CS4240 HW2 Solution (Spring 2019)

Question 1: Instruction Selection **[35 points]**

Consider the following sequence of IR instructions generated from the source code program fragment, "`a[b[i]] = a[j+1]`", for arrays with 4-byte elements (e.g., arrays of int's):

```
1.  t1 := i×4     // Offset calculation
2.  t2 := b+t1    // Address calculation - address in t2
3.  t3 := *t2     // Load the value at address t2 into t3
4.  t4 := t3×4
5.  t5 := a+t4
6.  t6 := 1+j
7.  t7 := t6×4
8.  t8 := a+t7
9.  t9 := *t8
10. *t5 := t9    // Store value t9 into location with address t5
```

The ISA that we are targeting for this problem includes the following instructions, with costs. This problem is focused on instruction selection. Assume that register allocation will be performed in a later pass, so you can generate instructions that refer to virtual IR registers such as i, j, a, b, and t1…t9.

add r3, r2, r1	r1 ← r2 + r3		Cost = 2
addi c, r2, r1	r1 ← r2 + c		Cost = 2
mul r3, r2, r1	r1 ← r2 × r3		Cost = 4
muli c, r2, r1	r1 ← r2 × c		Cost = 4
lshi c, r2, r1	r1← r2 << c	// Left shift r2's content by c bits, // (each shift equals multiplication by 2)	Cost = 2
lw r2, r1	r1 ← *r2	// Load value from address contained in r2 // into r1	Cost = 6
lwo r3, r2, r1	r1 ← *(r2 + 4 × r3)	// Do offset calculation with word offset // and load value from the computed address	Cost = 6
sw r2, r1	*r1 ← r2	// Store r2 into address contained in r1	Cost = 4
mm r2, r1	*r1 ← *r2	// Move (copy) value from address in r2 // to that in r1	Cost = 8
mmo r3, r2, r1	*r1 ← *(r2 + r3)	// Do offset calculation // and then move value	Cost = 10
mmc r3, r2, r1	*(r1+r3) ← *(r2+r3)	// Memory to memory move, r1 and r2 // are base pointers, r3 is the same offset	Cost = 11

a) Show the output of a simple instruction selection pass that emits machine instructions for one IR instruction at a time. The output should include the sequence of machine instructions with virtual registers, as well as the total cost. **[10 points]**

b) Show the dependence tree for the IR instructions, in which each vertex represents an IR instruction and each edge represents a dependence from a def to a use. **[5 points]**

c) Perform tiling using the greedy technique working top down to generate complex instructions that may subsume multiple IR operations and show the output of the resulting instruction selection pass. Compare the cost with that of part a) above. **[10 points]**

d) Perform tiling using the optimal dynamic programming technique, and show the output of the resulting instruction selection pass. Compare the cost with that of parts a) and b) above. **[10 points]**

a) The task was to simply generate one machine instruction per IR instruction. For algebraic identities (ex. (a x 4) , (a << 2)), it is desirable to choose the machine instruction with smallest cost.

b) Each non-terminal node of the dependence tree corresponds to an operation (arithmetic operations, load/store, etc). Terminal nodes of a dependence tree correspond to virtual registers or constants.

c) Greedy tiling attempts to cover up the dependence tree starting from the top node, choosing the largest tile applicable to the current node, and recursively going down to the children nodes of the chosen tile.

d) Below is an example pseudocode for using dynamic programming for tiling. Calling the function **"dp_tiling"** in the pseudocode below with the root node of a dependence tree as an argument will return the optimal tiling cost of the dependence tree.

```
# Tiles = List of all Tiles available
# (each tile corresponds to a machine instruction available in the host machine)

# node is a node in the dependence tree
int dp_tiling (node):
    if (node is terminal):
        return 0
    else: # node is non-terminal
        maxTile, maxCost = (None, 0)
        for tile in Tiles:
            if (tile can cover the top of the subtree where node is the root):
                # Fit the tile to node!
                currentTileCost = cost(tile)   # Cost of the tile alone.
                children = List of children nodes of tile
                for child_node in children:
                    currentTileCost += dp_tiling(child_node)
                if (maxCost < currentTileCost):
                    maxTile, maxCost = (tile, currentTileCost)
        return maxCost
```

The approach above is guaranteed to return the minimum cost needed for tiling the dependence tree, but it accompanies a lot of redundant calculation for the subtrees. Thus, using Dynamic Programming method with Memoization(Using a table to store the best tiling costs of the subtrees) can reduce redundant computation. Best tiling cost for each subtree will only be computed once when Memoization is used with Dynamic Programming method. The above logic will have to be modified to incorporate memoization.

The next 4 pages show a model solution for Q1 from one of our students, Yuuna Hoshi.

CS 4240 Homework 2, Spring 2019

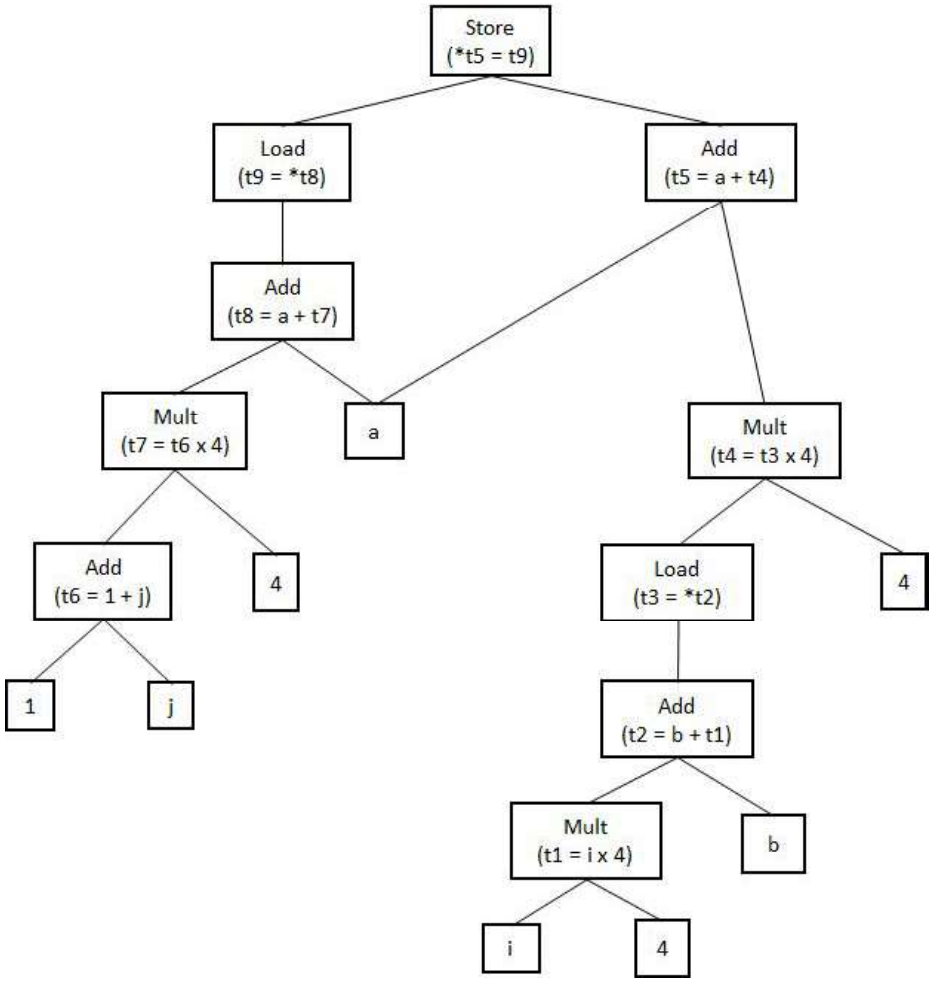
Yuuna Hoshi

Question 1: Instruction Selection

a) Simple instruction selection pass

IR	Machine code	Cost
t1 := i×4	**lshi 2, i, t1**	**2**
t2 := b+t1	**add b, t1, t2**	**2**
t3 := *t2	**lw t2, t3**	**6**
t4 := t3×4	**lshi 2, t3, t4**	**2**
t5 := a+t4	**add a, t4,**	**2**
t6 := 1+j	**addi 1, j, t6**	**2**
t7 := t6×4	**lshi 2, t6, t7**	**2**
t8 := a+t7	**add a, t7, t8**	**2**
t9 := *t8	**lw t8, t9**	**6**
*t5 := t9	**sw t9, t5**	**4**
		total cost = 30

b) Dependence tree



c) Greedy tiling

Machine instructions:

lwo i, b, t3

addi 1, j, t6

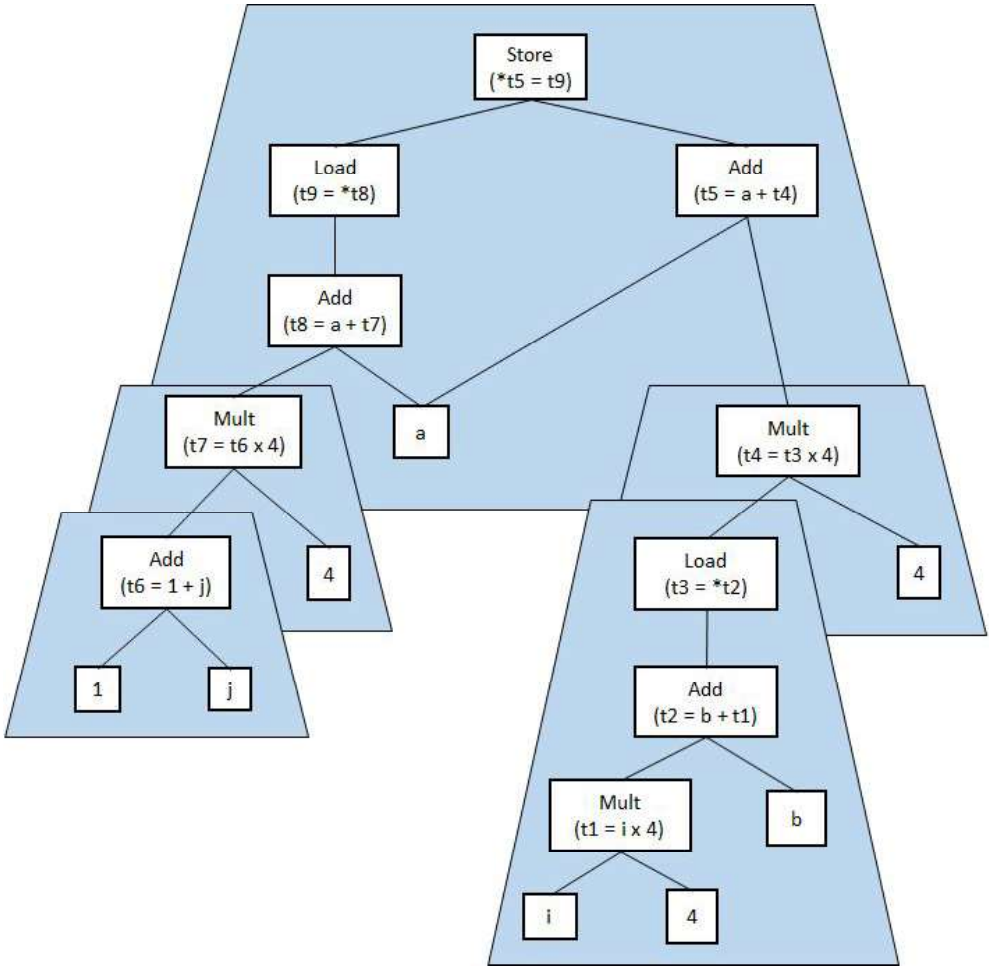
lshi 2, t6, t7

lshi 2, t3, t4

mmc a, t7, t4

Cost: 6 + 2 + 2+ 2 + 11 = 23

Cost is less than that of simple instruction selection (part a)



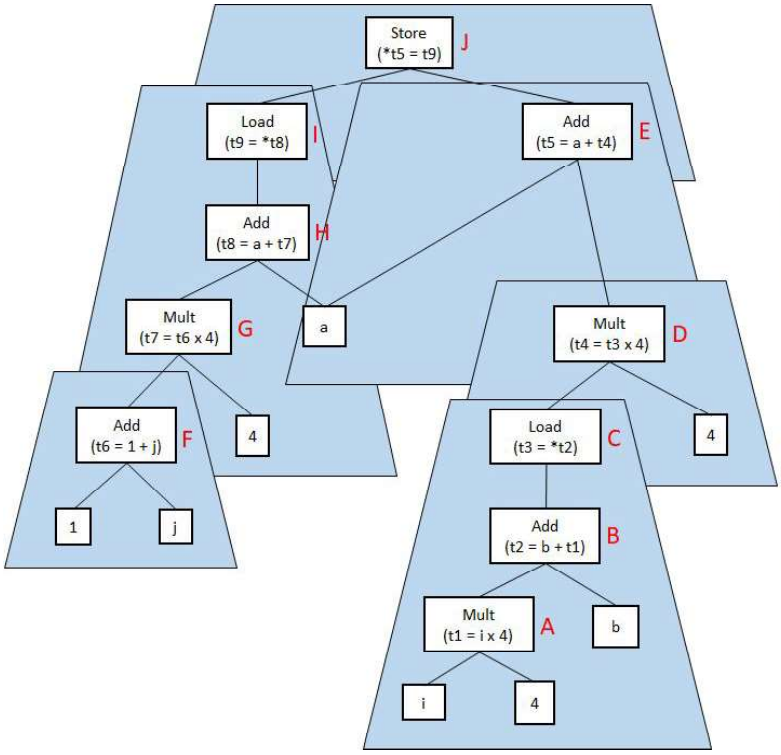
d) Dynamic programming tiling

Machine code:

lwo i, b, t3
lshi 2, t3, t4
add a, t4, t5
addi 1, j, t6
lwo t6, a, t9
sw t9, t5

Total cost: 6 + 2 + 2 + 2 + 6 + 4 = 22

Cost is less than that of both simple instruction selection (part a) and greedy tiling (part b)



Vertex	Instruction	Cost
A	lshi	2
B	add + best(A)	4
C	lwo	6
D	lshi + best(C)	8
E	add + best(D)	10
F	addi	2
G	lshi + best(F)	4
H	add + best(G)	6
I	lwo + best(F)	8
J	store + best(I) + best(E)	22

Question 2: Register Allocation **[35 points]**

Consider the following implementation, `fact`, of the factorial function in an IR, which takes an argument in virtual register **n** and returns a value stored in register **acc**:

```
1.  fact:              // label for start of program
2.    acc := 1
3.    ctr := n
4.  loop:              // label for loop entry
5.    c := ctr <= 1
6.    br c, done       // branch to label done if c = true
7.    acc := acc * ctr
8.    ctr := ctr-1
9.    jmp loop         // branch to label loop
10. done:              // label for loop exit return acc
```

a) Show the live ranges for each local variable (symbolic register) that occurs in **fact**. LIVE(v), the live range for variable v, is the set of program points at which variable v is live. We will follow the convention that a program point occurs at the entry point of a specific instruction. For example, variable **acc** is not live at the start of instruction 2, but is live at the start of instruction 3. Thus, LIVE(acc) should include {3}, but not include {2}. **[15 points]**

b) Show the interference graph for this program, in which each vertex corresponds to a local variable, and an undirected edge between two variables indicates that their LIVE sets have a non-empty intersection. **[10 points]**

c) Perform a spill-free register allocation for the above program, using the minimum number of physical registers. Your allocation should not assign the same register to more than one live variable at any program point. **[10 points]**

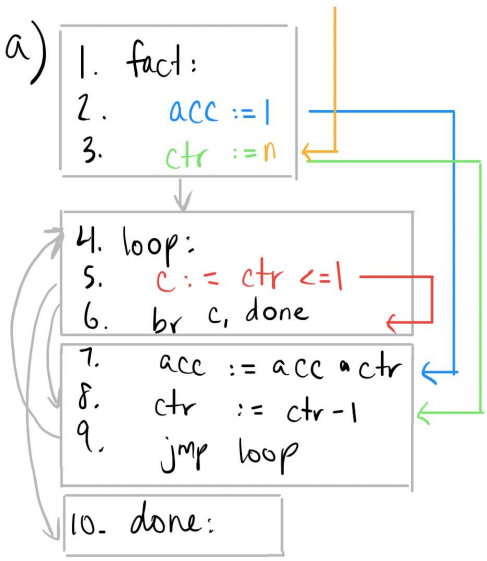
As per the descriptions from page 19 of lecture12 slides**, a value v is live at program point p if ∃ a path from p to some use of v along which v is not re-defined.**

By applying the above definition of a live variable at each program point, it is possible to figure out live variables at each program point. Using the dataflow equations showed in class with minimal basic blocks (one instruction per basic block) can also solve live ranges of the variables.

The next 2 pages show a model solution for Q2 from one of our students, Clifford Panos.

Question 2: Register Allocation

a)



LIVEIN = {n}
LIVEOUT = {acc, ctr}

LIVEIN = {acc, ctr}
LIVEOUT = {acc, ctr}

LIVEIN = {acc, ctr}
LIVEOUT = {acc, ctr}

LIVEIN = {acc}
LIVEOUT = {acc}

(Since returned by the function)

Result:

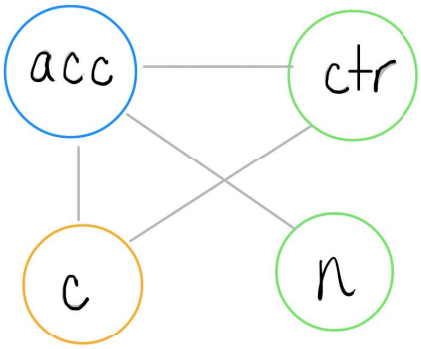
LIVE(n) = {1,2,3}

LIVE(acc) = {3,4,5,6,7,8,9,10}

LIVE(ctr) = {4,5,6,7,8,9}

LIVE(c) = {6}

b)



Total number of colors: 3

Total number of registers required given this coloring: 3

c)

```
1. fact:
2.    r1 ← 1
3.    r2 ← r2
4. loop:
5.    r3 ← r2 ≤ 1
6.    br r3, done
7.    r1 ← r1 * r2
8.    r2 ← r2 - 1
9.    jmp loop
10. done:
```

Assuming n starts in r2 before fact

Register Allocation:

acc: r1
ctr: r2
n: r2
c: r3

← See rewritten program.

Question 3: Phase Ordering of Instruction Scheduling and Register Allocation

a) Optimal Dependence Order:

v1
v5
v2
v4
v3
v6

⇒

Instruction Schedule:

1. t1 := X + i
2. t5 := *N
3. t2 := *t1
4. t4 := t1 + 1
5. *Z := t2
6. t6 := *(t4 + t5)

Register Allocation:

r1 ← X+i
r2 ← *N
r3 ← *r1
r1 ← r1 + 1
*Z ← r3
r3 ← *(r1+r2)

Register Allocation List

r1: t1, t4
r2: t5
r3: t2, t6

} Three registers obtained

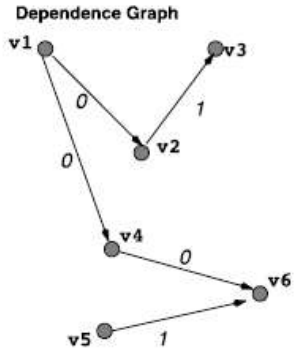
It can be seen that this approach requires three registers to be obtained. Additionally, the completion time of this scheduling is 6 cycles if you assume that v6 has a 0-cycle latency. If we were to incorporate v6's likely 1-cycle latency, then the completion time is 7 cycles.

Question 3: Phase Ordering of Instruction Scheduling and Register Allocation [30 points]

Consider the following sequence of assembly code instructions generated from source code of the form, "`Z = X[i]; print X[i+1+N];`" for a hypothetical RISC processor with machine instructions that are similar to IR instructions. For convenience, we use temporaries, t1, t2, t4, t5, and t6 to denote virtual/symbolic registers, and omit instructions for the print operation which will read the value of `X[i+1+N]` from `t6`. We also assume in this problem that no registers need to be allocated for integer i and addresses X, Z, and N, because they may be in registers as procedure arguments or they may be known constants.

```
1.  t1 := X + i     // Address of X[i] for single-byte elements
2.  t2 := *t1       // Load the value at address X[i] into t2
3.  *Z := t2        // Store the value of t2 into address Z
4.  t4 := t1 + 1
5.  t5 := *N        // Load the value at address N into t5
6.  t6 := *(t4+t5)  // Load the value at address X[i+1+N] into t6
```

The dependence graph for this machine code is shown below for instructions with virtual registers, where vertex `vi` corresponds to instruction `i`. Each outgoing edge from a load instruction (e.g., from v2 or v5) has a latency of 1 to indicate an extra delay slot needed to complete the load. For example, this implies that the start time of instruction v3 must be at least 2 cycles after the start time of instruction v2 in any legal schedule of this dependence graph.



In the following questions, you only need to show solutions to the problems. You are not expected to describe the algorithms used to obtain the solutions.

a) **Instruction scheduling before register allocation**. Show an instruction schedule with the smallest completion time for the dependence graph above with virtual registers. Then, show a register allocation for this schedule with the minimum number of physical registers. Report the completion time and number of registers obtained by this approach. **[10 points]**

b) **Register allocation before instruction scheduling.** Show a register allocation for the original instruction ordering (not the schedule from part a) above) with the minimum number of physical registers. Then, show how the dependence graph changes when physical registers are used instead of virtual registers. Finally, show an instruction schedule with the smallest completion time for this modified dependence graph with physical registers. Report the completion time and number of registers obtained by this approach. **[10 points]**

c) **Combined approach.** Perform both register allocation and instruction scheduling on this example so as to obtain a solution that uses only 2 registers, and has a completion time of 7 cycles. Show your solution. **[10 points]**

a) Instruction scheduling (re-ordering instructions) can lengthen or shorten the live-range of variables in the code. When a variable's live range is lengthened by instruction scheduling, register pressure is increased and this affects register allocation.

b) Register allocation maps variables (or virtual registers) to the limited pool of physical registers. This introduces additional data dependencies (ANTI, OUTPUT), which can possibly limit instruction scheduling.

c) In this case, the best possible completion time for b) was the same with c).

Students' answers for Q3 were considered as being correct if they were consistent with their assumptions.

The next 3 pages show a model answer for Q3 from one of the students, Aditya Parekh.

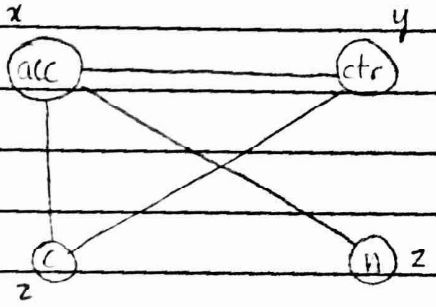
LIVE (acc) = { 3, 4, 5, 6, 7, 8, 9, 10 }

LIVE (ctr) = { 4, 5, 6, 7, 8, 9 }

LIVE (c) = { 6 }

LIVE (n) = { 1, 2, 3 }

b)



c) The inference graph in (b) requires 3 colours as indicated by x, y and z, hence the program requires a minimum of 3 physical registers.

acc : R_x

ctr : R_y

c : R_z

n : R_z

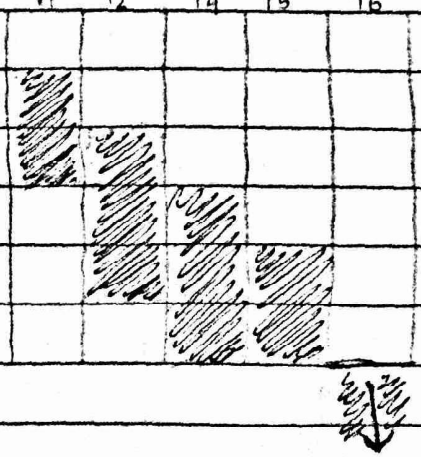
Question 3

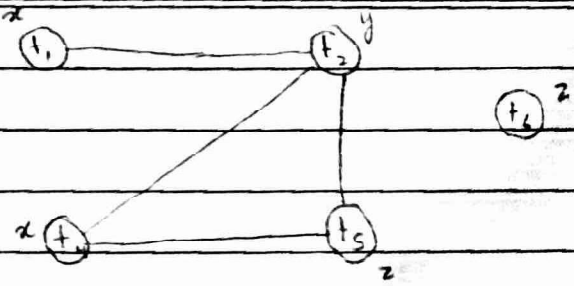
ASSUMPTION: v3 and v6 are "magic" instructions that have zero latency regardless of where they occur. This is the same as option (1) of the professor's Piazza post

a) Instruction Schedule. V_1 V_2 V_4 V_5 V_3 V_6

Cycles: 1 + 1 + 1 + 1 + 1 + 1 = 6 cycles

Register allocation

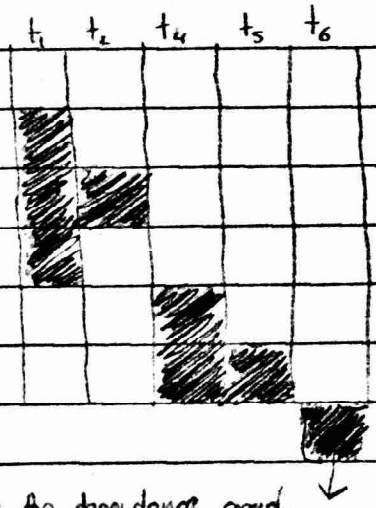




As indicated by the dependence graph, you need 3 registers. The letters x, y, z indicate the 3-colouring of the graph

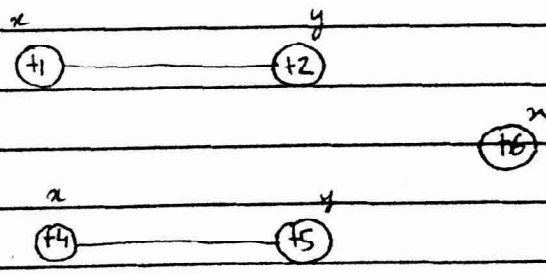
Registers = 3
Cycles = 6

b)



v_1 H := X + i
v_2 t2 := * t1
v_3 *Z := t2
v_4 t4 := H + 1
v_5 t5 := *N
v_6 t6 := * (t4 + t5)

Using the live ranges to draw the dependence graph



t1 : Rx
t2 : Ry
t4 : Rx
t5 : Ry
t6 : Rx

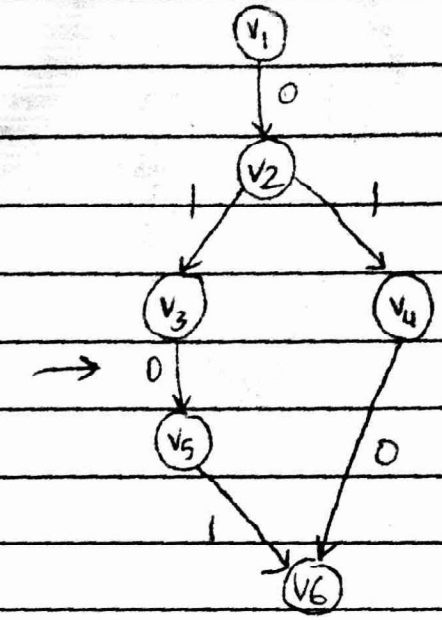
This graph is 2-colorable with x, y representing the colors.

Replacing virtual registers with physical registers, we get:

V_1 Rx := X + i
V_2 Ry := * Rx
V_3 *Z := Ry
V_4 Rx := Rx + 1
V_5 Ry := *N
V_6 Rx := * (Rx + Ry)

Date: / /

The dependence graph now looks like:



because of assumption stated earlier →

New Schedule: $V_1\ V_2$ — $V_3\ V_5$ $\underline{V_4}$ V_6

Cycles: 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 cycles using 2 registers.

∴ Completion Time = 7 cycles

Registers used = 2

c) Given the assumption stated above I achieved the same answer as in part (b).