CS415 Homework 2, Spring 2018

Sample solution

1. Bottom-up Register Allocation

```
loadI 1024 => r0
loadI 1 => r1         // a := 1
loadI 2 => r2         // b := 2
subI r2, 4 => r3      // c := b - 4
add r1, r2 => r4      // d := a + b
addI r4, 1 => r5      // e := d + 1
mult r3, r5 => r6     // c * e
sub r5, r6 => r7      // f := e - (c * e)
add r4, r5 => r8      // d + e
add r8, r7 => r9      // g := (d + e) + f
add r9, r1 => r10     // h := g + a
storeAI r10 => r0, 4 // printing requires value to be in memory
outputAI r0, 4       // print @h = 4 , h is only value in memory
```

Live ranges, MAX_LIVE = 4

```
loadI 1024 => r0
loadI 1 => r1      // r1
loadI 2 => r2      // r1 r2
subI r2, 4 => r3   // r1 r2 r3
add r1, r2 => r4   // r1    r3 r4
addI r4, 1 => r5   // r1    r3 r4 r5
mult r3, r5 => r6  // r1       r4 r5 r6
sub r5, r6 => r7   // r1       r4 r5    r7
add r4, r5 => r8   // r1                r7 r8
add r8, r7 => r9   // r1                      r9
add r9, r1 => r10  //                            r10
storeAI r10 => r0, 4
outputAI r0, 4
```

1.a Bottom-up(MAX_LIVE - 1 = 3)

The $*$ marked tripel indicates the mapping of physical registers to virtual registers. For example, for three allocatable registers r1, r2, and r3, $*${r3, r7, r9} means that virtual register r3 is mapped to physical register r1, virtual register r7 is mapped to phsical register r2, and virtual register r9 is mapped to physical

register r9. The code listed below is just an example. There are many correct answers.

```
loadI 1024 => r0
loadI 1 => r1      // loadI 1 => r1  *{r1,(),()}
loadI 2 => r2      // loadI 2 => r2 *{r1,r2,()}
subI r2, 4 => r3   // subI r2, 4 => r3 *{r1,r2,r3}
add r1, r2 => r2   // add r1, r2 => r4 *{r1,r4,r3}
storeAI r1 => r0, 0   // spill r1(r1's life range still not over)
addI r2, 1 => r1   // addI r4, 1 => r5 *{r5,r4,r3}
mult r3, r1 => r3  // mult r3, r5 => r6 *{r5,r4,r6}
sub r1, r3 => r3   // sub r5, r6 => r7 *{r5,r4,r7}
add r2, r1 => r2   // add r4, r5 => r8 *{r5,r8,r7}
add r2, r3 => r2   // add r8, r7 => r9  *{r5,r9,r7}
loadAI r0, 0 => r1    // get r1 back    *{r1,r9,r7}
add r2, r1 => r1   // add r9, r1 => r10 *{r10,r9,r7}
storeAI r10 => r0, 4
outputAI r0, 4
```

1.b Bottom-up(MAX_LIVE - 2 = 2)

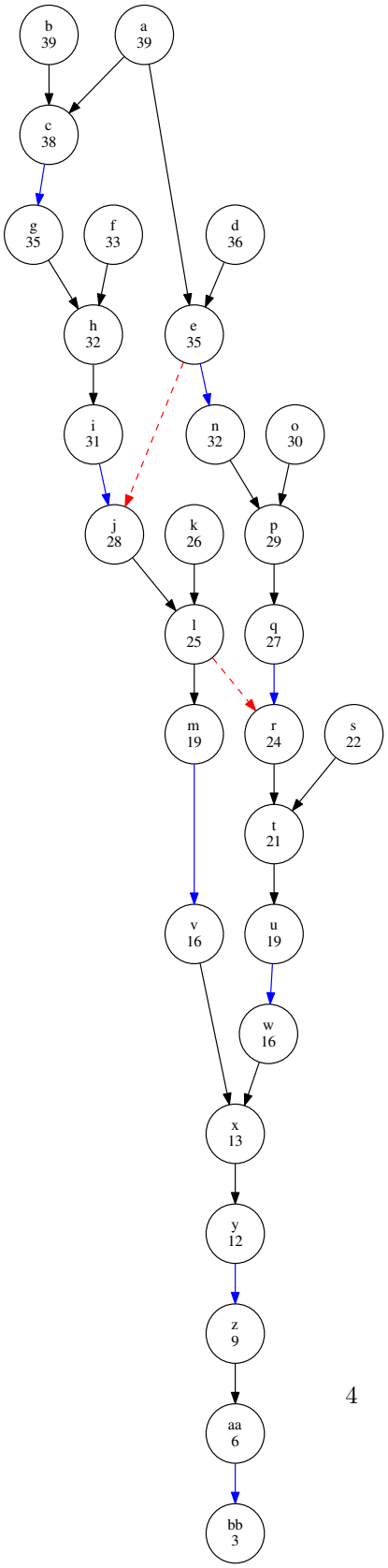
```
loadI 1024 => r0
loadI 1 => r1       // loadI 1 => r1     *{r1, ()}
loadI 2 => r2       // loadI 2 => r2     *{r1, r2}
storeAI r1 => r0,0                       *spill @r1 = 0
subI r2, 4 => r1    // subI r2, 4 => r3  *{r3, r2}
store r1 => r0,4                         * spill @r3 = 4
loadAI r0,0 => r1                        *{r1, r2} load r1 back
add r1, r2 => r1    // add r1, r2 => r4  *{r4, r2}
storeAI r2 => r0,8                       * spill @r2 = 8
addI r1, 1 => r2    //addI r4, 1 => r5  *{r4, r5}
storeAI r1 => r0, 12                     * spill @r4 = 12
loadAI r0, 4 => r1                       *{r3, r5} load r3 back
mult r1, r2 => r1   // mult r3, r5 => r6 *{r6, r5}
sub r2, r1 => r1    // sub r5, r6 => r7  *{r7, r5}
storeAI r1 => r0, 16                     * spill @r7 = 16
loadAI r0, 12 => r1                      *{r4, r5} load r4 back
add r1, r2 => r1    // add r4, r5 => r8  *{r8, r5}
loadAI r0, 16 => r2                      *{r8, r7} load r7 back
add r1, r2 => r1    // add r8, r7 => r9  *{r9, r7}
loadAI r0, 0 => r2                       *{r9, r1} load r1 back
add r1, r2 => r1    // add r9, r1 => r10 *{r10, r1}
storeAI r10 => r0, 4
outputAI r0, 4
```

2. Instruction Scheduling

Node name	Instruction			Latency
a	`loadI`	`1024`	`=> r0`	1
b	`loadI`	`0`	`=> r1`	1
c	`storeAI`	`r1`	`=> r0, 0`	3
d	`loadI`	`63`	`=> r3`	1
e	`storeAI`	`r3`	`=> r0, 4`	3
f	`loadI`	`5`	`=> r5`	1
g	`loadAI`	`r0, 0`	`=> r6`	3
h	`add`	`r5, r6`	`=> r7`	1
i	`storeAI`	`r7`	`=> r0, 8`	3
j	`loadAI`	`r0, 8`	`=> r3`	3
k	`loadI`	`9`	`=> r10`	1
l	`sub`	`r3, r10`	`=> r11`	1
m	`storeAI`	`r11`	`=> r0, 12`	3
n	`loadAI`	`r0, 4`	`=> r13`	3
o	`loadI`	`3`	`=> r14`	1
p	`mult`	`r13, r14`	`=> r15`	2
q	`storeAI`	`r15`	`=> r0, 16`	3
r	`loadAI`	`r0, 16`	`=> r3`	3
s	`loadI`	`7`	`=> r18`	1
t	`mult`	`r3, r18`	`=> r4`	2
u	`storeAI`	`r4`	`=> r0, 20`	3
v	`loadAI`	`r0, 12`	`=> r21`	3
w	`loadAI`	`r0, 20`	`=> r22`	3
x	`add`	`r21, r22`	`=> r23`	1
y	`storeAI`	`r23`	`=> r0, 24`	3
z	`loadAI`	`r0, 24`	`=> r25`	3
aa	`storeAI`	`r25`	`=> r0, 28`	3
bb	`outpuAI`	`r0, 28`		3

Dependency graph and path latencies

The dependency graph is as follows: True dependencies due to registers are shown as black edges, true dependencies due to memory loads/stores are shown as blue edges, and antidependencies due to registers are shown as dashed red edges.


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

Scheduling Result

We use three heuristics – the longest latency path (LLP), the highest single instruction latency (HL), and a custom heuristic, which we choose as the first available instruction (FA). That is, we choose to model the `Ready` queue as a FIFO and allocate the instruction that entered the queue earliest at each step.

Cycle - S(n)	LLP	HL	FA
0	a	a	a
1	b	b	b
2	c	c	d
3	d	d	f
4	e	e	k
5	g	g	o
6	f	f	s
7	n	n	c
8	h	h	e
9	i	i	-
10	o	k	g
11	p	o	n
12	j	j	-
13	q	p	h
14	k	s	i
15	l	q	p
16	r	l	-
17	s	m	j
18	m	r	q
19	t	-	-
20	-	v	l
21	u	t	m
22	v	-	r
23	-	u	-
24	w	-	v
25	-	-	t
26	-	w	-
27	x	-	u
28	y	-	-
29	-	x	-
30	-	y	w
31	z	-	-
32	-	-	-
33	-	z	x
34	aa	-	y
35	-	-	-
36	-	aa	-
37	bb	-	z
38	-	-	-
39	-	bb	-
40	-	-	aa
41	-	-	-
42	-	-	-
43	-	-	bb
44	-	-	-
45	-	-	-