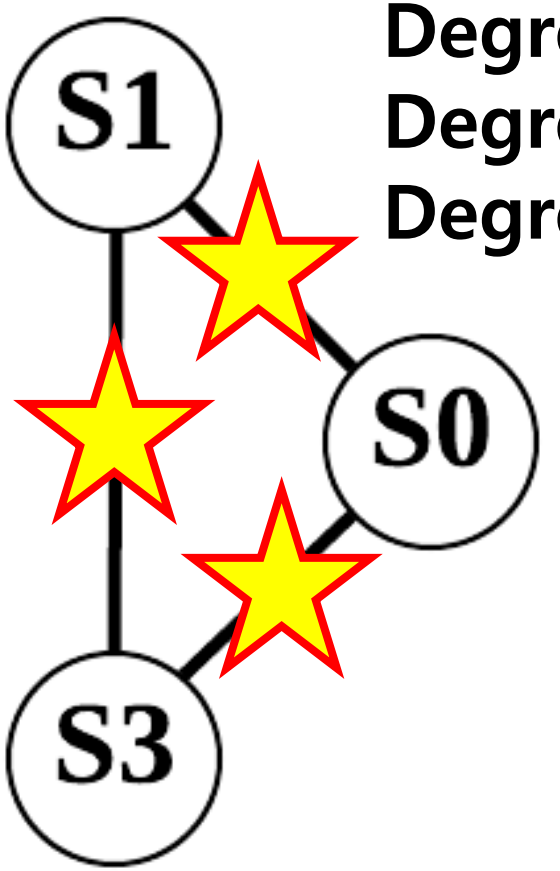
Chaitin's algorithm – 2 (pushing stage)





Chaitin's algorithm – 3 (pushing stage)

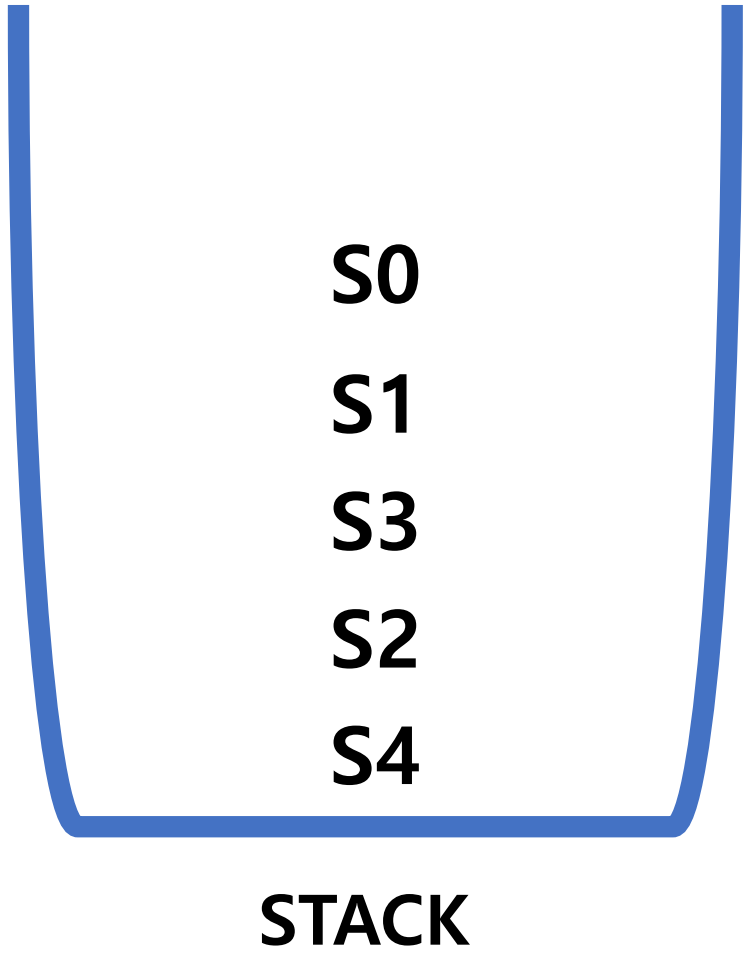




Degree(S0) = 2 < 3
Degree(S1) = 2 < 3
Degree(S3) = 2 < 3

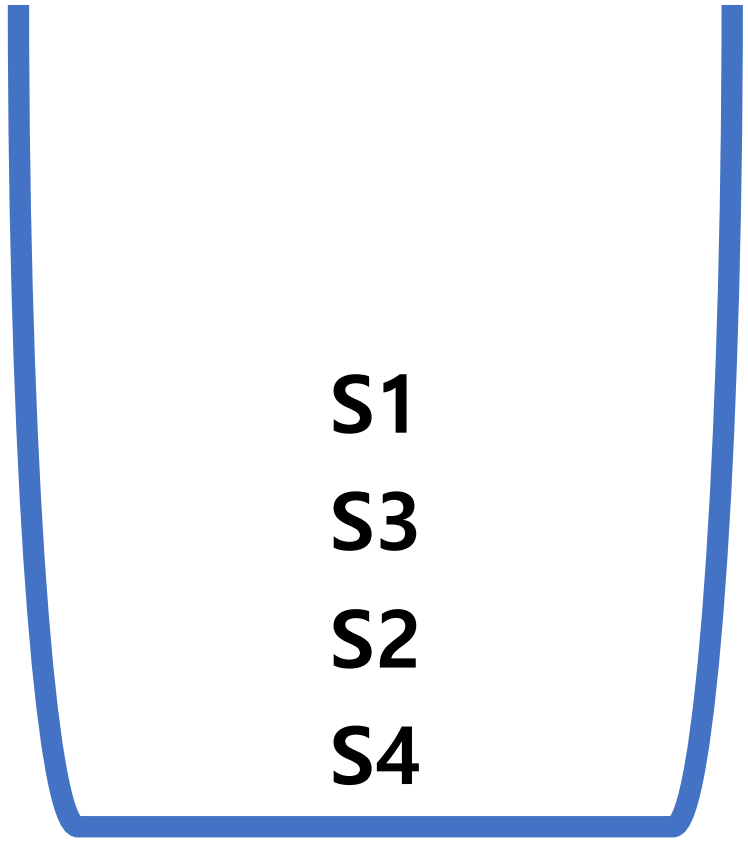
Chaitin's algorithm – 4 (pushing done)

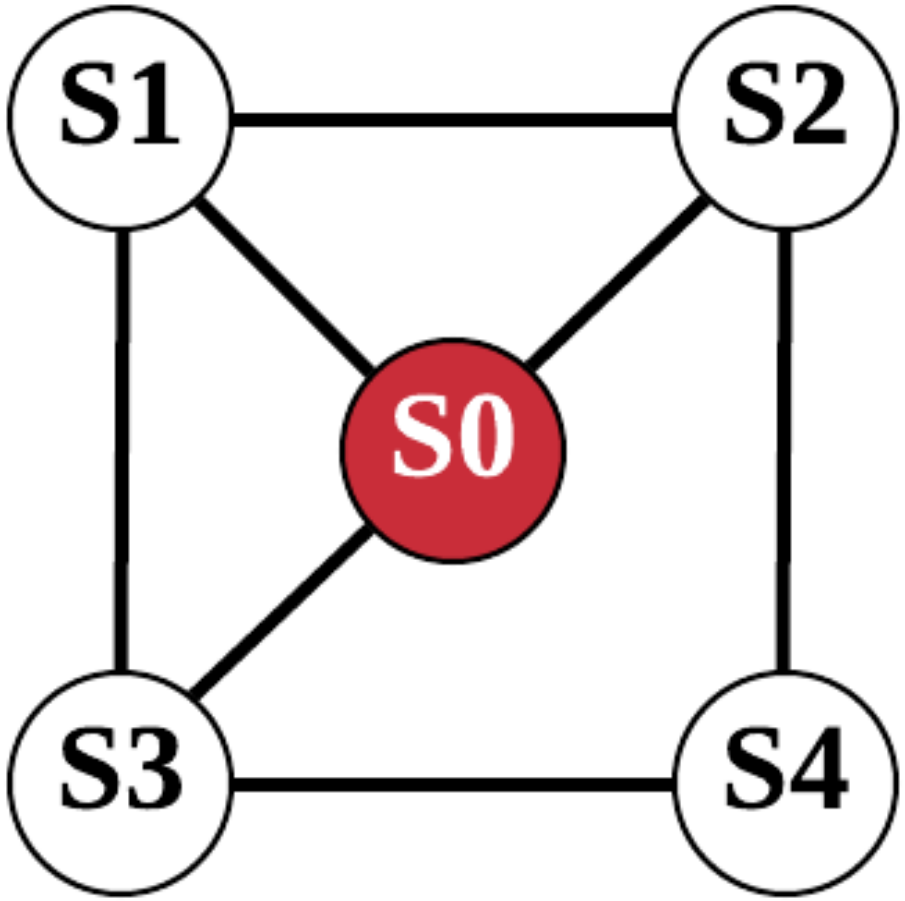
3 COLORS :



Chaitin's algorithm – 5 (popping stage)

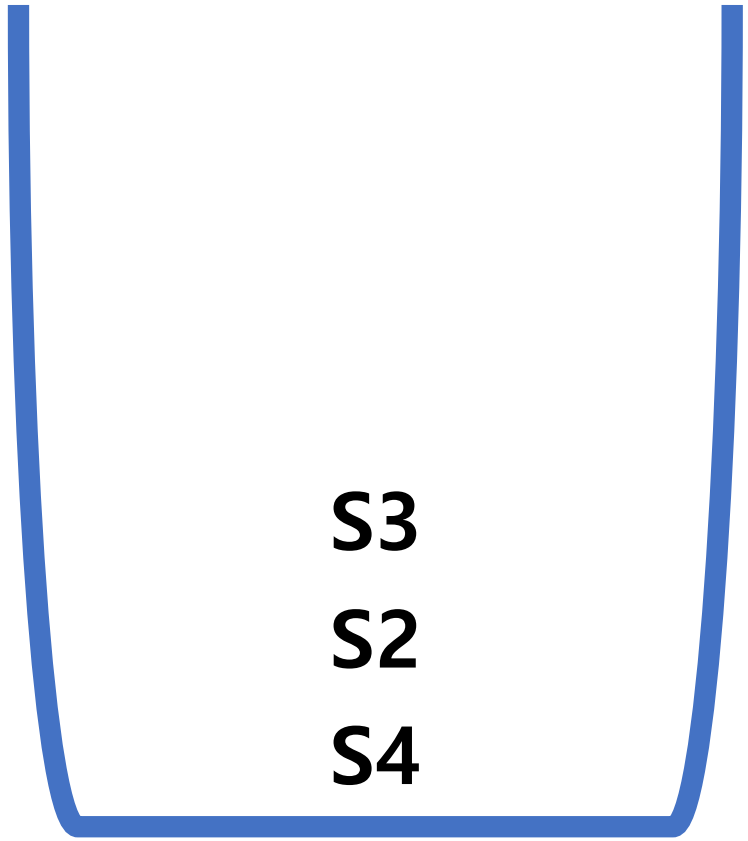
3 COLORS :

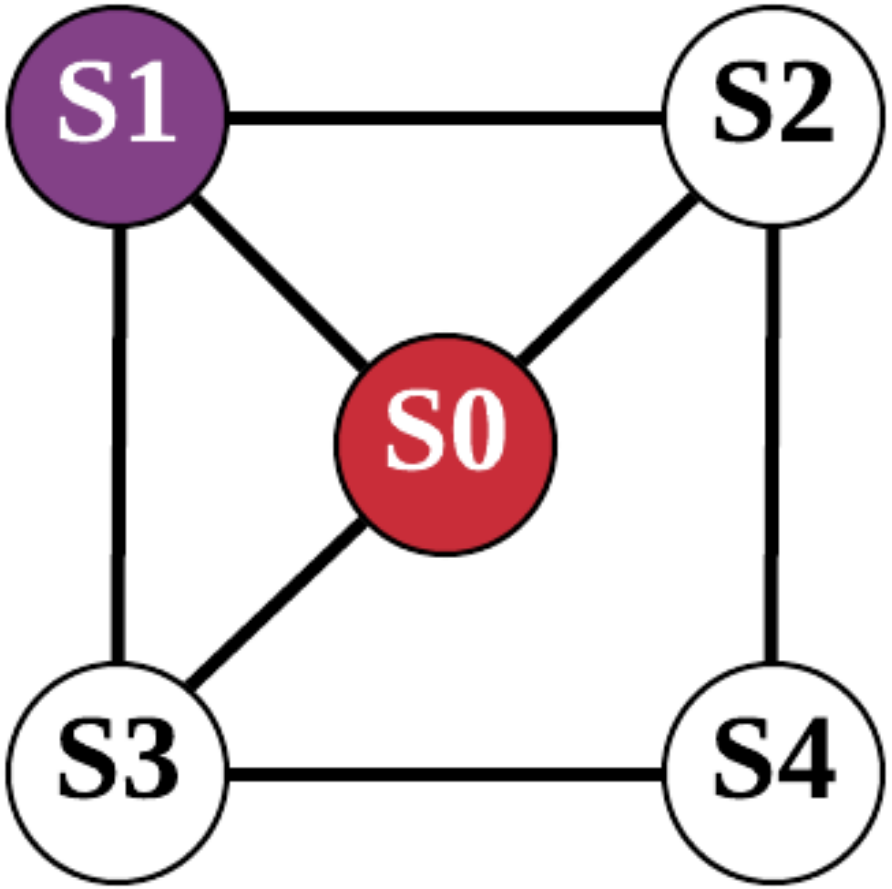




Chaitin's algorithm – 6 (popping stage)

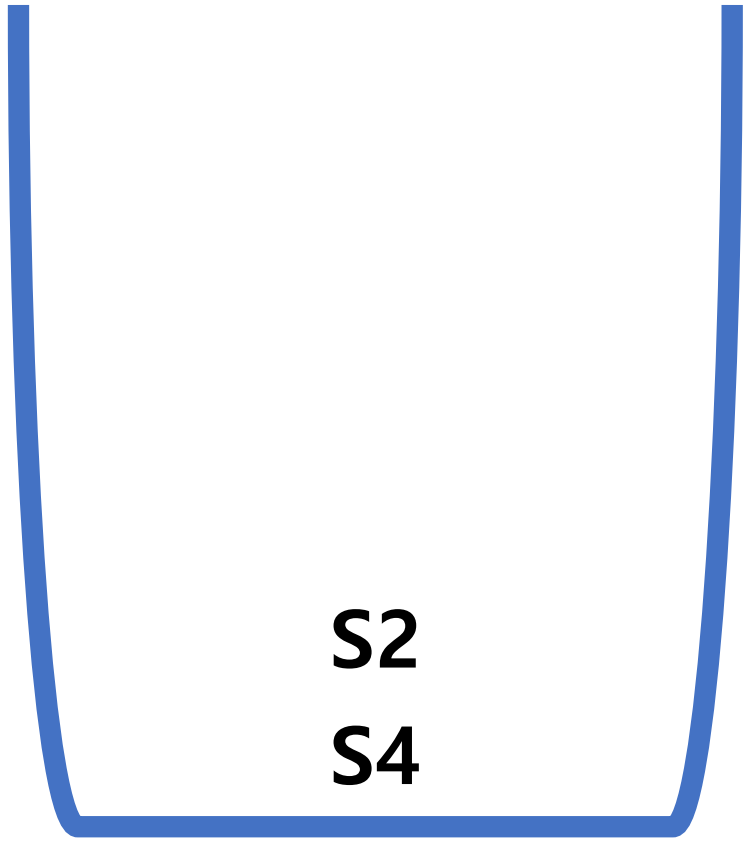
3 COLORS :

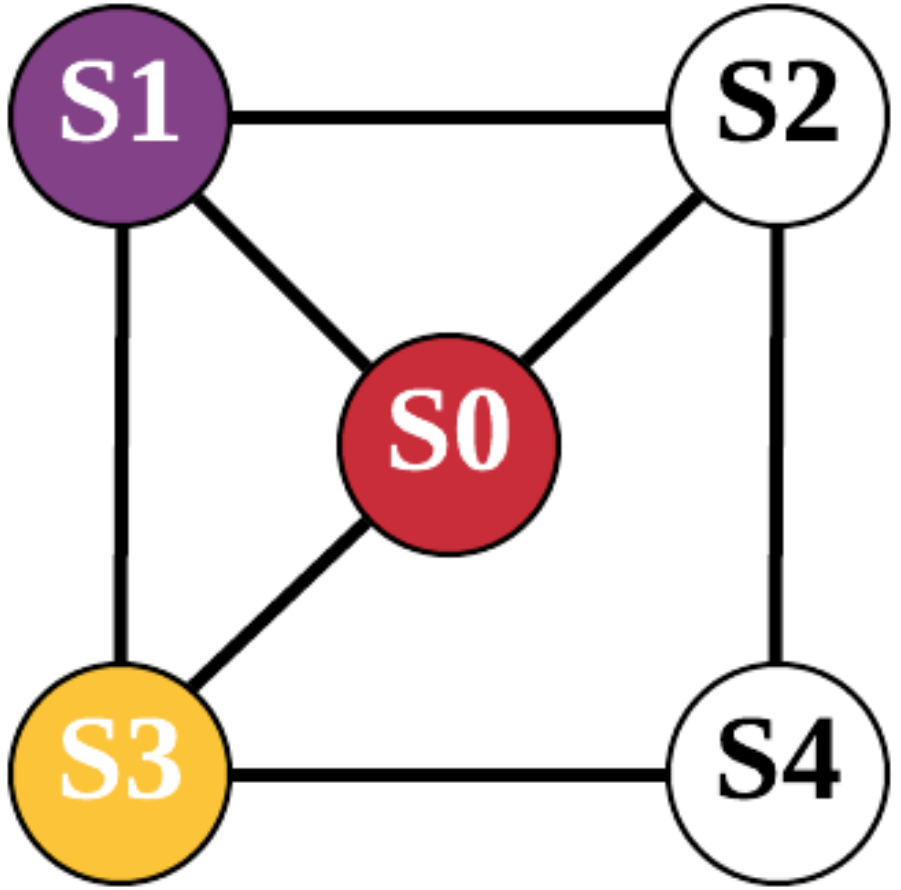




Chaitin's algorithm – 7 (popping stage)

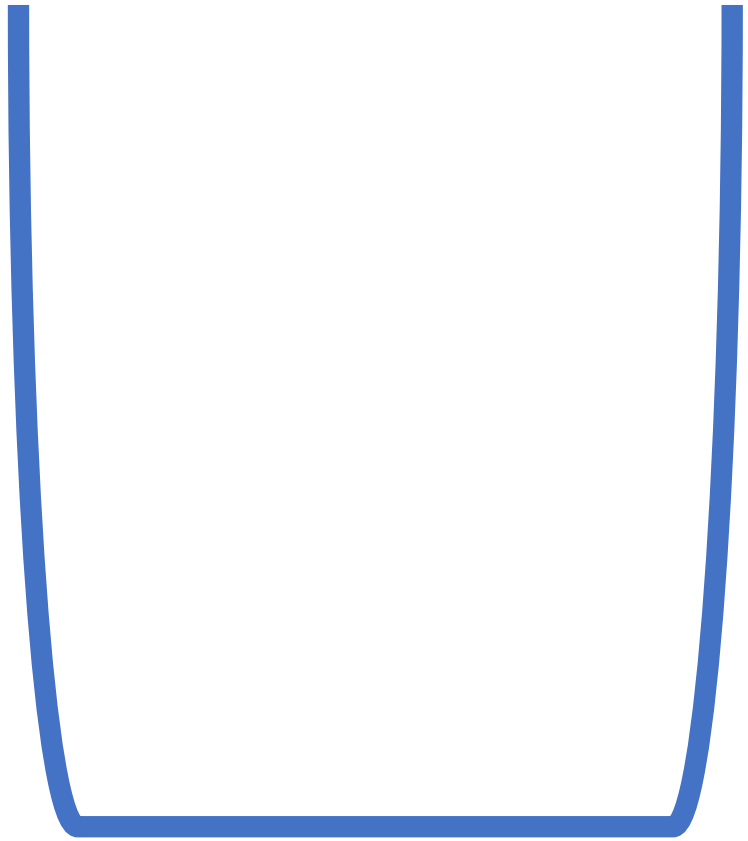
3 COLORS :





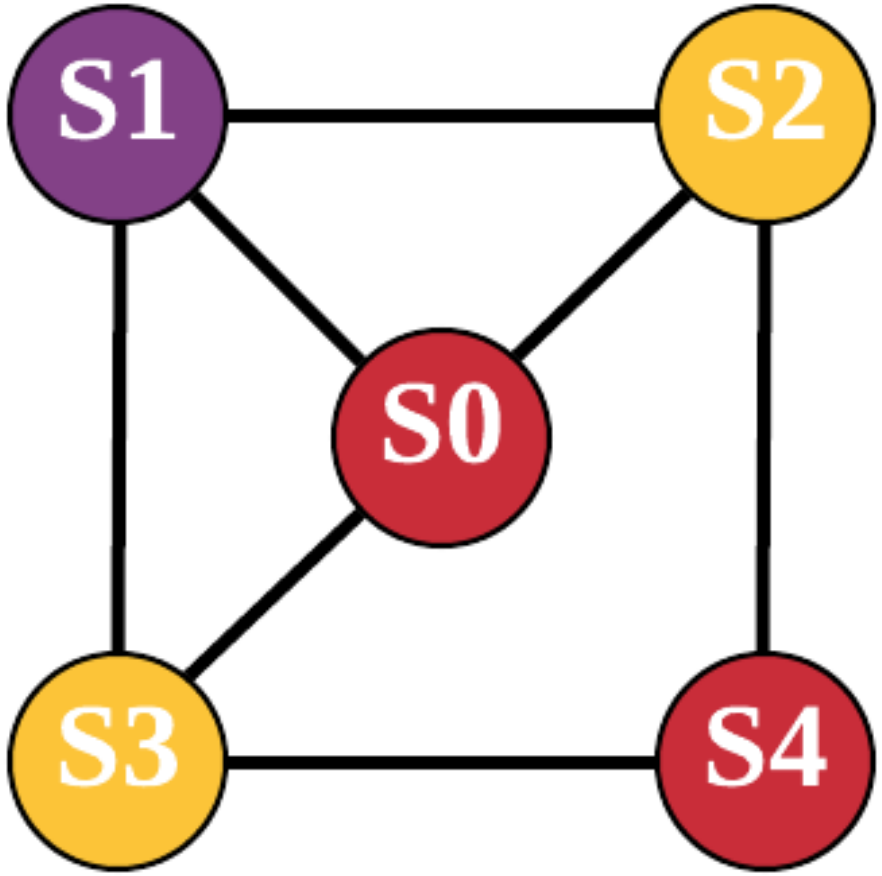
Chaitin's algorithm – 8 (popping done)

3 COLORS :



STACK

3-COLORABLE! (No Spills)



Questions from students

- The worksheet didn't provide the number of registers available for use. Is it possible to solve the worksheet without it?
- How should I set **K** when using Chaitin's or Chaitin-Briggs algorithm? Do I always have to set K to the number of registers available to my machine?
- Can dead-code elimination be done during the register allocation stage?

Chaitin vs **Chaitin-Briggs**

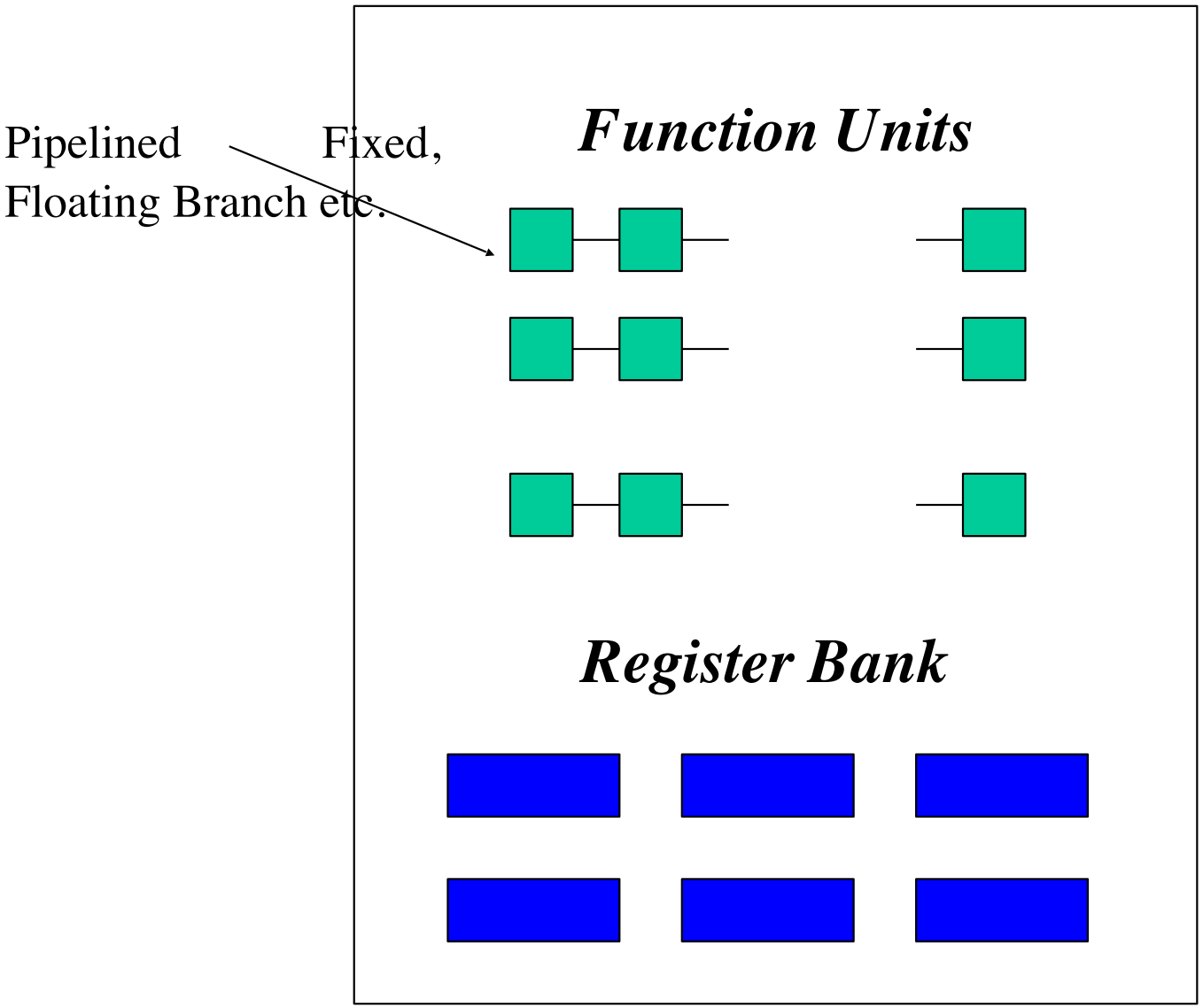
- During the pushing stage,
  when all nodes in the interference graph have Degree bigger than **K,**

- **Chaitin**
  immediately spills one of the nodes in the graph with the smallest spill-cost.
- **Chaitin-Briggs**
  simply pushes the node with the smallest spill-cost to the stack, without spilling. Spilling is done later, only if there is no available color left to color the popped node during the popping stage.

- For the worksheet's interference graph, there is always a node with degree less than K(=3) during the pushing stage.

  Both algorithms (Chaitin, Chaitin-Briggs) can identify that the given interference graph is 3-colorable.

Superscalar (RISC) Processors



Opportunity in Superscalars

- High degree of *Instruction Level Parallelism (ILP)* via multiple (possibly) *pipelined functional units (FUs).*

  Essential to harness promised performance.

- Clean simple model and Instruction Set makes *compile-time* optimizations feasible.
- Therefore, performance advantages can be harnessed automatically

Instruction Scheduling: The Optimization Goal

Given a source program P:

schedule the operations so as to minimize the overall execution time on the functional units in the target machine.

Alternatives for Embedded Systems:

- Minimize the amount of power consumed by functional units during execution of the program.
- Ensure operations are executed within given time constraints.

Cost Functions

- Effectiveness of the Optimizations: *How well can we optimize our objective function?*

  Impact on running time of the *compiled* code determined by the completion time.

- Efficiency of the optimization: *How fast can we optimize?*

  Impact on the time it takes to compile or cost for gaining the benefit of code with fast running time.

Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.

Data Dependence: Definition

A data dependence, $S_1 \rightarrow S_2$, exists between CFG nodes S_1 and S_2 with respect to variable X if and only if

1. there exists a path P: $S_1 \rightarrow S_2$ in *CFG*, with no intervening write to X, and

2. at least one of the following is true:

   (a) **(flow)** X is written by S_1 and later read by S_2, or

   (b) **(anti)** X is read by S_1 and later is written by S_2 or

   (c) **(output)** X is written by S_1 and later written by S_2

Impact of Control Flow

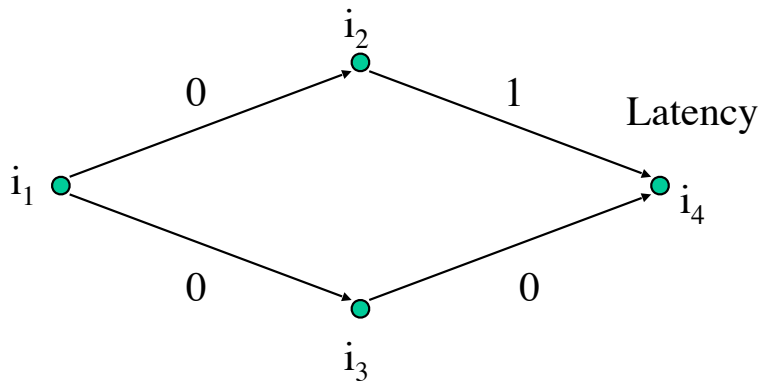
- *Acyclic* control flow is easier to deal with than *cyclic* control flow. Problems in dealing with cyclic flow:
  - A loop *implicitly* represent a large run-time program space compactly.
  - Not possible to open out the loops fully at compile-time.
  - Loop unrolling provides a partial solution.
  - Using the loop to optimize its dynamic behavior is a challenging problem.
  - Hard to optimize well without detailed knowledge of the range of the iteration.
  - In practice, profiling can offer limited help in estimating loop bounds

Acyclic Instruction Scheduling

- The acyclic case itself has two parts:
  - The simpler case that we will consider has no branching and corresponds to *basic block* of code, eg., loop bodies.
- Why basic blocks?
  - All instructions specified as part of the input must be executed.
  - Allows *deterministic* modeling of the input.
  - No "branch probabilities" to contend with; makes problem space easy to optimize using classical methods.

Example: Instruction Scheduling

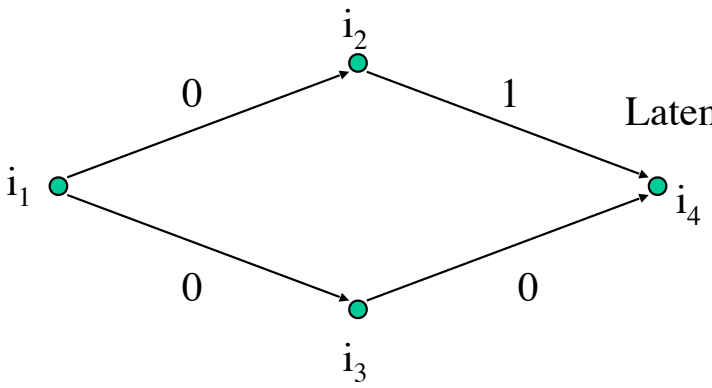
Input: *A basic block represented as a Directed Acyclic Graph (DAG)*

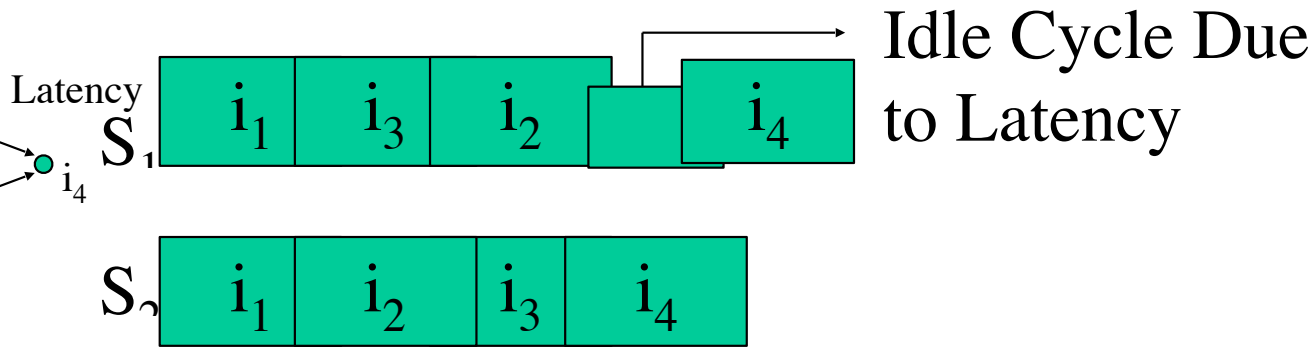


- $I_{\#}$ are instructions in the *basic block*; edges (*i, j*) represent dependence constraints
- i_2 is a load instruction.
- Latency of 1 on (i_2,i_4) means that i_4 cannot start for one cycle after i_2 completes.
- Assume 1 FU
- What are the possible schedules?

Example(cont): Possible Schedules

- Two possible schedules for the DAG
- The length of the schedule is the number of cycles required to execute the operations
  - Length(S_1) > Length(S_2)
- Which schedule is optimal?





Formalizing the Instruction Scheduling Problem

Input: DAG representing each basic block where:

1. Nodes encode *unit execution time* (single cycle) operations.

2. Each node requires a definite class of FU.

3. Additional time delays encoded as latencies on the edges.

4. Number of FUs of each type in the target machine.

more...

Formalizing the Instruction Scheduling Problem (contd)

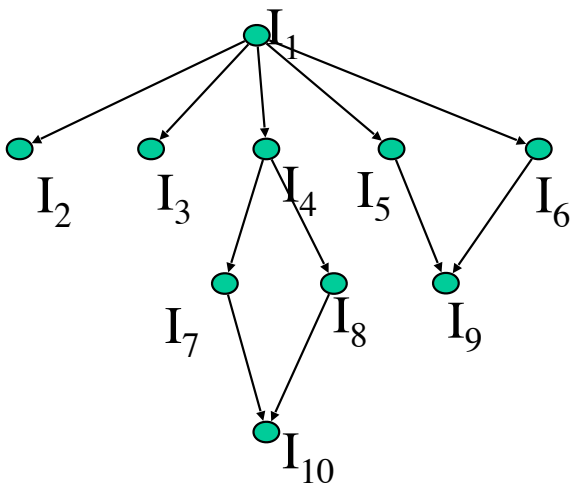
Feasible Schedule: A specification of a *start time* for each instruction such that the following constraints are obeyed:

1. Resource: Number of instructions of a given type of any time < corresponding number of FUs.
2. Precedence and Latency: For each predecessor *j* of an instruction *i* in the DAG, *i* is the started only δ cycles after *j* finishes where δ is the latency labeling the edge (*j*,*i*),

Output: A schedule with the minimum *overall completion time (makespan).*

Scheduling with infinite FUs

- Infinite FUs implies only the #2 constaint holds
  - Minimal length for a correct schedule can be obtained

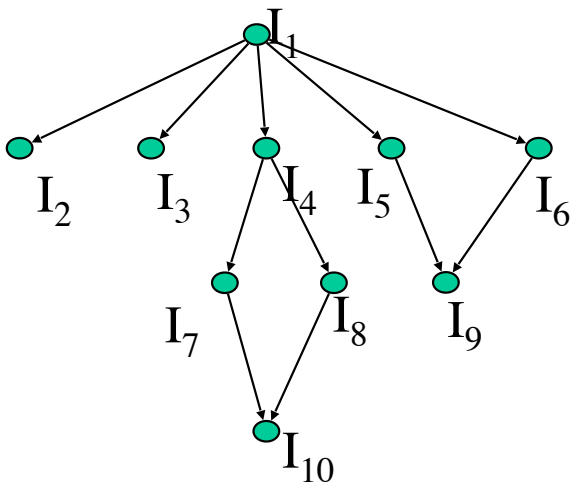


Input: DAG

Cycle	Ops
1.	I_1
2.	I_2, I_3, I_4, I_5, I_6
3.	I_7, I_8, I_9
4.	I_{10}

Scheduling with finite FUs

- Assuming 2 FUs
- What happens in Cycle #2? Must Choose from {I_2, I_3 , I_4, I_5, I_6}
  - How does an algorithm decide which ops to choose?
  - What factors may influence this choice?
    - Fanout
    - Height
    - Resources available



Input: DAG

Cycle	Ops
1.	I_1, <empty>
2.	??