a.

```
                ADDI R1,R0, #2000 // base address of S
                ADDI R2,R0, #3000 // base address of i
                ADDI R3,R0, #1000 // base address of A[0]
      LOOP:     LW R4,0(R3) // A[i]
                LW R5,0(R1) // S
                LW R6,0(R2) //i
                ADD R5,R5,R4
                SW R5,0(R1)
                ADDI R3,R3,#4
                ADDI R6,#1 // i <- i+1
                SW R6,0(R2)
                SLTI R7,R6,#100
                BNEZ R7,LOOP
```

Estimated execution time:
Instructions before the start of the loop take 3 cycles.
Each iteration except for the last one takes (2+2+2+1+1+1+1+1+1+3)=15 cycles, for a total of 99 iterations. The last loop iteration takes 13 cycles. Therefore the estimated execution time is 3+99*15+13 = 1501 cycles.

b.

```
                ADD R1,R0,R0 // S =0;
                ADDI R2,R0,#100
                ADDI R3,R0,#1000
                ADDI R5,R0,#2000
      LOOP:     LW R4,0(R3) // R4<- mem[R3]
                ADD R1,R1,R4 // S=S+A[i]
                ADDI R3,R3,#4
                SUBI R2,R2,#1
                BNEZ R2,LOOP
                SW R1,0(R5)
```

Estimated execution time:
To initialize variables before the loop starts, 4 ALU instructions execute in 4 cycles.
For 99 iterations, each iteration takes (2+1+1+1+3) =8 cycles. The last iteration takes (2+1+1+1+1) = 6 cycles.
Finally, the store instruction takes 1 cycle.
Therefore, estimated execution time is 4 + 99*8 + 6+1 = 803 cycles.

Problem 3.4

a. In the machine with no forwarding at all instructions must wait for their inputs until they have been written back to register. It takes 3 clocks for the result of an instruction in EX to exit WB. Thus the latency of operation of all instructions writing in a register is 3 clocks and the compiler must insert NOOPs accordingly.

```
      SEARCH:   LW R5,0(R3)        /I1  Load item
                NOOP
                NOOP
                NOOP
                SUB R6,R5,R2       /I2  Compare with key
                NOOP
                NOOP
                NOOP
                BNEZ R6,NOMATCH    /I3  Check for match
                ADDI R1,R1,#1      /I4  Count matches
      NOMATCH:  ADDI R3,R3,#4      /I5  Next item
                NOOP
                NOOP
                NOOP
                BNE R4,R3,SEARCH   /I6  Continue until all items
```

b.Without forwarding but with a hazard detection unit in ID, an instruction which is dependent on a prior instruction should be stalled in ID until the result from the prior instruction is written back to register. The latency of operation of all instructions is 3 just as in part a of this problem.

b.1) First assume a match.

```
      SEARCH:   LW R5,0(R3)        (1)
                SUB R6,R5,R2       (4)
                BNEZ R6,NOMATCH    (4)
                ADDI R1,R1,#1      (1)
      NOMATCH:  ADDI R3,R3,#4      (1)
                BNE R4,R3,SEARCH   (4+2)
```

The numbers between parentheses are the number of cycles associated with each instruction. Each instruction spends one cycle in the ID stage plus additional stall cycles due to the latency of prior instructions it depends on. In the case of a branch, the number of instructions flushed when the branch is taken must be added. For example, BNE spends 4 cycles in ID due to data dependency and 2 additional cycles are lost due to flushing when the branch is taken in EX.
Summing up all delays, it takes (1+4+4+1+1+4+2)= 17 cycles to execute one iteration of the loop.

b.2) On no match, BNEZ (I3) is taken.

```
      SEARCH:   LW R5,0(R3)        (1)
                SUB R6,R5,R2       (4)
                BNEZ R6,NOMATCH    (4+2)
      NOMATCH:  ADDI R3,R3,#4      (1)
                BNE R4,R3,SEARCH   (4+2)
```

After the BNEZ instruction is taken in ID, 2 instructions in IF and ID are flushed.
It takes 18 cycles to execute one iteration of the loop.

c. With register forwarding and a hazard detection unit, an instruction which is waiting for its operand from a previous instruction in ID can get the value when the previous instruction reaches WB. Therefore, the latency of operation of instructions writing to a register is 2 instead of 3.

c.1) On a match:

```
      SEARCH:   LW R5,0(R3)        (1)
                SUB R6,R5,R2       (3)
                BNEZ R6,NOMATCH    (3)
                ADDI R1,R1,#1      (1)
      NOMATCH:  ADDI R3,R3,#4      (1)
                BNE R4,R3,SEARCH   (3+2)
```

It takes 14 cycles to execute one iteration of the loop.

c.2) On no match:

```
      SEARCH:   LW R5,0(R3)        (1)
                SUB R6,R5,R2       (3)
                BNEZ R6,NOMATCH    (3+2)
      NOMATCH:  ADDI R3,R3,#4      (1)
                BNE R4,R3,SEARCH   (3+2)
```

It takes 15 cycles to execute one iteration of the loop.

d. With full forwarding and a hazard detection unit, the latency of all instructions is 0 except for loads, and so instructions in the code have no stall in ID except for the SUB instruction.

d.1) On a match

```
SEARCH:    LW R5,0(R3)        (1)
           SUB R6,R5,R2       (2)
           BNEZ R6,NOMATCH    (1)
           ADDI R1,R1,#1      (1)
NOMATCH:   ADDI R3,R3,#4      (1)
           BNE R4,R3,SEARCH   (1+2)
```

It takes 9 cycles to execute one iteration of the loop.

d.2) On no match

```
SEARCH:    LW R5,0(R3)        (1)
           SUB R6,R5,R2       (2)
           BNEZ R6,NOMATCH    (1+2)
NOMATCH:   ADDI R3,R3,#4      (1)
           BNE R4,R3,SEARCH   (1+2)
```

It takes 10 cycles to execute one iteration of the loop.

e. A basic block cannot have a branch or jump except at its bottom. Moreover, no instruction inside of it can be the target of a branch or a jump. Hence, there are 3 basic blocks in the given code (assuming no other instruction branches into the code).
Basic block 1: I1, I2, and I3
Basic block 2: I4
Basic block 3: I5 and I6

It is not possible to save cycles by local optimizations because the sizes of basic blocks are too small and we do not have enough instructions for local optimizations. In general it is not safe to move an instruction outside of its basic block. However, the program would still execute correctly if I5 was moved before BNEZ because I5 is always executed, whether or not the branch is successful. However global optimizations will be necessary to enable that move.

f. Yes. For example the LW and SUB for two consecutive execution of the loop can be moved up after register renaming, thus avoiding the stall on the SUB instruction. However to be able to do that global scheduling is needed because the LW and SUB must be moved up across a branch instruction.

With a delayed branch instruction the pointer to the next item could be updated in the delay slot of I3, thus saving one cycle in the case of no match.

```
SEARCH:    LW R5,0(R3)        /I1  Load item
           SUB R6,R5,R2       /I2  Compare with key
           DBNEZ R6,NOMATCH   /I3  Check for match
           ADDI R3,R3,#4      /I5  Next item
           ADDI R1,R1,#1      /I4  Count matches
NOMATCH:   BNE R4,R3,SEARCH   /I6  Continue until all items
```

Problem 3.5.

a. The goal is to balance the functions in the pipeline stages so that the delays in all stages are as close as possible. One possible partition is shown in Figure 1.

b.
Instruction latency in non-pipelined implementation: PC delay(1ns) + ICache(6ns) + Itype Decode(3.5ns) + Src. Decode(2.5ns) + Reg. Read(4ns) + Mux(1ns) + ALU(6ns) + Reg. Write(4ns) = 28 ns.
In the pipelined implementation, there are 5stages and each stage takes 7.5ns (the bottleneck is the

Trace 1: 14/10 = 1.4
Trace 2: 22/13 = 1.69
Trace 3: 20/12 =1.67
Trace 4: 18/11 = 1.64
Trace 5: 18/11 = 1.64
Trace 6: 16/10 = 1.6
Trace 7: 14/9 = 1.56

Problem 3.26

a. The code for processing each strip of 64 components is given below:

```
      LOOP:  L.V    V1,0(R2),R6          /load X; R6 contains the stride of 1.
             L.V    V2,0(R3),R6          /load Y; 0(R3) is the base address of Y
             MUL.V  V3,V2,V1             /Multiply two vector registers
             ADD.V  V4,V4,V3             /Partial sums accumulate in V4
             ADDI   R2,R2,#64            /This assumes that memory
             ADDI   R3,R3,#64            /addresses point to vector elements
             ADDI   R4,R4,#1
             BNE    R4,R5,LOOP
```

Partial vector sums accumulate in V4. At the end we simply have to add the components of V4. To simplify we have assumed that memory addresses point to vector elements. If vector components have 8 bytes (double precision) and memory is byte-addressable, then the increments on the address registers should be 512.
Given that vector operations are chained, one iteration of the loop takes:
Tite = latency(L.V) + latency(MUL.V)+ latency(ADD.V) +(V.L)-1 = 30+10+5+64-1 = 108 cycles.
Since the loop has to iterate 16 times (16=1024/64), the total number of cycles taken by a dot-product is 108*16 =1728 cycles (ignoring the scalar phase at the end).

b.
For the multiplication of two matrices, a component of the result matrix is obtained by a dot-product of two vectors. Therefore, we need 1024*1024= 1048576 dot-products to multiply two 1024*1024 matrices. The matrix multiply takes 1728*1048576 = $1.812*10^9$ cycles.

c. Unrolling the vector loop twice (after register renaming):

```
      LOOP:  L.V    V1,0(R2),R6          /load X
             L.V    V2,0(R3),R6          /load Y
             MUL.V  V3,V2,V1             /Multiply two vector registers
             ADD.V  V4,V4,V3             /Partial-sum
             ADDI   R2,R2, #64
             ADDI   R3,R3, #64
             L.V    V5, 0(R2), R6        /load X
             L.V    V6, 0(R3), R6        /load Y
             MUL.V  V7,V5,V6             /Multiply two vector registers
             ADD.V  V8,V8,V7             / partial-sum
             ADDI   R2,R2,#64
             ADDI   R3,R3,#64
             ADDI   R4,R4,#2
             BNE    R4,R5,LOOP
```

Load and multiplication on each strip of 64 components of the vector are chained and run in parallel. Partial sums are written in two different vector registers, V4 and V8. The scalar processor accumulates the components of the partial sum vectors at the end. Ignoring the scalar phases, the one iteration of the unrolled loop takes 108 cycles. Hence we can compute 128 elements of the vector in each iteration, and the number of iterations for the vector with size of 1024 is halved and it is 8(=1024/128).Therefore, the total number of cycle to execute the dot-product is 108 * 8 = 864