University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Spring 1998 D.A. Patterson
Solution #1

Solutions Set #1
CS252 Graduate Computer Architecture

4.2 There is a data dependence in `a[i]` from S1 to S2 and S3. There is an antidependence in `b[i]` from S1 to S2. There is an output dependence in `a[i]` from S3 from one iteration of the loop to S1 in the next.

To parallelize the loop, we have to remove any circular dependences. We can break the cycle by noticing that each value of `a[]` gets written twice (except for the first and last value). By removing one of these instructions (which happens to be part of the cycle), we can break the cycle.

```
for (i=1;i<100;i=i+1) {
    a[i] = b[i] + c[i];
    b[i] = a[i] + d[i];
}
a[100] = a[99] + e[99];    /* Write the last value of the loop */
```

4.10 If we have a fixed 2-cycle branch penalty:

$$
\begin{aligned}
\text{CPI} &= \text{CPI}_{\text{base}} + (\text{branch frequency}) * (\text{branch penalty}) \\
&= 1 + (.15) * 2 \\
&= 1 + .3 = 1.3
\end{aligned}
$$

If we are using the BTB:

$$
\begin{aligned}
\text{CPI} &= \text{CPI}_{\text{base}} + (\text{branch frequency}) * [(\text{buffer miss rate}) * (\text{buffer miss penalty}) \\
&\quad + (\text{buffer hit rate}) * (\text{misprediction rate}) * (\text{misprediction penalty})] \\
\text{CPI} &= 1 + (.15) * [(.1) * (3) + (.9) * (.1) * (4)] \\
\text{CPI} &= 1 + (.15) * [.3 + .36] = 1.099
\end{aligned}
$$

The the BTB is $\frac{1.3}{1.1099} = 1.18$ times faster.

4.14 This problem had a few ambiguous parts. It's OK if your solution is different than what is here, as long as you made reasonable assumptions and clearly stated what those were.

In some cases, the instructions were contradictory. I tried to do what made sense in these solutions.

To solve this problem, you have to determine how many execution cycles a FP ALU op has. This number depends on certain assumptions, but for a certain set of assumptions, every FP ALU op has the same number of execution cycles.

c) For the scoreboard: I assumed that a FP ALU op's Write Results stage and another dependent FP ALU op's Read Operands stage could not occur in the same stage (check out page 245 and 246 for two apparently contradictory places in the book where it discusses this).

To determine how many execution cycles FP ALU ops take, we use the information that there is a latency of 3 between FP ALU ops, and the fact that RO-WR can not overlap (as per our assumptions). Then, we can line up the instructions like this:

```
FP1   IS          RO      (some number of EX)       WR
FP2               IS             (three stalls)               RO
```

So, we can see that there must be two execute stages.

For the first solution given here, integer instructions pass through all four stages. I am also assuming that an instruction that depends on a load can not go through the RO stage until the load is in its WR stage.

Instruction	FP Unit & Res. St.	cycles in IS	RO	EX	WR	Comments
`LD F2, 0(R1)`	INT	1	2	3	4	
`MULTD F4, F2, F0`	MUL1	2	4	5-6	7	(waits for `LD`)
`LD F6, 0(R2)`	INT	5	6	7	8	(IS in 5 because of struc haz)
`ADDD F6, F4, F6`	ADD1	9	10	11-12	13	(stalls to prevent WAW)
`SD 0(R2), F6`	INT	10	13	14	15	(waits for `ADDD`)
`ADDI R1, R1, #8`	ADD	16	17	18	19	(IS in 16 because of struc haz)
`ADDI R2, R2, #8`	ADD	20	21	22	23	(IS in 20 because of struc haz)
`SGTI R3, R1, done`	INT	24	25	26	27	(IS in 24 because of struc haz)

d) For Tomasulo: a dependent instruction's execute can not start until the cycle after the producing instruction has completed its WR stage (see page 259). Only one instruction can write to the CDB per cycle. `SD` instructions do not write to the CDB. I'm also assuming the integer unit is pipelined and there are enough reservation stations so that no instruction has to wait to issue.

Again, we have to check how many execute stages FP ALU ops take.

```
FP1   IS          (some number of EX)       WR
FP2               IS             (three stalls)          EX
```

So there will be three execute stages.

Instruction Status					
Instruction	FP Unit & Res. St.	cycles in IS	EX	WR	Comments
`LD F2, 0(R1)`	LOAD1	1	2	3	
`MULTD F4, F2, F0`	MUL1	2	4-6	7	(waits for `LD`)
`LD F6, 0(R2)`	LOAD2	3	4	5	
`ADDD F6, F4, F6`	ADD1	4	8-10		(waits for `MUL`)
`SD 0(R2), F6`	STORE1	5			(waits for `ADDD`)
`ADDI R1, R1, #8`	INT1	6	7	8	
`ADDI R2, R2, #8`	INT2	7	8	9	
`SGTI R3, R1, done`	INT3	8	9	10	

Reservation Stations						
Reservation Station	Busy	Op	Vj	Vk	Qj	Qk
Add1	Yes	ADDD	Regs[F4]	Regs[F6]		

Store Buffers			
Store Buffer	Busy	Qi	Address
Store1	Yes	Add1	Regs[R2]

Register Status		
F0 – F4	F6	F8 – F30
	Add1	

4.19 Here is the original loop:

```
Loop: LD   F0, 0(R1)
      ADDD F4, F0, F2
      SD   0(R1), F4
      SUBI R1, R1, #8
      BNEZ R1, Loop
```

If the ADDD instruction has a latency of 5, there will be a stall even if we software pipeline the loop. If we unroll the loop once first:

```
Loop: LD   F0, 0(R1)
      LD   F6, 8(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      SD   0(R1), F4
      SD   8(R1), F8
      SUBI R1, R1, #16
      BNEZ R1, Loop
```

and then software pipeline it:

```
Loop: SD   40(R1), F8
      SD   32(R1), F4
      ADDD F8, F6, F2
      ADDD F4, F0, F2
      LD   F6, 8(R1)
      LD   F0, 0(R1)
      SUBI R1, R1, #16
      BNEZ R1, Loop
```

then we can execute the loop without stalls. (Of course, we also need to include some start-up and clean-up code.)

B.2 For this problem, I assumed that only one instruction could issue per cycle. Also, for independent instructions in the same convoy, I assumed that each instruction's first-result and last-result time is independent. I assumed that separate convoys could not overlap; pages B-30 to B-35 discuss the advantages and disadvantages of this technique for reducing the effect of startup overhead.

a) If there is no chaining, the LV can not be in the same convoy as the MULTV since the MULTV uses the result of the LV. The MULTV and the ADDV can be in the same convoy, but that convoy cannot include the first SV (since it uses a result of the MULTV). The two SVs cannot be in the same convoy, since there is only a single memory pipeline. Thus, we have a total of four convoys.

Convoy	Starting Time	First-result time	Last-result Time
1. `LV`	0	$0 + 12 = 12$	$11 + 64 = 75$
2. `MULTV, ADDV`	76	$76 + 7 = 83$	$82 + 64 = 146$
3. `SV`	147	$147 + 12 = 159$	$158 + 64 = 222$
4. `SV`	223	$223 + 12 = 235$	$234 + 64 = 298$

For `MULTV` starts on cycle 76 and produces its first result on cycle 83. The `ADDV` starts on cycle 77 and produces its first result on cycle 83. Since the total sequence takes 299 cycles, and we store 64 results, the number of clock cycles per result is $\frac{299}{64} = 4.67$.

b) Since we are now chaining the instructions, we can place combine the first two convoys. We can't combine those with the next convoy, because we still only have a single memory pipeline. For the same reason, the final two convoys can not be combined; so, we have a total of three convoys.

Convoy	Starting Time	First-result Time	Last-result Time
1. `LV, MULTV, ADDV`	0	$0 + 12 = 12$	$18 + 64 = 82$
2. `SV`	83	$83 + 12 = 96$	$95 + 64 = 159$
3. `SV`	160	$160 + 12 = 172$	$171 + 64 = 235$

For the first convoy, the `LV` will start on cycle 0 and produce its first result on cycle 12. The `MULTV` will start on cycle 12 and produce its first result on cycle 19. The `ADDV` will start on cycle 13 and produce its first result on cycle 19. Thus, the last result from the convoy is produced on cycle 19 + (n-1).

Since the total sequence takes 236 cycles, and we store 64 results, the number of clock cycles per result is $\frac{236}{64} = 3.69$.

c) Now that we have three memory pipelines, the entire sequence can be done in one convoy.

Convoy	Starting Time	First-result Time	Last-result Time
1. `LV, MULTV, ADDV, SV, SV`	0	$0 + 12 = 12$	$31 + 64 = 95$

The `LV` starts on cycle 0, and produces its first result on cycle 12. The `MULTV` starts on cycle 12, and produces its first result on cycle 19. The `ADDV` starts on cycle 13, and produces its first result on cycle 19. The first `SV` starts on cycle 19, and produces its first result on cycle 31. The second `SV` starts on cycle 20, and produces its first result on cycle 32. Thus, the last result is produced on cycle 95.

Since the total sequence takes 96 cycles, and we store 64 results, the number of clock cycles per result is $\frac{96}{64} = 1.5$.