```
// OutBinNibble.mal
//
// Note that this is nearly identical to the example
// given in Tanenbaum's book (Figure 4-17).
// Intrepretation of four additional opcodes (HALT, ERR, IN, OUT)
// have been added for completeness. Also, it interprets the opcode OUTBIN.
// Note:
//
// 1) SlashSlash-style ("//") comment characters have been added.
//
// 2) "nop" has been added as a pseudo-instruction to indicate that
//    nothing should be done except goto the next instruction.  It
//    is a do-nothing sub-instruction that allows us to have MAL
//    statements without a label.
//
// 3) instructions are "anchored" to locations in the control
//    store as defined below with the ".label" pseudo-instruction
//
// 4) a default instruction may be specified using the ".default"
//    pseudo-instruction.  This instruction is placed in all
//    unused locations of the control store by the mic1 MAL assembler.
//

// labeled statements are "anchored" at the specified control store address
.label      nop1        0x00
.label      bipush1     0x10
.label      ldc_w1      0x13
.label      iload1      0x15
.label      istore1     0x36
.label      rot1        0x20
.label      addd1       0x30
.label      outBin1     0x40
.label      pop1        0x57
.label      dup1        0x59
.label      swap1       0x5F
.label      iadd1       0x60
.label      isub1       0x64
.label      iand1       0x7E
.label      iinc1       0x84
.label      ifeq1       0x99
.label      iflt1       0x9B
.label      if_icmpeq1  0x9F
.label      goto1       0xA7
.label      ireturn1    0xAC
.label      ior1        0xB0
.label      invokevirtual1    0xB6
.label      wide1       0xC4
.label      in1         0xFC
.label      out1        0xFD
.label      err1        0xFE
.label      halt1       0xFF
.label      wide_iload1 0x115
.label      wide_istore1      0x136

// default instruction to place in any unused addresses of the control store
.default    goto err1
```

```
Main1 PC = PC + 1; fetch; goto (MBR)// MBR holds opcode; get next byte; dispatch

nop1  goto Main1                    // Do nothing

iadd1 MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
iadd2 H = TOS                       // H = top of stack
iadd3 MDR = TOS = MDR + H; wr; goto Main1 // Add top two words; write to TOS

isub1 MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
isub2 H = TOS                       // H = top of stack
isub3 MDR = TOS = MDR - H; wr; goto Main1 // Do subtraction; write to TOS

iand1 MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
iand2 H = TOS                       // H = top of stack
iand3 MDR = TOS = MDR AND H; wr; goto Main1     // Do AND; write to new TOS

ior1  MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
ior2  H = TOS                       // H = top of stack
ior3  MDR = TOS = MDR OR H; wr; goto Main1      // Do OR; write to new TOS

dup1  MAR = SP = SP + 1             // Increment SP and copy to MAR
dup2  MDR = TOS; wr; goto Main1     // Write new stack word

pop1  MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
pop2                                // Wait for new TOS to be read from memory
pop3  TOS = MDR; goto Main1         // Copy new word to TOS

swap1 MAR = SP - 1; rd        // Set MAR to SP - 1; read 2nd word from stack
swap2 MAR = SP                      // Set MAR to top word
swap3 H = MDR; wr                   // Save TOS in H; write 2nd word to TOS
swap4 MDR = TOS                     // Copy old TOS to MDR
swap5 MAR = SP - 1; wr        // Set MAR to SP - 1; write as 2nd word on stack
swap6 TOS = H; goto Main1           // Update TOS

bipush1     SP = MAR = SP + 1       // MBR = the byte to push onto stack
bipush2     PC = PC + 1; fetch      // Increment PC, fetch next opcode
bipush3     MDR = TOS = MBR; wr; goto Main1
                                    // Sign-extend constant and push on stack

iload1      H = LV                  // MBR contains index; copy LV to H
iload2      MAR = MBRU + H; rd      // MAR = address of local variable to push
iload3      MAR = SP = SP + 1 // SP points to new top of stack; prepare write
iload4      PC = PC + 1; fetch;wr // Inc PC; get next opcode; write top of stack
iload5      TOS = MDR; goto Main1   // Update TOS


istore1     H = LV                  // MBR contains index; Copy LV to H
istore2     MAR = MBRU + H          // MAR = address of local variable to store
into
istore3     MDR = TOS; wr           // Copy TOS to MDR; write word
istore4     SP = MAR = SP - 1; rd   // Read in next-to-top word on stack
istore5     PC = PC + 1; fetch      // Increment PC; fetch next opcode
istore6     TOS = MDR; goto Main1   // Update TOS
```

```
wide1 PC = PC + 1; fetch; goto (MBR OR 0x100)
                                    // Multiway branch with high bit set
wide_iload1 PC = PC + 1; fetch      // MBR contains 1st index byte; fetch 2nd
wide_iload2 H = MBRU << 8           // H = 1st index byte shifted left 8 bits
wide_iload3 H = MBRU OR H           // H = 16-bit index of local variable
wide_iload4 MAR = LV + H; rd; goto iload3
                                    // MAR = address of local variable to push

wide_istore1      PC = PC + 1; fetch // MBR contains 1st index byte; fetch 2nd
wide_istore2      H = MBRU << 8     // H = 1st index byte shifted left 8 bits
wide_istore3      H = MBRU OR H     // H = 16-bit index of local variable
wide_istore4      MAR = LV + H; goto istore3
                              // MAR = address of local variable to store into

ldc_w1      PC = PC + 1; fetch      // MBR contains 1st index byte; fetch 2nd
ldc_w2      H = MBRU << 8           // H = 1st index byte << 8
ldc_w3      H = MBRU OR H           // H = 16-bit index into constant pool
ldc_w4      MAR = H + CPP; rd; goto iload3 // MAR = address of constant in pool

iinc1 H = LV                        // MBR contains index; Copy LV to H
iinc2 MAR = MBRU + H; rd            // Copy LV + index to MAR; Read variable
iinc3 PC = PC + 1; fetch            // Fetch constant
iinc4 H = MDR                       // Copy variable to H
iinc5 PC = PC + 1; fetch            // Fetch next opcode
iinc6 MDR = MBR + H; wr; goto Main1 // Put sum in MDR; update variable

goto1 OPC = PC - 1                  // Save address of opcode.
goto2 PC = PC + 1; fetch            // MBR = 1st byte of offset; fetch 2nd byte
goto3 H = MBR << 8                  // Shift and save signed first byte in H
goto4 H = MBRU OR H                 // H = 16-bit branch offset
goto5 PC = OPC + H; fetch           // Add offset to OPC
goto6 goto Main1                    // Wait for fetch of next opcode

iflt1 MAR = SP = SP - 1; rd         // Read in next-to-top word on stack
iflt2 OPC = TOS                     // Save TOS in OPC temporarily
iflt3 TOS = MDR                     // Put new top of stack in TOS
iflt4 N = OPC; if (N) goto T; else goto F // Branch on N bit

ifeq1 MAR = SP = SP - 1; rd         // Read in next-to-top word of stack
ifeq2 OPC = TOS                     // Save TOS in OPC temporarily
ifeq3 TOS = MDR                     // Put new top of stack in TOS
ifeq4 Z = OPC; if (Z) goto T; else goto F // Branch on Z bit

if_icmpeq1  MAR = SP = SP - 1; rd   // Read in next-to-top word of stack
if_icmpeq2  MAR = SP = SP - 1       // Set MAR to read in new top-of-stack
if_icmpeq3  H = MDR; rd             // Copy second stack word to H
if_icmpeq4  OPC = TOS               // Save TOS in OPC temporarily
if_icmpeq5  TOS = MDR               // Put new top of stack in TOS
if_icmpeq6  Z = OPC - H; if (Z) goto T; else goto F
                              // If top 2 words are equal, goto T, else goto F
T     OPC = PC - 1; fetch; goto goto2
                              // Same as goto1; needed for target address
F     PC = PC + 1                   // Skip first offset byte
F2    PC = PC + 1; fetch            // PC now points to next opcode
F3    goto Main1                    // Wait for fetch of opcode
```

```
invokevirtual1    PC = PC + 1; fetch // MBR = index byte1; inc. PC, get 2nd byte
invokevirtual2    H = MBRU << 8     // Shift and save first byte in H
invokevirtual3    H = MBRU OR H     // H = offset of method pointer from CPP
invokevirtual4    MAR = CPP + H; rd // Get pointer to method from CPP area
invokevirtual5    OPC = PC + 1      // Save Return PC in OPC temporarily
invokevirtual6    PC = MDR; fetch   // PC points to new method; get param count
invokevirtual7    PC = PC + 1; fetch // Fetch 2nd byte of parameter count
invokevirtual8    H = MBRU << 8     // Shift and save first byte in H
invokevirtual9    H = MBRU OR H     // H = number of parameters
invokevirtual10   PC = PC + 1; fetch // Fetch first byte of # locals
invokevirtual11   TOS = SP - H      // TOS = address of OBJREF - 1
invokevirtual12   TOS = MAR = TOS + 1     // TOS = address of OBJREF (new LV)
invokevirtual13   PC = PC + 1; fetch // Fetch second byte of # locals
invokevirtual14   H = MBRU << 8     // Shift and save first byte in H
invokevirtual15   H = MBRU OR H     // H = # locals
invokevirtual16   MDR = SP + H + 1; wr    // Overwrite OBJREF with link pointer
invokevirtual17   MAR = SP = MDR;   // Set SP, MAR to location to hold old PC
invokevirtual18   MDR = OPC; wr     // Save old PC above the local variables
invokevirtual19   MAR = SP = SP + 1 // SP points to location to hold old LV
invokevirtual20   MDR = LV; wr      // Save old LV above saved PC
invokevirtual21   PC = PC + 1; fetch // Fetch first opcode of new method.
invokevirtual22   LV = TOS; goto Main1 // Set LV to point to LV Frame

ireturn1    MAR = SP = LV; rd       // Reset SP, MAR to get link pointer
ireturn2                            // Wait for read
ireturn3    LV = MAR = MDR; rd      // Set LV to link ptr; get old PC
ireturn4    MAR = LV + 1            // Set MAR to read old LV
ireturn5    PC = MDR; rd; fetch     // Restore PC; fetch next opcode
ireturn6    MAR = SP                // Set MAR to write TOS
ireturn7    LV = MDR                // Restore LV
ireturn8    MDR = TOS; wr; goto Main1 // Save return value on original TOS

halt1 goto halt1

err1  OPC = H = -1
      OPC = H + OPC
      MAR = H + OPC                 // compute IO address
      OPC = H = 1                   // 1
      OPC = H = H + OPC             // 10
      OPC = H = H + OPC             // 100
      OPC = H = H + OPC             // 1000
      OPC = H = H + OPC + 1         // 10001
      OPC = H = H + OPC             // 100010
      MDR = H + OPC + 1; wr         // 1000101 'E'
      OPC = H = 1                   // 1
      OPC = H = H + OPC             // 10
      OPC = H = H + OPC + 1         // 101
      OPC = H = H + OPC             // 1010
      OPC = H = H + OPC             // 10100
      OPC = H = H + OPC + 1         // 101001
      MDR = H + OPC; wr             // 1010010 'R'
      nop
      MDR = H + OPC; wr             // 1010010 'R'
      OPC = H = 1                   // 1
      OPC = H = H + OPC             // 10
      OPC = H = H + OPC             // 100
      OPC = H = H + OPC + 1         // 1001
      OPC = H = H + OPC + 1         // 10011
      OPC = H = H + OPC + 1         // 100111
      MDR = H + OPC + 1; wr         // 1001111 'O'
      OPC = H = 1                   // 1
      OPC = H = H + OPC             // 10
      OPC = H = H + OPC + 1         // 101
      OPC = H = H + OPC             // 1010
      OPC = H = H + OPC             // 10100
      OPC = H = H + OPC + 1         // 101001
      MDR = H + OPC; wr             // 1010010 'R'
      goto halt1

out1  OPC = H = -1
      OPC = H + OPC
      MAR = H + OPC                 // compute OUT address
      MDR = TOS; wr                 // write to output
      nop
      MAR = SP = SP - 1; rd         // decrement stack pointer
      nop
      TOS = MDR; goto Main1

in1   OPC = H = -1
      OPC = H + OPC
      MAR = H + OPC; rd             // compute IN address ; read from input
      MAR = SP = SP + 1             // increment SP; wait for read
      TOS = MDR; wr; goto Main1     // Write

//
// OUTBIN - written by Prabu  -  44 micro instructions
//     Pop the top element of stack and print the value in binary format
//       as 8 nibbles that are separated by a space
//
//   Pseudo code
//    - pop top of stack (value to be printed) to TOS and store CPP value
//    - push the following constants to stack
//          8 (number of nibbles to printed - outer loop iterations)
//          4 (number of bits in a nibble - inner loop iterations)
//         32 (ascii value for char blank)
//         48 (ascii value for char zero)
//    - use H, OPC, TOS, MDR, and CPP as temporary registers
//          OPC: holds nibble size (inner loop counter variable)
//          TOS: holds operand value to be printed out
//          CPP: holds -3 (address for output operation)
//    - outer loop (nibble iteration) - 8 times
//          * inner loop (bit iteration) - 4 times
//                print the bit as '1' or '0'
//          * print a blank
//    - at the end, pop old CPP value to CPP
//
```

```
outBin1     MAR = SP; rd            // pop the top of stack element to TOS
      nop
      TOS = MDR                     // Hold the operand to TOS
      MDR = CPP; wr                 // Save CPP - write to stack
      MDR = H = 1
      MDR = H = H + MDR
      MDR = H = H + MDR
      MDR = H + MDR
      MAR = SP = SP + 1; wr         // push nibble count (8) on stack
      OPC = MDR = H                 // OPC = 4
      MAR = SP = SP + 1; wr         // push bitcount (nibble size as 4) on stack
      MDR = H = H + MDR
      MDR = H = H + MDR
      MDR = H + MDR                 // MDR = 32
      MAR = SP = SP + 1; wr         // push ascii char space on stack
      MDR = H + MDR                 // MDR = 48
      MAR = SP = SP + 1; wr         // push ascii char zero on stack
      CPP = -1
      CPP = CPP - 1
      CPP = CPP - 1                 // CPP = -3 (address for output operation)
bitLoop     MAR = SP; rd            // read top of stack (ascii char zero)
      N = TOS; if (N) goto incOutCh; else goto printCh
incOutCh    MDR = MDR +1
printCh     MAR = CPP; wr           // print the bit value as ascii char
      H = TOS
      TOS = H + TOS                 // Left shift the operand
      OPC = OPC - 1; if (Z) goto decNibbles; else goto bitLoop
                                    // Decrement bitcount & test
decNibbles  MAR = SP - 1; rd        // read space char from the stack
      nop
      MAR = CPP; wr                 // write a space char to output device
      MDR = SP - 1
      MAR = MDR - 1; rd             // MAR = SP-2
      H = CPP
      MAR = H + SP; rd              // read previous nibble count i.e. MAR= SP-3
      OPC = MDR
      MDR = MDR - 1; wr; if (Z) goto endOutBin; else goto bitLoop1
                                    // write decremented nibble count
endOutBin   H = CPP - 1             // H = -4
      MAR = SP = H + SP; rd
      nop
      CPP = MDR                     // Restore CPP value
      MAR = SP = SP - 1; rd
      nop
      TOS = MDR; goto Main1         // Set TOS with the current TOS value
bitLoop1    goto bitLoop
```