Bruno J. Lucarelli
NetID: bjl145

Compilers HW2

**Problem 1**

1) Below is my bottom-allocated code with 3 available registers for allocation. Let r0 be the immutable machine register. Let new/physical r1, r2, and r3 be the allocatable registers. In the comments, we will denote physical registers as "new" rx, x being 1/2/3.

```
loadI 1024 => r0
loadI 1 => r1 // r1 = new r1;
loadI 2 => r2 // r1 = new r1; r2 = new r2
subI r2, 4 => r3 // r1 = new r1; r2 = new r2; r3 = new r3
// for next instruction: virtual r2's live range has ended, so we can use r2
// for allocating r4
add r1, r2 => r2 // r1 = new r1; r4 = new r2; r3 = new r3
// for next instruction: r1, r4, and r3 currently allocated. r1 is next
// referenced at 11, r3 at 7, and r4 at 6. thus we spill r1 (new r1).
storeAI r1 => r0, 4 // r4 = new r2; r3 = new r3; r1 = (r0, 4);
addI r2, 1 => r1 // r5 = new r1; r4 = new r2; r3 = new r3; r1 = (r0, 4);
// for next instruction (7): r3's live range has ended, so we can use rd for
// allocating r6
mult r3, r1 => r3 // r5 = new r1; r4 = new r2; r6 = new r3; r1 = (r0, 4);
// for next instruction (8): r5, r4, and r6 are currently allocated. r6's
// live range has ended, so we can use r3 for allocating r7.
sub r1, r3 => r3 // r5 = new r1; r4 = new r2; r7 = new r3; r1 = (r0, 4);
// for next instruction (9): r5, r4, and r7 are currently allocated. r4's
// live range has ended, so we can use rc for allocating r8.
add r2, r1 => r2 // r5 = new r1; r8 = new r2; r7 = new r3; r1 = (r0, 4);
// for next instruction (10): r5, r8, and r7 are currently allocated. r5's
// live range has ended, so we can use r1 for allocating r9.
add r2, r3 => r1 // r9 = new r1; r8 = new r2; r7 = new r3; r1 = (r0, 4);
// need to spill r1 into a register. r7 and r8 both have ended their live
// ranges, so we can use physical r3 for r1 and then physical r2 for r10.
loadAI r0, 4 => r3 // r9 = new r1; r8 = new r2; r1 = new r3;
add r1, r3 => r2 // r9 = new r1; r10 = new r2; r1 = new r3;
// can just execute the instructions as before.
storeAI r2 => r0, 8
outputAI r0, 8
```

```
OUTPUT:
-bash-4.2$ ./sim < bruno1-1.i
20
Executed 15 instructions and 15 operations in 25 cycles.
```

2) Below is my bottom-allocated code with 2 available registers for allocation. This part, I think, will be significantly more challenging because I need to spill way more frequently. Let r0 be the immutable machine register. Let new/physical r1 and r2 be the allocatable registers. In the comments, we will denote physical registers as "new" rx, x being 1 or 2. I copy-pasted it into a .i file and it turns out 20, the proper result.

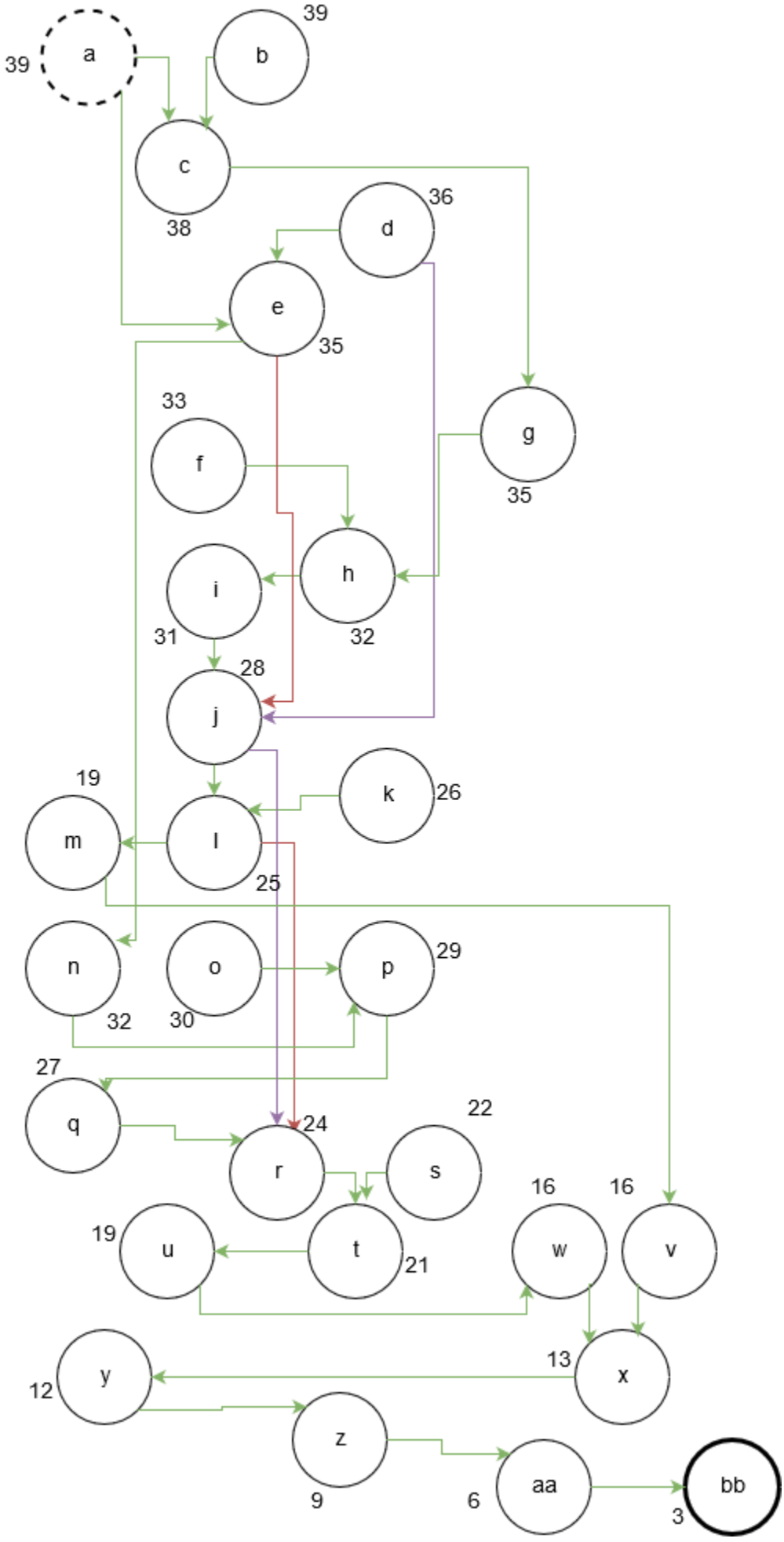
```
loadI 1024 => r0
loadI 1 => r1
// r1 = new r1;
loadI 2 => r2
// r1 = new r1; r2 = new r2;
// for next instruction (4): need a register open for r3. r1 is next used
// at instruction 5, r2 at 4. spill r1 to (r0, 4).
storeAI r1 => r0, 4
// r2 = new r2; r1 = (r0, 4);
subI r2, 4 => r1
// r3 = new r1; r2 = new r2; r1 = (r0, 4);
// need to take r1 back into a register. r3 is next used at instruction 7,
// and r2 is obviously used right here. spill r3, then take r1 back in.
storeAI r1 => r0, 8
// r2 = new r2; r1 = (r0, 4); r3 = (r0, 8);
loadAI r0, 4 => r1
// r1 = new r1; r2 = new r2; r1 = (r0, 4) (dirty); r3 = (r0, 8);
// we need a register to allocate for r4. r2's live range has ended, so use
// physical r2.
add r1, r2 => r2
// r1 = new r1; r4 = new r2; r1 = (r0, 4) (dirty); r3 = (r0, 8);
// for next instruction (6): need to take r5 into a register. r1 next used
// at 11, r4 immediately. spill r1.
storeAI r1 => r0, 4
// r4 = new r2; r1 = (r0, 4); r3 = (r0, 8);
addI r2, 1 => r1
// r5 = new r1; r4 = new r2; r1 = (r0, 4); r3 = (r0, 8);
// need r3 in a register for instruction 7. r4 next used in 9, r5 used
// immediately, so spill r4 to open up physical r2.
storeAI r2 => r0, 12
// r5 = new r1; r1 = (r0, 4); r3 = (r0, 8); r4 = (r0, 12);
loadAI r0, 8 => r2
// r5 = new r1; r3 = new r2; r1 = (r0, 4); r3 = (r0, 8); r4 = (r0, 12);
// for next instruction (7): need register for r6. r3's live range has ended,
// so store r6 in physical r2.
mult r2, r1 => r2
// r5 = new r1; r6 = new r2; r1 = (r0, 4); r4 = (r0, 12);
// for next instruction (8): need register for r7. r6's live range has ended,
// so store r7 in physical r2.
sub r1, r2 => r2
// r5 = new r1; r7 = new r2; r1 = (r0, 4); r4 = (r0, 12);
// need r4 in a register for instruction 9. r5 next used immediately, r7 at
// instruction 10. spill r7 (physical r2).
storeAI r2 => r0, 16
```

```
// r5 = new r1; r1 = (r0, 4); r4 = (r0, 12); r7 = (r0, 16);
loadAI r0, 12 => r2
// r5 = new r1; r4 = new r2; r1 = (r0, 4); r4 = (r0, 12); r7 = (r0, 16);
// for next instruction (9): need r8 in a register. r5 and r4 both have ended
// their live ranges, so use physical r1 for r8.
add r2, r1 => r1
// r8 = new r1; r1 = (r0, 4); r7 = (r0, 16);
// for next instruction (10): need r7 in a register. obviously, r2 is free,
// so we can just load it into there.
loadAI r0, 16 => r2
// r8 = new r1; r7 = new r2; r1 = (r0, 4); r7 = (r0, 16);
// for next instruction (10): need r9 in a register. r8 and r7 both have
// ended their live ranges, so we can just use physical r1 for r9.
add r1, r2 => r1
// r9 = new r1; r1 = (r0, 4);
// for next instruction (11): need r1 in a register. use physical r2 for r1.
loadAI r0, 4 => r2
// r9 = new r1; r1 = new r2; r1 = (r0, 4);
// since r9 and r1 both have ended their live ranges, store r10 in phys. r1.
add r1, r2 => r1
// r10 = new r1
// up to memory address 20 now. store r10 in that, and output from there.
storeAI r1 => r0, 20
outputAI r0, 20


OUTPUT:
-bash-4.2$ ./sim < bruno1-2.i
20
Executed 23 instructions and 23 operations in 51 cycles.
```

**Problem 2**

1) The dependence graph, with longest latency path labels, is given below. Treat green connections as true (W-A-R) dependences, purple connections as output (W-A-W) dependences, and red connections as anti (R-A-W) dependences.


a
39
b
39
c
38
d
36
e
35
f
33
g
35
h
32
i
31
j
28
k
26
l
25
m
19
n
32
o
30
p
29
q
27
r
24
s
22
t
21
u
19
v
16
w
16
x
13
y
12
z
9
aa
6
bb
3

2) I've labeled the nodes by the longest-latency-path heuristic above. Below, I will provide a table detailing my use of the aforementioned heuristic to do forward instruction scheduling. The "schedule" can be viewed as the sequence of instructions down the third column in bold, "Issued".

| Cycle | Ready (before issue) | **Issued** | Ready (after issue) | Active (after issue) | Done (after cycle) |
|---|---|---|---|---|---|
| 0 | a, b, d, f, o, s, k | **a** | b, d, f, o, s, k | a | a |
| 1 | b, d, f, o, s, k | **b** | d, f, o, s, k | b | b |
| 2 | c, d, f, o, s, k | **c** | d, f, o, s, k | c | |
| 3 | d, f, o, s, k | **d** | f, o, s, k | c, d | d |
| 4 | e, f, o, s, k | **e** | f, o, s, k | c, e | c |
| 5 | f, g, o, s, k | **g** | f, o, s, k | e, g | |
| 6 | f, o, s, k | **f** | o, s, k | e, f, g | e, f |
| 7 | n, o, s, k | **n** | o, s, k | g, n | g |
| 8 | h, o, s, k | **h** | o, s, k | n | h |
| 9 | i, o, s, k | **i** | o, s, k | i, n | n |
| 10 | o, s, k | **o** | s, k | i, o | o |
| 11 | p, s, k | **p** | s, k | i, p | i |
| 12 | j, s, k | **j** | s, k | p, j | p |
| 13 | q, s, k | **q** | s, k | j, q | |
| 14 | s, k | **s** | k | j, s, q | j, s |
| 15 | k | **k** | | k, q | k, q |
| 16 | l, r | **r** | l | r | |
| 17 | l | **l** | | r | l |
| 18 | m | **m** | | m, r | r |
| 19 | t | **t** | | m, t | |
| 20 | | **NOP** | | m, t | m, t |
| 21 | u, v | **u** | v | u | |
| 22 | v | **v** | | u, v | |
| 23 | | **NOP** | | u, v | u |
| 24 | w | **w** | | v, w | v, w |
| 25 | | **NOP** | | w | |
| 26 | | **NOP** | | w | w |
| 27 | x | **x** | | x | x |
| 28 | y | **y** | | y | |
| 29 | | **NOP** | | y | |
| 30 | | **NOP** | | y | y |
| 31 | z | **z** | | z | |
| 32 | | **NOP** | | z | |
| 33 | | **NOP** | | z | z |
| 34 | aa | **aa** | | aa | |
| 35 | | **NOP** | | aa | |
| 36 | | **NOP** | | aa | aa |
| 37 | bb | **bb** | | bb | |

| | | | | | |
|---|---|---|---|---|---|
| 38 | | **NOP** | | bb | |
| 39 | | **NOP** | | bb | bb |

3) I will now do a forward-list scheduling, but instead by using the highest-latency node heuristic rather than longest-latency-weighted path. If I have multiple instructions with the same latency, I'll just arbitrarily pick the one with the earliest alphabetic precedence (a > b > c).

**Note**: after finishing this, I've noticed that I've yielded the exact same table as in part 2). *I could, however, have arbitrarily selected a different order of instructions at points when I'm deciding between multiple instructions of the same latency, yielding different outcomes.* For example, if I had selected instructions 'o' and 'k' before instructions 'a' and 'b', I would have wasted two "free" moves that could have been spent waiting for 'c', which was immediately ready after selecting 'a' and 'b', all the while not actually freeing any higher-latency instructions. Thus, this heuristic could have yielded a greater number of cycles had I used a different tie-breaker!

| Cycle | Ready (before issue) | **Issued** | Ready (after issue) | Active (after issue) | Done (after cycle) |
|---|---|---|---|---|---|
| 0 | a, b, d, f, o, s, k | **a** | b, d, f, o, s, k | a | a |
| 1 | b, d, f, o, s, k | **b** | d, f, o, s, k | b | b |
| 2 | c, d, f, o, s, k | **c** | d, f, o, s, k | c | |
| 3 | d, f, o, s, k | **d** | f, o, s, k | c, d | d |
| 4 | e, f, o, s, k | **e** | f, o, s, k | c, e | c |
| 5 | f, g, o, s, k | **g** | f, o, s, k | e, g | |
| 6 | f, o, s, k | **f** | o, s, k | e, g | e |
| 7 | n, o, s, k | **n** | o, s, k | g, n | g |
| 8 | h, o, s, k | **h** | o, s, k | n | h |
| 9 | i, o, s, k | **i** | o, s, k | i, n | n |
| 10 | o, s, k | **o** | s, k | i | o |
| 11 | p, s, k | **p** | s, k | i, p | i |
| 12 | j, s, k | **j** | s, k | p, j | p |
| 13 | q, s, k | **q** | s, k | q, j | |
| 14 | s, k | **s** | k | q, j, s | j, s |
| 15 | k | **k** | | q, k | q, k |
| 16 | l, r | **r** | l | r | |
| 17 | l | **l** | | l, r | l |
| 18 | m | **m** | | m, r | r |
| 19 | t | **t** | | m, t | |
| 20 | | **NOP** | | m, t | m, t |
| 21 | u, v | **u** | v | u | |
| 22 | v | **v** | | u, v | |
| 23 | | **NOP** | | u, v | u |
| 24 | w | **w** | | v, w | v |
| 25 | | **NOP** | | w | |
| 26 | | **NOP** | | w | w |
| 27 | x | **x** | | x | x |
| 28 | y | **y** | | y | |
| 29 | | **NOP** | | y | |
| 30 | | **NOP** | | y | y |

| | | | | | |
|---|---|---|---|---|---|
| 31 | z | **z** | | z | |
| 32 | | **NOP** | | z | |
| 33 | | **NOP** | | z | z |
| 34 | aa | **aa** | | aa | |
| 35 | | **NOP** | | aa | |
| 36 | | **NOP** | | aa | aa |
| 37 | bb | **bb** | | bb | |
| 38 | | **NOP** | | bb | |
| 39 | | **NOP** | | bb | bb |