3.7

3.7 [15] <3.4> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 3.48 shows example code that we would like our hardware to rename. A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them T registers, and assume that there are 64 of them, T0 through T63) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the src (source) register designation, and the value in the table is the T register of the last destination that targeted that register. (Think of these table values as producers, and the src registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 3.48. Every time you see a destination register in the code, substitute the next available T, beginning with T9. Then update all the src registers accordingly, so that true data dependences are maintained. Show the resulting code. (*Hint:* see Figure 3.49.)

```
Loop:   fld      f2,0(Rx)
I0:     fmul.d   f5,f0,f2
I1:     fdiv.d   f8,f0,f2
I2:     fld      f4,0(Ry)
I3:     fadd.d   f6,f0,f4
I4:     fadd.d   f10,f8,f2
I5:     sd       f4,0(Ry)
```

Figure 3.48 Sample code for register renaming practice.

```
I0:     fld      T9,0(Rx)
I1:     fmul.d   T10,F0,T9
...
```

Figure 3.49 Expected output of register renaming.

```
Loop: fld T9,0(Rx)
I0: fmul.d T10,f0,T9
I1: fdiv.d T11,f0,T9
I2: fld T12,0(Ry)
I3: fadd.d T13,f0,T12
I4: fadd.d T14,T11,f2
I5: sd T15,0(Ry)
```

f2 T9

f5 T10

f8 T11

f4 T12

f6 T13

f10 T14

f4 T15

3.8

指令改为：

```
I0:   SUBD    F1,F2,F3
I1:   ADDD    F4,F1,F2
I2:   MULTD   F6,F4,F1
I3:   DIVD    F0,F2,F6
```

3.8 [20] <3.4> Exercise 3.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it, but superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A SimpleScalar processor would therefore look up both src register mappings for each instruction and allocate a new dest mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any dest-to-src relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 3.50. Assume that we would like to simultaneously

```
Loop:   fld      f2,0(Rx)
I0:     fmul.d   f5,f0,f2
I1:     fdiv.d   f8,f0,f2
I2:     fld      f4,0(Ry)
I3:     fadd.d   f6,f0,f4
I4:     fadd.d   f10,f8,f2
I5:     sd       f4,0(Ry)
```

Figure 3.48 Sample code for register renaming practice.

```
I0:     fld      T9,0(Rx)
I1:     fmul.d   T10,F0,T9
...
```

Figure 3.49 Expected output of register renaming.

```
I0:     fmul.d   f5,f0,f2
I1:     fadd.d   f9,f5,f4
I2:     fadd.d   f5,f5,f2
I3:     fdiv.d   f2,f9,f0
```

Figure 3.50 Sample code for superscalar register renaming.

rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 3.51 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code shown in Figure 3.50. Assume the table starts out with every entry equal to its index (T0=0; T1=1, ...) (Figure 3.51).

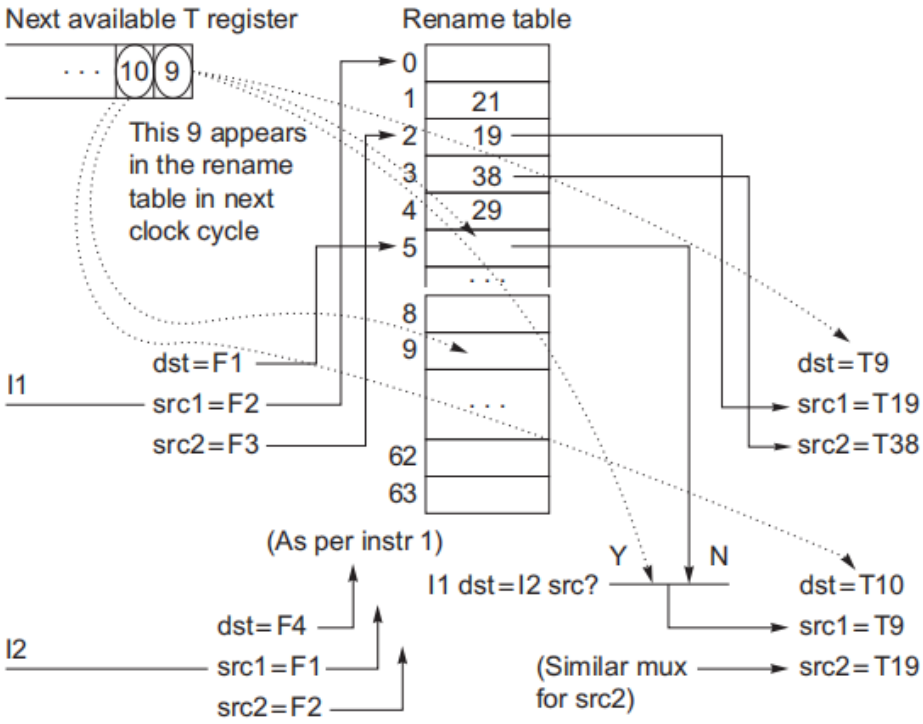

Figure 3.51 Initial state of the register renaming table.

```
I0:   SUBD    F1,F2,F3
I1:   ADDD    F4,F1,F2
I2:   MULTD   F6,F4,F1
I3:   DIVD    F0,F2,F6
```

第一次issue I0和I1，此时可用的Tteg是 T10 T9

f1 f2 f3

f4 f1 f2　同时分配给两个rd

第二次issue同理，而且要把真依赖标记清楚

变为

T9 F2 F3

T10 T9 F2

T11 T10 F2

T12 F2 T11

3.11

指令改为

```
Loop:  LW      R3,0(R0)
       LW      R1,0(R3)
       ADDI    R1,R1,#1
       SUB     R4,R3,R2
       SW      R1,0(R3)
       BNZ     R4, Loop
```

3.11 [10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 3.53. All ops are one cycle except LW and SW, which are 1+2 cycles, and branches, which are 1+1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?

b. [10] <3.3> Assume a static branch predictor, capable of recognizing a backward branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

```
Loop:   lw      x1,0(x2);     lw      x3,8(x2)
        <stall>
        <stall>
        addi    x10,x1,1;     addi    x11,x3,1
        sw      x1,0(x2);     sw      x3,8(x2)
        addi    x2,x2,8
        sub     x4,x3,x2
        bnz     x4,Loop
```

Figure 3.52 Sample VLIW code with two adds, two loads, and two stalls.

```
Loop:   lw x1,0(x2)
        addi    x1,x1, 1
        sw      x1,0(x2)
        addi    x2,x2,4
        sub     x4,x3,x2
        bnz     x4,Loop
```

Figure 3.53 Code loop for Exercise 3.11.

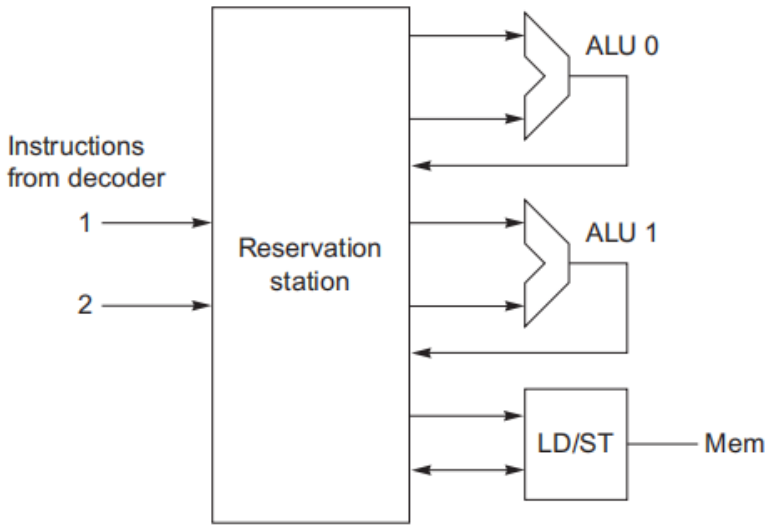

Figure 3.54 Microarchitecture for Exercise 3.12.

c. [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

```
Loop:   LW      R3,0(R0)
        LW      R1,0(R3)
        ADDI    R1,R1,#1
        SUB     R4,R3,R2
        SW      R1,0(R3)
        BNZ     R4, Loop
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
LW R3,0(R0)	F	D	E	M	.	.	W											
LW R1,0(R3)		F	D	.	.	.	E	M	.	.	W							
ADDI R1,R1,1			F	.	.	.	D	.	.	.	E	M	W					
SUB R4,R3,R2							F	.	.	.	D	E	M	W				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
SW R1,0(R3)											F	D	E	M	.	.	W	
BNZ R4,LOOP												F	D	E	.	.	M	W
LW R3,0(R0)																	F	D

a)

从BNZ 的fetch到下一个循环开始的lw的fetch 之间一共是五个时钟周期（或者这么算 F D E E E 一共需要五个阶段才能拿到结果），所以overhead就是5 cycles

b)

如果使用静态预测器，这里是指能够在decode阶段就知道跳转到哪里，此时仍然需要 F D阶段的overhead

c)

动态预测器能够直接提供跳转的地址，如果猜测正确，那么跳转指令将不会有任何的开销

3.13

3.13 [25] <3.7, 3.8> In this exercise, you will explore performance trade-offs between three processors that each employ different types of multithreading (MT). Each of these processors is superscalar, uses in-order pipelines, requires a fixed three-cycle stall following all loads and branches, and has identical L1 caches. Instructions from the same thread issued in the same cycle are read in program order and must not contain any data or control dependences.

- Processor A is a superscalar simultaneous MT architecture, capable of issuing up to two instructions per cycle from two threads.
- Processor B is a fine-grained MT architecture, capable of issuing up to four instructions per cycle from a single thread and switches threads on any pipeline stall.

- Processor C is a coarse-grained MT architecture, capable of issuing up to eight instructions per cycle from a single thread and switches threads on an L1 cache miss.

Our application is a list searcher, which scans a region of memory for a specific value stored in R9 between the address range specified in R16 and R17. It is parallelized by evenly dividing the search space into four equal-sized contiguous blocks and assigning one search thread to each block (yielding four threads). Most of each thread's runtime is spent in the following unrolled loop body:

```
loop: lw x1,0(x16)
      lw x2,8(x16)
      lw x3,16(x16)
      lw x4,24(x16)
      lw x5,32(x16)
      lw x6,40(x16)
      lw x7,48(x16)
      lw x8,56(x16)
      beq x9,x1,match0
      beq x9,x2,match1
      beq x9,x3,match2
      beq x9,x4,match3
      beq x9,x5,match4
      beq x9,x6,match5
      beq x9,x7,match6
      beq x9,x8,match7
      DADDIU x16,x16,#64
      blt x16,x17,loop
```

Assume the following:

- A barrier is used to ensure that all threads begin simultaneously.
- The first L1 cache miss occurs after two iterations of the loop.
- None of the BEQAL branches is taken.
- The BLT is always taken.
- All three processors schedule threads in a round-robin fashion.

Determine how many cycles are required for each processor to complete the first two iterations of the loop.

[25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can

处理器A：

一次同时能够发射两个线程，此处有四个线程，总计8*4 = 32个lw指令和 32 个跳转指令

A开始第一个cycle 线程01 发射第一个lw x1 ， E M . . . 总共需要五个cycle，所以在第五个cyle才能发射 lw x2

第二个cycle 线程23 发射，同样要等到第六个cycle才能发射 lw x2

这里每次的跨度是4cycle，

线程01 会在1 +32cycle之后发射beq指令

线程23 会在2 + 32cycle后发射beq指令，

beq和lw一样都是需要3个stall，

线程01 会在1 +32 + 32cycle之后发射ADD指令

线程23 会在2 +32 + 32cycle之后发射ADD指令

线程01 会在1 +32 + 32 + 2cycle之后发射blt指令，延迟为三个stall，所以线程01同时结束于1 +32 + 32 + 6=70cycle 结束 此时可以去开始下一轮循环的Fetch了，所以一个循环需要70cycle

处理器B：

每个线程一次能够做四条指令

第一个cycle 线程0 发射了lw x1,x2,x3,x4 四个指令

第二个cycle 线程0还在stall，填入线程1的四个指令

第三个cycle填入线程2的四个指令

第四个cycle填入线程3的四个指令

在第五个cycle，线程0的四个指令都已经做完

以此类推，线程0将会在1 + 4*（8/4）个cycle之后发射beq指令，但是只能发射一个，并且stall三个cycle，这三个cycle正好可以让其余的三个线程分别issue，形成,负载为1/4的流水

线程0将会在1 + 8 + 4*8 = 41 cycle之后发射add指令，然后在45cycle时候发射blt

线程123分别在46 47 48 发射blt指令，等到线程3结束 可以开始新一轮的循环

整个循环消耗48cycles

3.16

3.16 [10] <3.4> Tomasulo's algorithm has a disadvantage: only one result can compute per clock per CDB. Use the hardware configuration and latencies from the previous question and find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

假设mul x1,x2,x4 在cycle 1发射，在cycle 10写CDB

此时连续发射多个add指令，分别在4 6 8 10 写CDB，最后会发生CDB的写冲突

mul x1,x2,x4 10

add x1,x2,x2 4

add x1,x2,x2 6

add x1,x2,x2 8

add x1,x2,x2 10(stall)

add x1,x2,x2 11(stall)