

# CMSC 430 (Spring 2009)

## Practice Problems 6 Solutions

---

### 1. Instruction scheduling

Consider scheduling the code below using list scheduling. All instructions must complete before executing the *jmp* instruction. Assume the following instruction latencies:

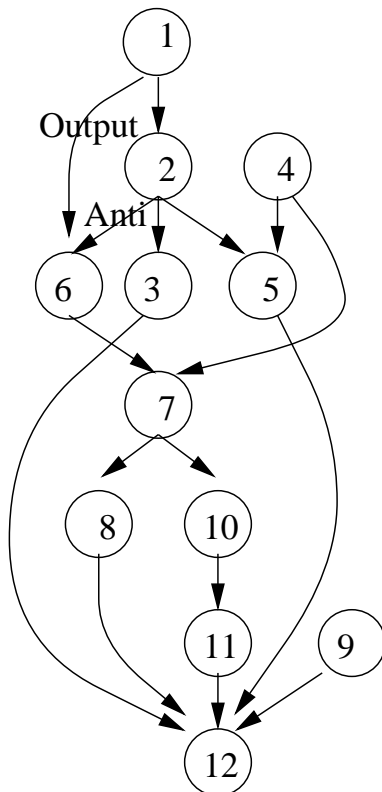
- 2-cycle latency for load
- 1-cycle latency otherwise

<op> <dst, s1, s2>

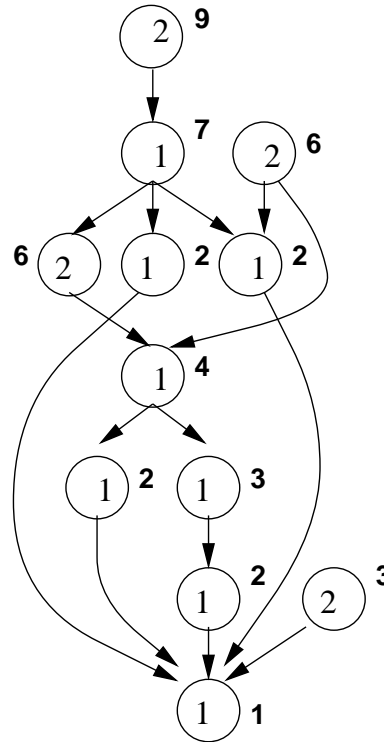
```

1 load  r1, a
2 add   r2, r1, #4
3 store x, r2
4 load  r3, b
5 mult  r4, r3, r2
6 load  r1, c
7 add   r5, r1, r3
8 store y, r5
9 load  r6, d
10 mult r7, r5, #1
11 store z, r7
12 jmp
    
```

- (a) Build the precedence graph for the instructions. Mark dependences as flow, anti, or output. You can ignore transitive dependences.



- (b) Calculate the critical path for the instructions. The following graph shows the critical path for a node as a number next to the node. The number inside a node indicates the latency of its instruction.



- (c) Schedule the instructions for a single-issue processor, using forward list scheduling. Showing candidates instructions at each cycle. Prioritize candidates using 1) critical path, 2) latency of instruction, 3) number of children.

Time	1-issue		2-issue	
	Cand	Inst	Cand	Inst
1	1,4,9	1	1,4,9	1,4
2	4,9	4	9	9
3	2,9	2	2	2
4	3,5,6,9	6	3,5,6	6,3
5	3,5,9	9	5	5
6	3,5,7	7	7	7
7	3,5,8,10	10	8,10	8,10
8	3,5,8,11	11	11	11
9	3,5,8	3	12	12
10	5,8	5		
11	8	8		
12	12	12		

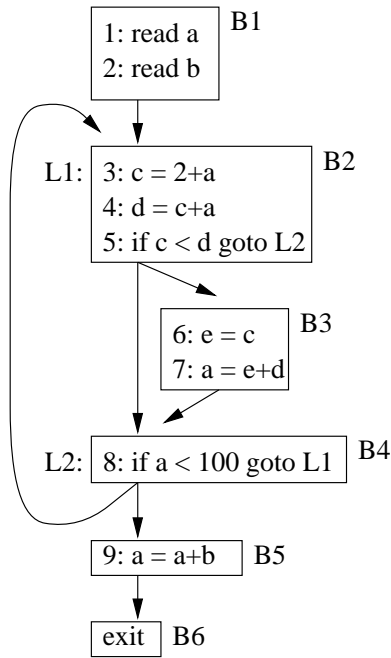
- (d) Schedule the instructions as above, for a two-issue VLIW processor. See above. Results show that issuing two instructions per cycle is sufficient to achieve maximum performance (equal to critical path).

- (e) How could you change register assignments to improve instruction schedules in the code?

Using a new register instead of r1 in instruction 6 & 7 would avoid an antidependence between instruction 2 & 6.

## 2. Register allocation

Consider the flow graph below. Each statement is labeled by its statement number and each basic block is labeled in the upper right hand corner.



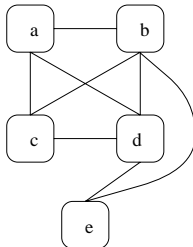
Use statement numbers or basic block numbers to indicate live ranges as appropriate.

- (a) What are the live ranges for a global top-down allocator?

a	1-5, 7-8
b	2-8
c	3-5
d	4-6
e	6

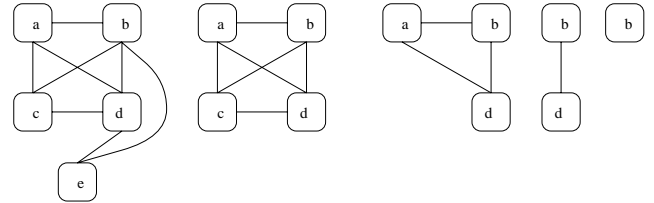
Displaying live sets on exit for each statement. E.g., live range for statement  $n$  indicates variable is live between statement  $n$  and  $n + 1$ .

- (b) Draw the interference graph for the live ranges.



- (c) Use the graph-simplification method to find a coloring for this graph.

Assume we need 4 registers (since each node has at least 3 neighbors). Begin simplification of the graph by iteratively removing all nodes with fewer than 4 neighbors as follows:



Since we simplify the graph down to a single node, we know 4 colors are sufficient. We can now assign colors to nodes in reverse order. For instance, one assignment would be:  $b=r1$ ,  $d=r2$ ,  $a=r3$ ,  $c=r4$ ,  $e=r4$ .

- (d) Can you color this graph with fewer colors?  
No, since nodes  $a, b, c, d$  are completely connected. They will thus require at least 4 colors.

- (e) If spilling is needed, which live range would be spilled first? Why?

Live range for  $b$ , since its spill cost is lowest due to its references not being in the loop (spill cost of instruction containing use or def is multiplied by loop nesting depth of instruction).

- (f) Draw the spill code needed if the value for  $c$  is spilled.

```

3: c = 2+a
   store mem[c], c // spill c
   load c, mem[c] // reload c
4: d = c+a
   load c, mem[c] // reload c
5: if c<d goto L2
   load c, mem[c] // reload c
6: e = c
  
```

- (g) What is rematerialization? Are there any such opportunities in the code?

Rematerialization refers to reducing the amount of spill code by recomputing a constant or near-constant value, rather than spilling and reloading it from memory. There are no opportunities in this example, since all values are non-constant.

### 3. Dependence analysis

Consider the following loop in Fortran:

```

A(N,N), B(N)
do i = 1,N
  do j = 2,N
    A(i,j) = A(i,j-1)+B(j)
  enddo
enddo

```

- (a) What are the dependences in the loop?

*From both references to A, there is a loop-carried true/flow dependence on the j loop with distance vector (0,1), since different iterations will access the same elements of A, with at least one of the references being a write.*

*Similarly, for the reference to B there is a loop-carried input dependence on the i loop with distance vector (1,0), since different iterations of i access the same iterations of J each time..*

- (b) What reuse exists in the loop? For each, give type, reference(s), and loop carrying reuse.

*Since arrays are stored column-major in Fortran, consecutive column elements are stored next to each other. As a result there is group-temporal reuse for A and spatial reuse for B for the j loop. There is spatial reuse for A and temporal reuse for B on the i loop.*

- (c) Which loop permutation is preferred? Calculate by estimating number of cache lines accessed by each loop. Show your work.

	loop i	loop j
A	$N/4 * N$	$N * N$
B	$1 * N$	$N/4 * N$
total	$N^2/4 + N$	$5N^2/4$

*The preferred permutation is j,i, since loop i accesses fewer cache lines (has more locality).*

- (d) Is it legal to interchange the i & j loops to make j the outer loop and i the inner loop?

*Loop interchange is legal. Input dependences do not affect the legality of program transformations, so the only dependence that needs to be considered is the loop-carried true/flow dependence between the two references to A. The distance vector of the dependence on A is (0,1), and would become (0,1) after applying the loop interchange program transformation. Since it does not become lexicographically negative, loop interchange is legal.*

- (e) Are either of the loops parallel? Explain?

*The j loop is not parallel. because it carries a true/flow dependence with distance vector (0,1). The i loop carries an input dependence with distance vector (1,0), but input dependences do not affect parallelism. Since the i loop does not carry any other dependences, it is parallel.*