CSE 112 Lab 3

2.1(1)

```
/* Preamble */
.begin:
        mLoadIR
        mdecode
        madd pc, 4
        mswitch

mmovi regSrc, 2, <read>                 /* read from r2 */
mmov A, regVal                          /* A = r2 */
mmovi B, 2, <div>                       /* move in B the value in r2/2 */
mmov regData, aluResult                 /* write in r3 the value in B */
mmovi regSrc, 3, <write>                /* r3 = r2/2 */

mmovi A, 1                              /* move in A the immediate value of 1 */
mmovi B, 1                              /* move in B the immediate value of 1 */
mmovi regSrc, 0
mmov regData, 1, <write>                /* r0=1 */


/* A=1 and B=1 */

.loop:
        mmov B, B, <multiply>           /* aluResult = A*B */
        mmov B, aluResult               /*B = aluResult*/
        mmovi regSrc, 2, <read>
        mmov A, regVal                  /* A=r2 */
        mbeq A, B, .true                /* if B==A return true */
        mmov regSrc, 0
        mmovi regData, 1, <add>         /* r0+=1 */
        mmov regSrc, 0, <read>
        mmov A, regVal                  /* A = r0 */
        mmovi regSrc, 3, <read>
        mmov B, regVal                  /* B = r3 */
        mbeq A, B, .false               /* if the counter has reached to half
                                                of the value then return false. */
        mmov B, A                       /* B = A */
        mb .loop

.true:
        mmovi regSrc, 0
        mmovi regData, 1, <write>
        mb .begin

.false:
        mmovi regSrc, 0
        mmovi regData, 0, <write>
        mb .begin
```

The explanation to the code is as following:

I have written preamble which has to be written for the execution of any code in microassembly. Next, the value from r2 is read and put in A. r2 contains the input which number is to be checked whether it is a perfect square or not. I move in B r2/2 which is n/2 to which I will check the square root of the number as the square root of any number lies within n/2. So, to start with I have put 1 to both A and B. I calculate A*B in every loop and put it in the aluResult. I check if aluResult is equal to the input. If it is equal then the input is a perfect square else, I jump to the next iteration. In every iteration I check if the number of iteration is equal to n/2, if it is equal I break the loop to .false label. For false and true I have created two different labels which consist the Boolean value stored in r0. For true, 1 is stored in r0 and for false 0 is stored. Then the program jumps to .begin to carry on execution.

2.1(2)

```
/* value stored in r2
   the number of bits to which the palindrome needs to be checked is in r0
   boolean result stored in r0 */

.begin:
        mLoadIR
        mdecode
        madd pc, 4
        mswitch

mmovi regSrc, 2, <read>              /* read from r2 */
mmov A, regVal                       /* A = r2 */
mmovi B, 1, <lsl>                    /* aluResult = A shifted left by 1 bit so that
                                        it can become normal when it gets into loop */

mmovi regSrc, 0, <read>              /* read from r0 */
mmov sr1, regVal                     /* sr1 = r0 */
madd sr1, -1                         /* value decremented by 1 */
mmov regData, sr1, <write>           /* r0-=1, the places that need to be
                                        shifted right */
mmovi regSrc, 0, <read>              /* read from r0 */
mmov sr2, regVal                     /* sr2 = r0 */
mmov sr2, 2, <div>                   /* sr2 = r0/2 */

mmovi regSrc, 5
mmov regData, -1, <write>            /* r5 is initialised with -1 to keep count
                                         of bits checked */



.loop:
        mmovi regSrc, 5, <read>      /* value of r5 is read */
        mmov B, regVal               /* B = r5 */
        madd B, 1                    /* B+=1 */
```

```
        mmov aluResult, B            /* aluResult = B */
        mmov regData, aluResult      /* value put in regData */
        mmovi, regSrc, 5, <write>    /* r5 = aluResult */
        mmovi regSrc, 2, <read>      /* read from r2 */
        mmov A, regVal               /* A = r2 */
        mmovi B, 1                   /* B = 1 */
        mmov B, B, <lsr>             /* aluResult = A right shifted by 1 bit */
        mmov A, aluResult            /* A = aluResult */
        mmov B, B, <and>             /* aluResult = A&B; B has the rightmost bit of input */
        mmov regData, aluResult      /* value put in regData */
        mmovi regSrc, 3, <write>     /* result is written in r3 */
        mmovi regSrc, 0, <read>      /* value is read from r0 */
        mmov B, regVal               /* B = r0 */
        mmovi regSrc, 2, <read>      /* read from r2 */
        mmov A, regVal               /* A = r2 */
        mmov B, B, <lsr>             /* aluResult = A shifted right by value stored in r0 */
        mmovi B, 1                   /* B = 1 */
        mmov A, aluResult            /* A = aluResult */
        mmov B, B, <and>             /* aluResult contains the leftmost bit of input */
        mmov regData, aluResult      /* value put in regData */
        mmovi regSrc, 4, <write>     /* result in written in r4 */
        mmovi regSrc, 5, <read>      /* value is read from r5 */
        mmov A, regVal               /* A = r5 */
        mbeq A, sr2, .true           /* if equal bits from either side are equal
                                        then it is a palindrome */
        mmovi regSrc, 0, <read>      /* value is read from r0 */
        mmov A, regVal               /* A = r0 */
        madd A, -1                   /* A-=1 */

        mmov aluResult, A            /* A is moved to aluResult */
        mmov regData, aluResult      /* value put in regData */
        mmovi regSrc, 0, <write>     /* r0 is decremented by 1 */
        mmovi regSrc, 3, <read>      /* r3 is read */
        mmov A, regVal               /* A = r3 */
        mmovi regSrc, 4, <read>       /* r4 is read */
        mmov B, regVal                /* B = r4 */
        mbeq A, B, .loop              /* jump to next iteration if the bits are equal */
        mb .false


.true:
        mmovi regSrc, 0
        mmovi regData, 1, <write>
        mb .begin

.false:
        mmovi regSrc, 0
        mmovi regData, 0, <write>
        mb .begin
```

Explanation to the code:

To execute this program I again write the preamble required for execution of any microassembly code. The number to be checked is in r2 so I put it in A and also shift it left by 1 bit because in loop to check the palindrome I will have to shift it right. So to preserve the original number I did this. Next, I use two scratch registers. Sr1 to store the number of bits to be checked for palindrome and sr2 to keep its halved value. The logic used in the loop works as follows: I check right most bit with the left most bit first. To get the leftmost bit I shift the value in A by required bits in every iteration and then AND it with 1. I do the same with rightmost bit and then store the results in r3 and r4. If r3 and r4 are equal I jump to

next iteration else, the program jumps to .false label. The number of bits to be shifted to right to check the leftmost bit is in kept in r0 and it is decremented by 1 in every iteration so that we can get the leftmost bit then right to it and so on. I check till half the string, if all are bits are equal then program outputs the result as a palindrome by jumping to .true label. This also works for odd length string as if the mid bit is not compared with anyone and if all the bits to its left and right are correspondingly equal then also it is a palindrome.

2.2(1)a.

```
mul r8, r9, r10
add r1, r2, r3
nop
sub r4, r1, r1
cmp r8, r9
beq .foo
nop
nop
```

In this code reordering is done to remove RAW dependency in the mul and cmp instructions. So we need to have three instructions between them. The sub instruction is an independent instruction as it is subtracting r1 from r1 so I have put add and sub between mul and cmp. Also, I have put one nop in between them. For the branch instruction, I have used two delay slots by putting two nops after that.

b. O represents bubble.

Let the reordered code be:

1. mul r8, r9, r10
2. add r1, r2, r3
3. cmp r8, r9
4. beq .foo
5. sub r4, r1, r1
6. nop

Cycle >	1	2	3	4	5	6	7	8	9	10	11	12
IF	1	2	3	4	4	4	5	O				
OF		1	2	3	3	3	4	5	O			
EX			1	2	O	O	3	4	5	O		
MA				1	2	O	O	3	4	5	O	
RW					1	2	O	O	3	4	5	O

The pipelining diagram will be as made above.

Since interlocking is a hardware technique to eliminate the pipelining hazards after reordering when the 3rd instruction will reach the OF stage it will stall it there as it will need the value of r8 to carry out the instruction. When the correct value will be available in cycle 6 it will then send the 3rd instruction to next stage. For the branch instruction there should be 2 instructions below it to eliminate the erroneous execution. So, one instruction is the sub instruction and for the 2nd instruction an nop is passed which appears as bubble in the pipeline diagram.

c.

1. add r1, r2, r3
2. sub r4, r1, r1
3. mul r8, r9, r10
4. cmp r8, r9
5. beq .foo
6. nop
7. nop

Cycle >	1	2	3	4	5	6	7	8	9	10	11
IF	1	2	3	4	5	O	O				
OF		1	2	3	4	5	O	O			
EX			1	2	3	4	5	O	O		
MA				1	2	3	4	5	O	O	
RW					1	2	3	4	5	O	O

Here, there is no need to reorder the code. Since sub is independent as explained in above part also and the only dependency is in mul and cmp which can be rectified by forwarding from MA to EX stage in cycle 6. Also, since the hardware will not know that 2 is an independent instruction we would need forwarding from MA to EX in 4th cycle. Moreover, for beq instruction we need to pass two nops after that which appears as bubble in the diagram.

2.2(2)a.

add r4, r3, r3

mul r8, r9, r10

div r8, r9, r10

nop

st r3, 10[r4]

ld r2, 10[r4]

nop

nop

nop

add r4, r2, r6

There is a RAW dependence between the first add and st instruction. To rectify this there must be three instructions in between them so I have inserted the mul and div instructions with an nop. Next, there is again RAW dependency in the ld and add instruction so inserted three nops to rectify.

b.

1. add r4, r3, r3
2. mul r8, r9, r10
3. div r8, r9, r10
4. st r3, 10[r4]
5. ld r2, 10[r4]
6. add r4, r2, r6

Cycle >	1	2	3	4	5	6	7	8	9	10	11	12	13	14
IF	1	2	3	4	5	5	6							
OF		1	2	3	4	4	5	6	6	6	6			
EX			1	2	3	O	4	5	O	O	O	6		
MA				1	2	3	O	4	5	O	O	O	6	
RW					1	2	3	O	4	5	O	O	O	6

When the 4th instruction reaches the OF stage, it should be stalled for 1 cycle as till the value of r4 will be written by instruction 1 at the RW stage. Now, the 6th instruction will need to be stalled for 3 cycles so that we get the correct value of r4.

c.

1. add r4, r3, r3
2. st r3, 10[r4]
3. ld r2, 10[r4]
4. mul r8, r9, r10
5. div r8, r9, r10
6. add r4, r2, r6

Cycle >	1	2	3	4	5	6	7	8	9	10
IF	1	2	3	4	5	6				
OF		1	2	3	4	5	**6**			
EX			1	**2**	**3**	4	5	6		
MA				**1**	2	3	4	5	6	
RW					**1**	2	**3**	4	5	6

The forwarding after reordering will be done in the manner as described in the above diagram. The 2nd instruction would need the value to write r3 to it so it will be forwarded in 4th cycle from MA to EX stage. Next, since the 3rd instruction also needs to write to r2, forwarding will be carried out in cycle 5. The 6th instruction needs value of r4 which would be finally forwarded from RW to OF stage in the 7th cycle.