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

Fall 1998 J.D. Kubiatowicz
Solution Set #1

Solution Set #1
CS252 Graduate Computer Architecture

4.2

(a) The given loop has the following dependencies:

true data dependence	S1: `a[i]`	$\longrightarrow$	S2: `a[i]`
true data dependence	S1: `a[i]`	$\longrightarrow$	S3: `a[i]`
anti-dependence	S1: `b[i]`	$\longrightarrow$	S2: `b[i]`
output dependence	S3: `a[i+1]`	$\longrightarrow$	S1: `a[i]` (loop-carried)

(b) To parallelize the loop, we must remove any cycles of dependencies. The cycle here is on `a[i]`, from S1 to S3 within the loop and then from S3 to S1 across the loop. Notice that each element of `a[]` gets written twice (except for the first and last elements). By removing one of these writes (which happens to be part of the cycle), we can break the cycle.

The rewritten loop is:

```
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];
```

4.10

We start by calculating the CPI for the processor with a fixed 2-cycle branch penalty:

$$\begin{aligned}\mathrm{CPI}_{\mathrm{fixed}} &= \mathrm{CPI}_{\mathrm{base}} + (\text{branch frequency}) * (\text{branch penalty}) \\ &= 1 + (0.15) * 2 \\ &= 1 + 0.3 = 1.3\end{aligned}$$

Now, consider the branch penalty with the BTB:

$$\begin{aligned}\text{branch penalty with BTB} &= \text{penalty on BTB hit but mispredict} + \text{penalty on BTB miss} \\ &= (\text{BTB hit rate})(\text{fraction mispredicted})(\text{penalty cycles on mispredict}) + \\ &\quad (1 - \text{BTB hit rate})(\text{penalty cycles on BTB miss}) \\ &= (0.90)(0.10)(4) + (0.10)(3) = 0.36 + 0.3 = 0.66\end{aligned}$$

Then:

$$\frac{t(\text{fixed 2-cycle branch penalty})}{t(\text{with BTB})} = \frac{CPI(\text{fixed 2-cycle branch penalty})}{CPI(\text{with BTB})}$$

$$= \frac{1.3}{\text{CPI}_{\text{base}} + \text{branch stall CPI with BTB}}$$

$$= \frac{1.3}{1 + (0.15 \text{ branches/instr})(\text{branch penalty with BTB})}$$

$$= \frac{1.3}{1 + (0.15)(0.66)} = 1.18$$

So the processor with the BTB is 18% faster.

4.14

(c) We assume the read-operand stage of an instruction FP2 cannot overlap with the write-result stage of a previous instruction FP1 that writes any of FP2's operand registers, i.e. operands are read the cycle after they are written back to the register file. This includes data produced by loads. We also assume that there are 4 cycles spent in the execute stage for any FP ALU operation, and 1 cycle for all integer and load/store operations.

The following table describes the instruction status at the time when the SGTI reaches write-result:

Instruction		FP Unit & Res. St.	cycles in				Comments
			IS	RO	EX	WR	
`LD`	`F2, 0(R1)`	INT	1	2	3	4	
`MULTD`	`F4, F2, F0`	MUL1	2	5	6–9	10	waits for LD
`LD`	`F6, 0(R2)`	INT	5	6	7	8	issue in 5 due to struc haz
`ADDD`	`F6, F4, F6`	ADD	9	11	12–15	16	issue stalled to prevent WAW
`SD`	`0(R2), F6`	INT	10	17	18	19	waits for ADDD result
`ADDI`	`R1, R1, #8`	INT	20	21	22	23	issue in 20 due to struc haz
`ADDI`	`R2, R2, #8`	INT	24	25	26	27	issue in 24 due to struc haz
`SGTI`	`R3, R1, done`	INT	28	29	30	31	issue in 28 due to struc haz

(d) Here, we assume that a dependent instruction cannot start executing until the cycle after the producing instruction has completed its write stage (see page 259). Only one instruction can write the CDB each cycle. `SD` instructions do not write to the CDB. Finally, we assume that the integer unit is pipelined with a repeat rate of 1, and there are enough reservation stations so that no operation has to wait to issue.

The following tables describes the instruction and reservation station status at the time when the SGTI reaches write-result:

Instruction Status

Instruction		FP Unit & Res. St.	cycles in			Comments
			IS	EX	WR	
`LD`	`F2, 0(R1)`	LOAD1	1	2	3	
`MULTD`	`F4, F2, F0`	MUL1	2	4–7	8	waits for `LD`
`LD`	`F6, 0(R2)`	LOAD2	3	4	5	
`ADDD`	`F6, F4, F6`	ADD1	4	9–12		waits for `MUL`
`SD`	`0(R2), F6`	STORE1	5			waits for `ADDD`
`ADDI`	`R1, R1, #8`	INT1	6	7	9	CDB conflict on WR
`ADDI`	`R2, R2, #8`	INT2	7	8	10	CDB conflict on WR
`SGTI`	`R3, R1, done`	INT3	8	10	11	waits for first ADDI

Reservation Stations						
Resv. 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

The software-pipelined loop in the text (with the 1-cycle ADDD) is not sufficient when the ADDD has a 5-cycle latency, since there is a data dependency stall between the ADDD in one iteration of the loop and the SD in the next. To avoid this stall, we need to have more independent instructions to schedule between the ADDD and the SD. Since the loop iterations are independent, we unroll the (original) loop once, introducing two new registers (F6 and F8) to hold the data for that iteration. The unrolled loop is:

```
Loop2:  LD      F0,0(R1)
        ADDD    F4,F0,F2
        SD      0(R1),F4
        LD      F8,-8(R1)
        ADDD    F6,F8,F2
        SD      -8(R1),F6
        SUBI    R1,R1,#16
        BNEZ    R1,Loop2
```

Just unrolling doesn't remove the stalls, since the stores still immediately follow the adds. To remove the stalls, we need to software pipeline the loop. We start by unrolling a few iterations manually, then we select instructions (marked with arrows, below) to combine into the final loop (note: there are several ways to do this; only one is shown below):

```
Iteration i:    LD      F0,0(R1)
                ADDD    F4,F0,F2
                SD      0(R1),F4
                LD      F8,-8(R1)
                ADDD    F6,F8,F2
                SD      -8(R1),F6       <===
Iteration i+1:  LD      F0,0(R1)
                ADDD    F4,F0,F2
                SD      0(R1),F4
                LD      F8,-8(R1)
                ADDD    F6,F8,F2        <===
                SD      -8(R1),F6
Iteration i+2:  LD      F0,0(R1)
                ADDD    F4,F0,F2
                SD      0(R1),F4
                LD      F8,-8(R1)       <===
                ADDD    F6,F8,F2
                SD      -8(R1),F6
Iteration i+3:  LD      F0,0(R1)
                ADDD    F4,F0,F2
                SD      0(R1),F4        <===
```

```
                LD      F8,-8(R1)
                ADDD    F6,F8,F2
                SD      -8(R1),F6
Iteration i+4:  LD      F0,0(R1)
                ADDD    F4,F0,F2        <===
                SD      0(R1),F4
                LD      F8,-8(R1)
                ADDD    F6,F8,F2
                SD      -8(R1),F6
Iteration i+5:  LD      F0,0(R1)        <===
                ADDD    F4,F0,F2
                SD      0(R1),F4
                LD      F8,-8(R1)
                ADDD    F6,F8,F2
                SD      -8(R1),F6
```

By placing these selected instructions into a loop, we end up with the following stall-free software-pipelined loop (ignoring startup and cleanup code):

```
Loop3:  SD      -8(R1),F6       ; stores into M[i-1]
        ADDD    F6,F8,F2        ; adds to M[i-3]
        LD      F8,-40(R1)      ; loads M[i-5]
        SD      -48(R1),F4      ; stores into M[i-6]
        ADDD    F4,F0,F2        ; adds to M[i-8]
        LD      F0,-80(R1)      ; loads M[i-10]
        SUBI    R1,R1,#16
        BNEZ    R1,Loop3
```

This loop is stall-free since there are 6 instructions between each `ADDD` and the use of its data, so the `ADDD` must complete before its data is needed. Of course, the adder must be pipelined (or replicated) for this technique to work.

The above solution is not unique; for example, another stall-free version is:

```
Loop4:  SD      40(R1),F8       ; stores into M[i+5]
        SD      32(R1),F4       ; stores into M[i+4]
        ADDD    F8,F6,F2        ; adds to M[i+3]
        ADDD    F4,F0,F2        ; adds to M[i+2]
        LD      F6,8(R1)        ; loads M[i+1]
        LD      F0,0(R1)        ; loads M[i]
        SUBI    R1,R1,#16
        BNEZ    R1,Loop4
```

4.24

A simple solution to this problem is to split the clock cycle into two halves, such that the instruction is issued to (entered in) the reservation station on the first half of the cycle, and the CDB tag match and data transfer is performed on the second half of the cycle. This is similar to the mechanism used in the simple 5-stage DLX pipeline to avoid the need for forwarding when a register is being read and written in the same pipeline cycle. One downside of this approach is that it may negatively impact the machine's cycle time if the reservation station issue time or the CDB tag match time is longer than half of the cycle time.

Another approach is to use a bit of extra state hardware to pipeline the CDB check across multiple cycles (much like pipelined writes in a CPU cache like the Alpha 21064's). During each cycle, the tags and results broadcast on the CDB are latched into a bit of extra state in each reservation station (or block of reservation stations). During the next cycle, the tags of any instructions that were issued to the reservation station during the previous cycle are compared with the value latched in the previous cycle, and the operand information is updated if the tags match. This approach uses more hardware, but has a potentially smaller impact on cycle time. Note that there are some subtleties to this approach, notably that the CDB matches from the previous cycle must take precedence over any matches during the current cycle.

B.2

Note: in this problem, I assumed that only one instruction could issue per cycle, even if multiple independent instructions were contained within a convoy.

We have the following vector code run on a 200 MHz DLXV with VL=64:

```
LV      V1,Ra
MULTV   V2,V1,V3
ADDV    V4,V1,V3
SV      Rb,V2
SV      Rc,V4
```

(a) We want the number of chimes and number of cycles for this sequence, assuming one memory pipeline and no chaining.

First, we group the instructions into convoys:

The `LV` goes into the first convoy. The `MULTV` depends on `V1` from the `LV`, so it goes into the second convoy. The `ADDV` is independent of the `MULTV`, so it also occupies the second convoy. The first `SV` depends on the `MULTV`, so it goes into the third convoy, and the last `SV` requires a fourth convoy to avoid a structural hazard with the previous `SV`.

So:

Convoy 1: `LV`
Convoy 2: `MULTV ADDV`
Convoy 3: `SV`
Convoy 4: `SV`

Thus, this sequence takes 4 chimes.

Now we will include start-up overhead: 12 cycles for a LV/SV, 7 cycles for a MULTV, and 6 cycles for an ADDV. n is the vector length. We assume that instructions within a convoy issue in separate cycles.

Convoy	Start Time	First Result Time	Last Result Time
1. `LV`	0	12	$11+n$
2. `MULTV ADDV`	$12+n$	$12+n+7$	$18+2n$
3. `SV`	$19+2n$	$19+2n+12$	$30+3n$
4. `SV`	$31+3n$	$31+3n+12$	$42+4n$

Note that the `MULTV` starts on cycle $12+n$ and produces its first result on cycle $19+n$. The `ADDV` starts on cycle $13+n$ and produces its first result on cycle $19+n$. The total number of cycles is $43+4n$, or 299 cycles for a vector length of 64. Since 64 results are produced, this works out to $4+(43/64)=4.67$ clocks per result.

(b) Now we add chaining. The convoys now look like:

Convoy 1: `LV MULTV ADDV`
Convoy 2: `SV`
Convoy 3: `SV`

The two stores still go in separate convoys to avoid structural hazards due to the fact that DLXV has only one load/store unit.

Note that both the `MULTV` and `ADDV` in the first convoy must wait for the startup time of the `LV`, since they both depend on the `LV`'s result. However, since the `MULTV` and `ADDV` are independent, they can issue one cycle apart.

For 64-element vectors, the total number of clock cycles is now:

$(\texttt{LV}_{\text{startup}} + \texttt{MULTV}_{\text{startup}} + 64) +$ /* convoy 1 */
$(\texttt{SV}_{\text{startup}} + 64) +$ /* convoy 2 */
$(\texttt{SV}_{\text{startup}} + 64)$ /* convoy 3 */
$= 12 + 7 + 64 + 12 + 64 + 12 + 64 = 235$ cycles

which means that there are $235/64 = 3.67$ clocks per result.

(c) Now we assume there are 3 memory pipelines, chaining, and no bank conflicts. We can thus move all the instructions into one convoy:

Convoy 1: `LV MULTV ADDV SV SV`

As before, both the `MULTV` and `ADDV` have to wait for the `LV` startup. The first `SV` has to wait for the `MULTV` startup, and the second `SV` has to wait for the `ADDV` startup. The second `SV` is delayed one cycle since both the `MULTV` and `ADDV` produce their first results in the same cycle, and we can't initiate both `SV`s during the same cycle.

So the critical timing path is:

$\texttt{LV}_{\text{startup}} + \texttt{MULTV}_{\text{startup}} + 1 + \texttt{SV}_{\text{startup}} + 64$
$= 12 + 7 + 1 + 12 + 64$
$= 96$ cycles.

So there are $96/64 = 1.5$ clocks per result.