Assignment 1

Compilers - CS415 - 02/11/2022

Reagan McFarland

NETID - rpm141

Problem 1 - ILOC code shape

Original code:

```
a := 2;
b := 3;
c := a + b;
d := a * b + a * c;
d := a + c + d;
PRINT d;
```

1. All variables may be aliased

Since all variables may be aliased, we have to make sure to commit changes to memory and load them before making operations

```
loadI 1024 => r0

// a = 2
loadI 2 => r1
storeAI r1 => r0,4

// b = 3
loadI 3 => r2
storeAI r2 => r0,8

// c = a + b
loadAI r0,4 => r1
loadAI r0,8 => r2
add r1,r2 => r3
storeAI r3 => r0,12

// a * b
loadAI r0,4 => r1
loadAI r0,8 => r2
mult r1,r2 => r3

// a * c
mult r1,r4 => r4
```

```
// d = a * b + a * c
add r3,r4 => r5
storeAI r5 => r0,16

// d = a + c + d
loadAI r0,4 => r1
loadAI r0,12 => r3
add r1, r3 => r4
add r4, r5 => r6

storeAI r6 => r0, 20
outputAI r0, 20
```

2. Variables a and b, and c and d may be aliased. However, both a and b are not aliased with c or d. Memory consistency has to be preserved.

We only have to make sure to commit changes to memory and load them before making operations involving a and b, and c and d.

```
loadI 1024 => r0

// a = 2
loadI 2 => r1
storeAI r1 => r0,4

// b = 3
loadI 3 => r2
storeAI r2 => r0,8

// c = a + b
// have to re load a since b can be aliased with it
loadAI r0,4 => r1
add r1,r2 => r3
storeAI r3 => r0,12

// a * b
// no need to load a or b again, since c cant be aliased with them
mult r1,r2 => r4

// a * c
mult r1, r3 => r5

// d = a * b + a * c
add r4, r5 => r5
storeAI r5 => r0, 16

// d = a + c + d
// have to re load c, since d can be aliased with it
loadAI r0, 12 => r3
add r1, r3 => r3 // a + c
add r3, r5 => r5
storeAI r5 => r0,16
```

```
outputAI r0, 16
```

3. No two variables are aliased. Memory consistency has to be preserved

With no variables able to be aliased, we don't have to load after stores. However, we do have to store to keep memory consistency

```
loadI 1024 => r0

// a = 2
loadI 2 => r1
storeAI r1 => r0,4

// b = 3
loadI 3 => r2
storeAI r2 => r0,8

// c = a + b
// no variables are aliased, no need to re load
add r1, r2 => r3
storeAI r3 => r0,12

// a * b
mult r1,r2 => r4

// a * c
mult r1,r3 => r5

// d = a * b + a * c
add r4, r5 => r6
storeAI r6 => r0, 16

// d = a + c + d
add r1, r3 => r7
add r6, r7 => r6
storeAI r6 => r0, 16

outputAI r0, 16
```

4. No two variables are aliased. Memory consistency may not be preserved

With no variables able to be aliased, we don't have to load after stores. Also, since memory consistency does not have to be preserved, we can do all of our storeAI's at the end of the block.

```
loadI 1024 => r0

// a = 2
loadI 2 => r1
```

```
// b = 3
loadI 3 => r2

// c = a + b
add r1, r2 => r3

// a * b
mult r1, r2 => r4

// a * c
mult r1, r3 => r5

// d = a * b + a * c
add r4, r5 => r6

// d = a + c + d
add r1, r3 => r7
add r6, r7 => r6

storeAI r1 => r0,4
storeAI r2 => r0,8
storeAI r3 => r0,12
storeAI r6 => r0,16

outputAI r0, 16
```

Problem 2 - Anti-Dependencies

Assuming the following latencies (from Lecture 4)

```
load = 3
loadI = 1
loadAI = 3
store = 3
storeAI = 3
add = 1
mult = 2
fadd = 1
fmult = 2
shift = 1
output = 1
outputAI = 1
```

1. What is the number of cycles needed to run this code assuming the latencies used in class (see lecture 3)? Do not reorder the instructions?

- a takes 1 cycle
- b takes 1 cycle
- c takes 3 cycles
- d takes 1 cycle

- e takes 3 cycles
- f takes 3 cycles
- g takes 1 cycle
    - g can be pipelined directly after f, since it has no dependencies to f. Therefore, it will only take 1 additional cycle (2/3 of g will be pipelined with f)
- h takes 1 cycle
- i takes 3 cycles
- j takes 1 cycle

It takes a total of 18 cycles for this code to run

> Note: I'm assuming we are allowed to pipeline g with f, if not then this will take 20 cycles not 18.

2. Can you remove the anti dependencies? If so, give the code. What is the number of cycles needed to run the modified code without anti-dependencies using latencies above. Do not reorder or eliminate any instructions?

To remove the first anti-dependency, we can change `d` to use `r4` instead of `r1`, also making sure to update `e` to `store r4 => ...`. To get rid of of the other anti-dependency, we also change `f` to loading into `r5` instead of `r1`, also making sure to update `h` to use `r5` instead of `r1`.

Modified code:

```
a   loadI   1024  => r0
b   loadI   2     => r1
c   storeAI r1    => r0, 4
d   loadI   3     => r4
e   storeAI r4    => r0, 8
f   loadAI  r0, 4 => r5
g   loadAI  r0, 8 => r2
h   add     r5, r2 => r3
i   storeAI r3    => r0, 12
j   outputAI r0, 12
```

The number of cycles needed to run the modified code is actually the same as the original code, since even though we got rid of the anti-dependencies, we didn't reorder any instructions and are just using different registers.

3. What are the advantages and disadvantages of removing anti-dependencies?

One of the most clear advantages of removing anti-dependencies is that it removes constraints on the reordering of instructions for the compiler's instruction scheduler.

One clear disadvantage of removing anti-dependencies is the increased demand for registers. Renaming registers to remove anti-dependencies increase the demand for registers, potentially forcing the register allocator to spill more values. Spilling these values into memory can cause very (relatively) long latency operations by having to access the memory when there are more named registers than actual registers on that particular architecture.

Problem 3 - Instruction Scheduling

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

1. Show the assignment S(n) of instruction issue times to instructions when no list scheduling is performed. How many cycles does the program take?

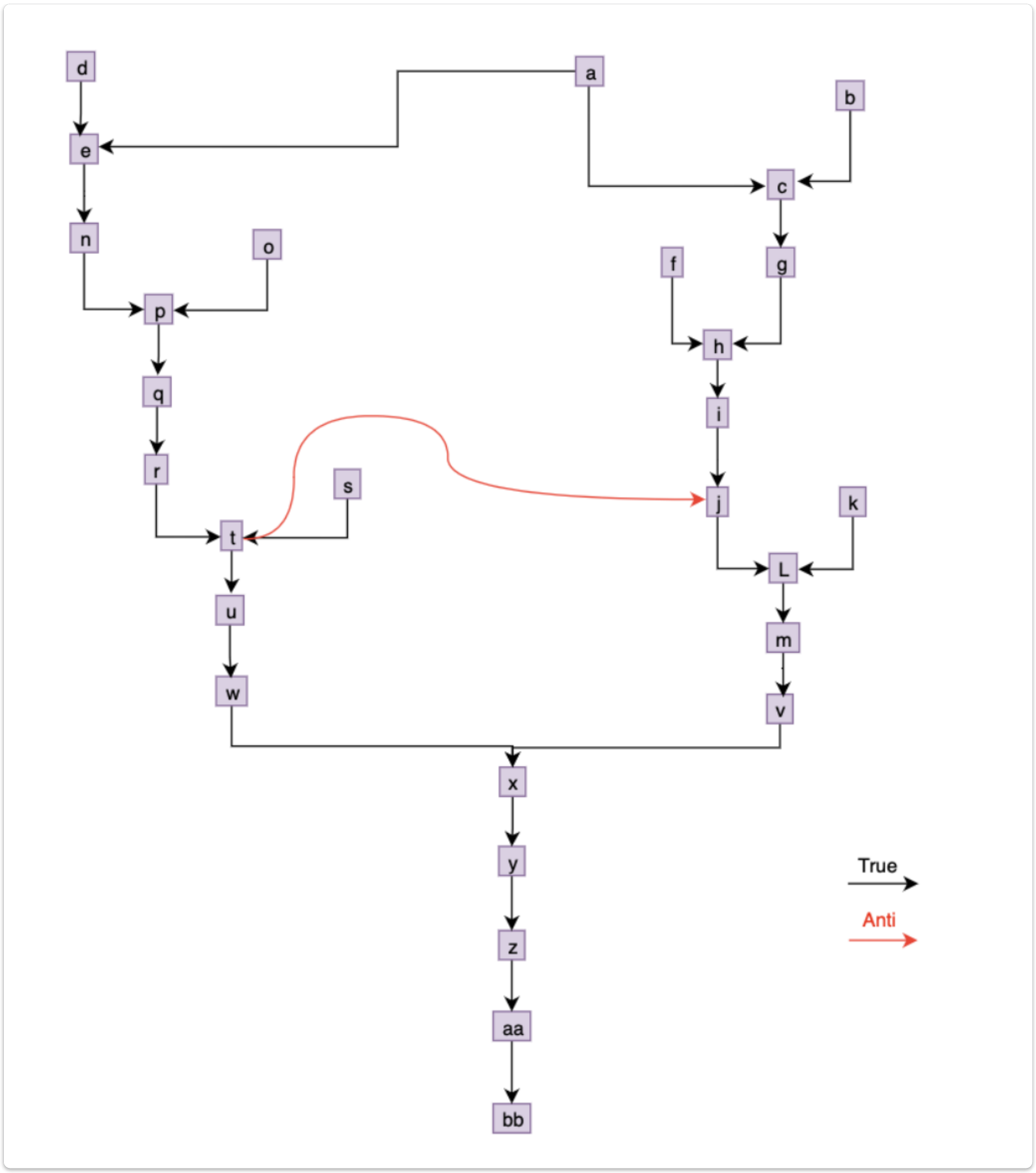
Assuming we can pipeline instructions:

- a = S(0)
- b = S(1)
- c = S(2) (r1 and @0 can't be used until S(5))
- d = S(3)
- e = S(4) (r3 and @4 can't be used until S(7))
- f = S(5)
- g = S(6) (r6 can't be used until S(9))
- h = S(9)
- i = S(10) (r7 and @8 can't be used until S(13))

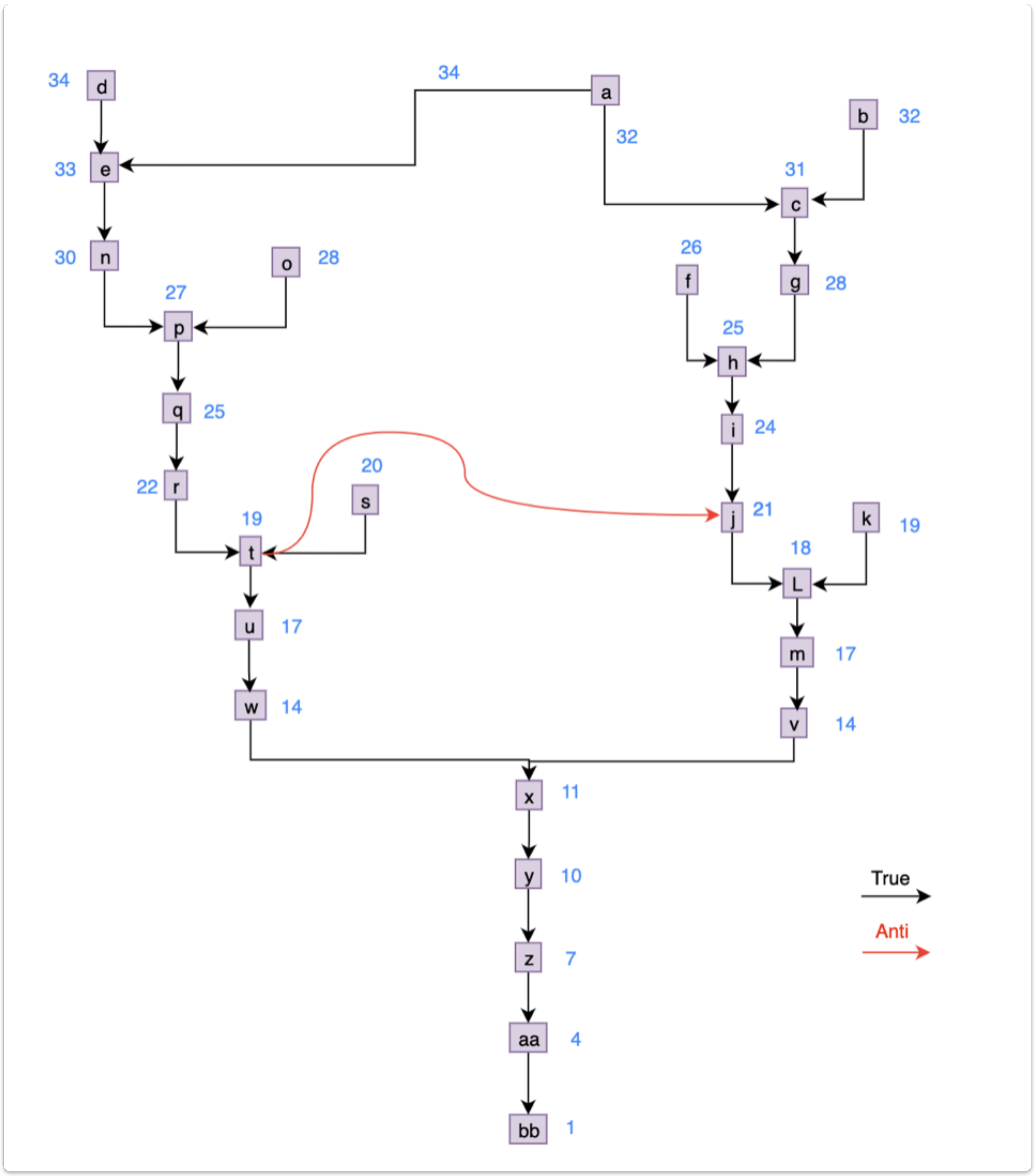
- j = S(13) (r3 and @8 can't be used until S(16))
- k = S(14)
- l = S(16) (assuming sub is 1 cycle)
- m = S(17) (r11 and @12 can't be used until S(20))
- n = S(18) (@4 and r13 can't be used until S(21))
- o = S(19)
- p = S(21) (r15 can't be used until S(23))
- q = S(23) (r15 and @16 can't be used until S(26))
- r = S(26) (@16 and r3 can't be used until S(29))
- s = S(27)
- t = S(29) (r4 can't be used until S(31))
- u = S(31) (r4 and @20 can't be used until S(34))
- v = S(32) (r21 and @12 can't be used until S(35))
- w = S(34) (r22 and @20 can't be used until S(37))
- x = S(37)
- y = S(38) (r23 and @24 can't be used until S(41))
- z = S(41) (r25 and @24 can't be used until S(44))
- aa = S(44) (@28 can't be used until S(47))
- bb = S(47)

The program takes a total of 48 cycles (since outputAI is scheduled on the 47th cycle and takes 1 cycle to finish)

2. Show the dependency graph for the basic block



3. Label the nodes in the dependency graph based on the longest latency-weighted path.



4. Show the result of forward list scheduling, i.e., S(n) using the longest latency weighted path heuristic. How many cycles does the program take

Cycle	Ready set	Active set	What happened
	a,d,b,o,f,s,k		
0	d,b,o,f,s,k	a	Move a to the Active set
1	b,o,f,s,k	d	a finished. Move d to the Active set
2	b,o,f,s,k	e	d finished, which add e to the Ready set. Move e to the Active set
3	o,f,s,k	e,b	e has not finished. Add b to the Active set

Cycle	Ready set	Active set	What happened
4	o,f,s,k	e,c	b has finished, which moves c to the Ready set. Move c to the Active set
5	f,s,k	e,c,o	e and c have not finished. Move o to the Active set
6	f,s,k	c,n	e and o finish, which adds n to the Ready set. Move n to the Active set
7	f,s,k	n,g	c finished, which adds g to the Ready set. Move g to the Active set
8	s,k	n,g,f	n and g have not finished. Move f to the Active set
9	s,k	g,p	n and f both finish, which adds p to the Ready set. Move p to the Active set
10	s,k	p,h	g finished, which adds h to the Ready set. Move g to the Active set
11	s,k,i	q	p and h both finish, adding i and q to the Ready set. Move q to the Active set
12	s,k	q,i	q has not finished, which adds i to the Active set
13	k	q,i,s	q and i have not finished, which adds s to the Active set
14	k	i,r	q and s finished, which adds r to the Ready set. Move r to the Active set
15		r,k	i finished, which doesn't add anything to the ready set (because t and j form an anti-dependency). Move k to the Active set
16		r	k finished, which doesn't add anything to the ready set. Nothing to do (noop)
17		t	r finished, which adds t to the Ready set. Move t to the Active set.
18		t	t is not yet finished, nothing to do (noop)
19	u	j	t finished, which adds u and j to the Ready set. Move j to the Active set
20		j,u	Move u to the Active set.
21		j,u	Nothing to do (noop)
22		u,l	j finishes, which adds l to the Ready set. Move l to the Active set.
23	w	m	u and l both finish, adding m and w to the Ready set. Move m to the Ready set.
24		m,w	m is not finished yet, add w to the Active set.
25		m,w	Nothing to do (noop)

Cycle	Ready set	Active set	What happened
26		w,v	m finishes, which moves v to the Ready set. Move v to the Active set
27		v	w finishes, nothing to add to Active set (noop)
28		v	v hasn't finished, nothing to do (noop)
29		x	v has finished, adding x to the Ready set. Move x to the Active set
30		y	x has finished, move y to the Ready set. Move y to the Active set.
31		y	waiting on y, noop
32		y	noop
33		z	y finished, adding z to the Ready set. Move z to the Active set.
34		z	waiting on z, noop
35		z	noop
36		aa	z finished, adding aa to the Ready set. Move aa to the Active set.
37		aa	waiting on aa, noop
38		aa	noop
39		bb	aa finished, adding bb to the Ready set. Move bb to the Active set.
40			Everything done.

Reordered code

```
a    loadI 1024 => r0
d    loadI 63 => r3
e    storeAI r3 => r0,4
b    loadI 0 => r1
c    storeAI r1 => r0,4
o    loadI 3 => r14
n    loadAI r0,4 => r13
g    loadAI r0,0 => r6
f    loadI 5 => r5
p    mult r13, r14 => r15
h    add r5,r6 => r7
q    storeAI r15 => r0,16
i    storeAI r7 => r0,8
s    loadI 7 => r18
r    loadAI r0,16 => r3
k    loadI 9 => r10

t    mult r3, r18 => r4

```

```
19   j    loadAI r0,8 => r3
20   u    storeAI r4 => r0,20
21
22   l    sub r3, r10 => r11
23   m    storeAI r11 => r0,12
24   w    loadAI r0, 20 => r22
25
26   v    loadAI r0,12 => r21
27
28
29   x    add r21, r22 => r23
30   y    storeAI r23 => r0,24
31
32
33   z    loadAI r0, 24 => r25
34
35
36   aa   storeAI r25 => r0,28
37
38
39   bb   outputAI r0, 28
40
```

The program now takes **40** cycles