CS99S
Laboratory 7 Preparation

November 14, 2001

Objectives:

1. Understand principles of microcoded machine design
2. Build a simple traffic light controller using microcode

Variations on a theme

Our FSM traffic light controllers have become a big hit. They are much more convenient than the mechanical controllers that have previously been used. However, this has created tremendous work for your engineering department as every customer has a slightly different set of requirements – different timings for the different stages, different rules for when to switch lights, etc…. To allow all of these requirements to be met with a single hardware design you have decided to re-implement your traffic light controller using a microcoded engine.

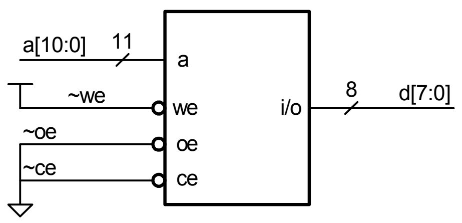
Microcode

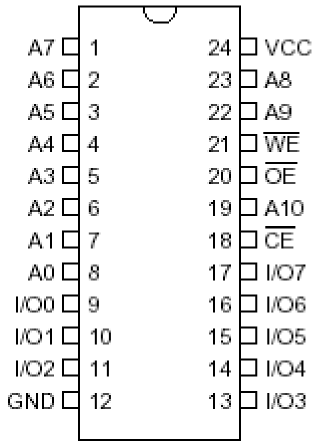
Microcode is a technique developed by Maurice Wilkes, a famous computer pioneer, in 1952 for structuring the control logic of a computer or other digital system. With microcode you replace the combinational logic that realizes the next-state and output functions of a finite-state machine with a memory, usually a read-only memory. Microcode reduces complexity and makes it easier to change the function of the logic. Complexity is reduced because many logic gates are replaced by a single memory. Flexibility is enhanced since the function of the finite-state machine can be changed without changing its wiring – just by changing the contents of the memory.

Wilkes used microcode for the control unit of the EDSAC-II in 1952. From that time until the mid-1980s it was the primary method used to implement the control units of most computer systems. Since the 1980s RISC processors and superscalar pipelines have emerged that require control that is not well suited to microcode. However, microcode controllers are still widely used in compute peripherals and other special-purpose digital systems.

The 28C16 EEPROM

The heart of our microcoded engine will be a read-only memory or ROM. This device stores a table of constants. When we apply a given *a*-bit address, A, to its address pins, it returns the *d*-bit wide constant stored at location A, D[A]. By storing the proper values into the *d*-bit wide constant array, D, we can realize any *d* combinational logic functions of *a* inputs. In particular we can realize the next-state and output functions for our FSM. After it has been programmed, the ROM is a completely combinational device. The output D depends only on the contents of the ROM (set at programming time) and the current address input, A.



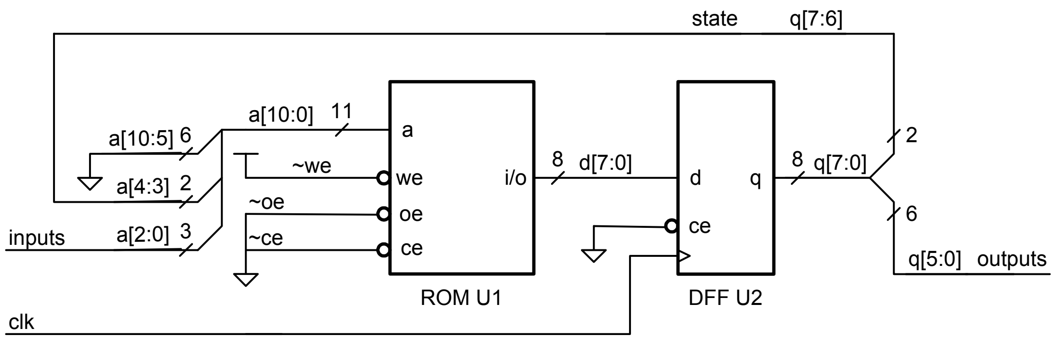


AT28C16

The 28C16 is a 16Kbit ROM organized as 2K 8-bit words, so the address is 11 bits wide and the constant data is 8 bits wide. A schematic symbol for the 28C16 and its pinout are shown above. Because we want the outputs always enabled we tie the low true chip enable (~CE) and the low true output enable (~OE) low. Because we are using the chip in read only mode we set the low true write enable (~WE) high. The write enable pin is used when programming the ROM.

To save room and to make our schematics easier to read we indicate all eleven address lines with a single line. The slash across the line and associated label "11" indicates the width of this *bus*. Similarly we show the eight data lines in the schematic as a single line with a slash indicating that this bus is eight bits wide.

A Bone-Simple Microcode Engine



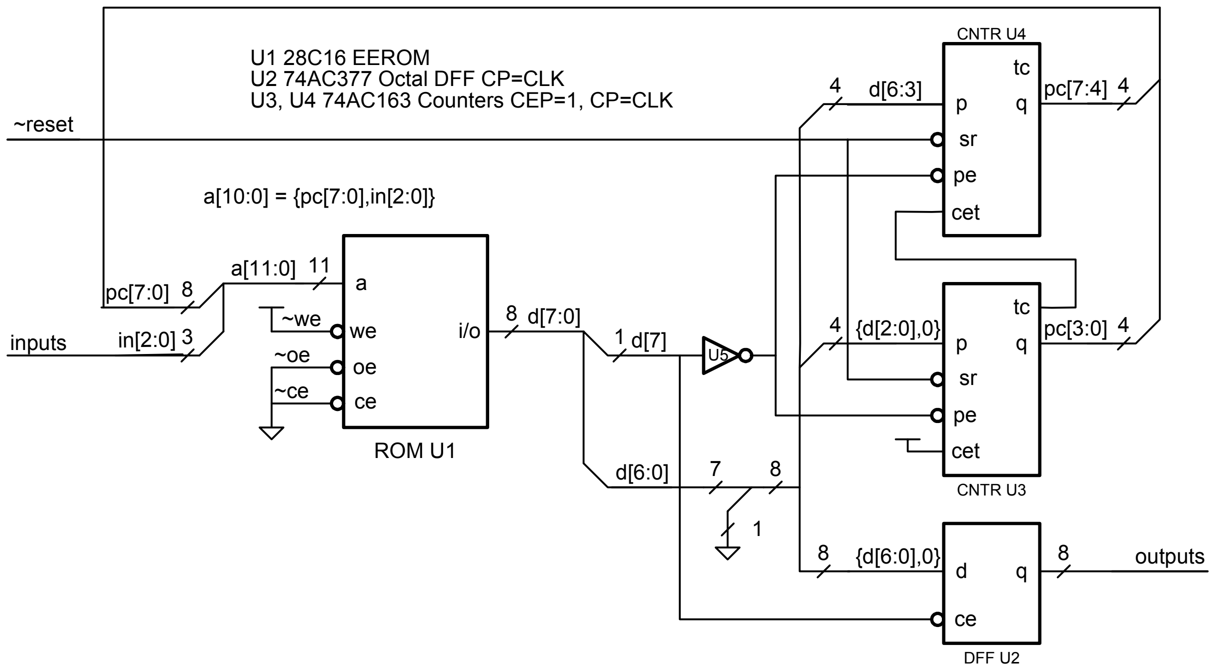
We can build a simple microcoded controller using just the ROM and an octal flip-flop (like a 74AC377) as shown above. Each cycle, the ROM calculates eight output bits d[7:0] as a function of the present state a[4:3] and the inputs a[2:0]. These eight data bits are sampled on the rising edge of the clock by the octal D-FF. The output of this flip-flop is then the current state and the outputs. This is a very simple two-chip state machine.

This schematic illustrates the notation we use to *split* a bus when we want access to the individual bits. The 11-bit address bus is split into three smaller buses, 3, 2, and 6 bits wide, and the 8-bit q bus (the output of the D FF) is split into a 6-bit output bus and a 2-bit state bus. Note that this 2-bit bus, q[7:6], is the same bus as a[4:3].

The problem with this design is that we don't have enough bits out of the ROM. We need six output bits to control our six lights. This only leaves two state bits which limits us to four states – not enough for even the simple traffic light controller from lab 6. There are three possible solutions to this problem:

1. Brute force – we can add a second ROM chip with inputs tied in parallel to this ROM and a second DFF. This gives us 16-bits to divide between output and next state. While this works, its costly – doubling the number of expensive ROM chips in our design.
2. Decode the output – instead of bringing out six outputs to directly drive the LEDs, we can encode the output into a two-bit field (00 = A green, 01 = A yellow, 10 = B green, 11 = B yellow – the state of the red light is implied) and use logic gates to decode this field. This will then leave us with six state bits, enough for 64 states. While this will work, decoding the output with logic gates will take at least two chips and loses the flexibility of generating the output with a ROM.
3. Sequence counter – we can add a microprogram counter so that we can move from state to state sequentially without needing the ROM output. We examine this solution in more depth below.

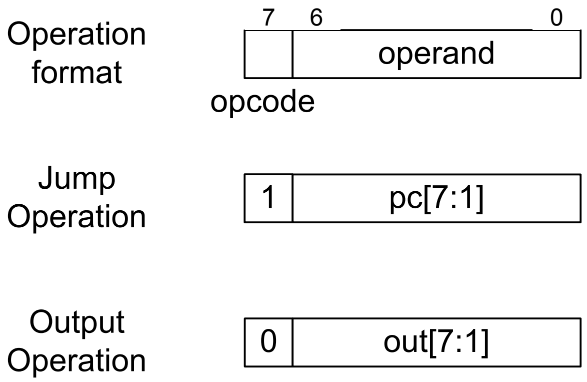
Adding a Micro Program Counter



Adding a microprogram counter as shown above enables our microcoded state machine to have a very large number of states (256 in this case) and at the same time support a large number of outputs (7 in this case) both with an 8-bit ROM. The two 74AC163 counter chips[1] (U3 and U4) form an 8-bit program counter, pc[7:0], that sequence through states. The 74AC377 now serves as an output register, holding the current value of the output lines – six of which are connected to our lights. Note that to keep this schematic from getting cluttered we have omitted the clock

[1] An appendix describes the 74AC163 and gives its pinout.

inputs from U2-U4 and the CEP input from U3 and U4. These inputs are connected as indicated in the note.



Each ROM word contains a micro-operation that consists of a one-bit opcode and a seven bit operand as illustrated above. The one-bit opcode, d[7], encodes two micro-operation types: jump and output. If d[7] is low, the ROM word contains an output operation. During this operation d[6:0] are placed in the high seven bits of the output register (the low bit will always be zero), and hence onto the output lines, and the microprogram counter is incremented. If d[7] is high, the ROM word contains a jump micro-operation. This operation causes d[6:0] to be loaded into the high seven bits of the microprogram counter – effectively jumping to a new instruction and the output register holds its previous state. The microprogram counter is reset to zero upon reset, so the controller always starts in a known state. We can express the action of reset and the two operation types on the program counter (pc) and the output register (out) in a C-like register transfer notation as:

```
Reset:        pc = 0 ;

Out(d):       out = d[6:0]<<1 ;
              pc = pc+1 ;

Jump(d):      pc = d<<1 ;
```

The current microinstruction is selected by the concatenation of the microprogram counter, pc[7:0], and the current input, in[2:0] (carA, carB, and function). Thus there are eight operations for each program counter value, one for each possible combination of the three input values. For the rest of this lab we will refer to one of these eight possible operand/opcode pairs as an *operation* and to the set of eight operations selected by a particular program counter value as an *instruction.*

While selecting operations using the input bits like this lets us jump eight ways out of a single instruction, it is rather wasteful of ROM space. We could eliminate this waste by employing an input multiplexer – to look at a single input at a time, and possibly making our jump conditional upon this input. However, for this lab we can afford to waste the ROM space, so we won't add the complexity of an input mux.

Prep Question 1: Prepare a *wiring table* for the microcontroller design above. This table should list each **input** of each of the five chips (ROM, 377, two 163s, and one 04) and for that input list (1) the name of the input, (2) the name of the signal connected to that input, (3) the unit number and pin number for the input, and (4) the unit number and pin number of the source that drives the input. For example, ROM address pin a[4] should have the following entry:

a[4]	pc[1]	U1-4	U3-13

You should write a corresponding line for every input pin on every chip. Check these tables carefully as you will be wiring from them. A small mistake here will cost you lots of debugging time later.

Prep Question 2 (Optional Challenge): Draw a schematic (to the level of detail of the schematic above) of a microcode engine that allows a full 2048 states by multiplexing the three inputs together (might as well make it four) and making jumps conditional upon a selected input. You should also allow an unconditional jump. What is your "micro instruction set"? Make sure that you still have at least six outputs. (Hint: Use a 74AC153 for the multiplexer and use a third 74AC163 to extend the program counter to 11 bits).

Some Example Microcode

To make our microcoded engine perform a specified function, we need to write microcode for it. While we eventually need to translate the microcode into binary to load it into the ROM, it is easier to think about the microcode if we write it in a mnemonic assembly language. For example, the following microprogram implements a simple traffic light controller:

```
//----------------------------------------------------------------------
// outputs are ag,ay,ar,bg,by,br
#define GREENA   0x21  // 10 0001
#define YELLOWA  0x11  // 01 0001
#define REDBOTH  0x09  // 00 1001
#define GREENB   0x0c  // 00 1100
#define YELLOWB  0x0a  // 00 1010
//----------------------------------------------------------------------
int reset, await, byellow, bwait, ayellow ;
//----------------------------------------------------------------------
void microcode() {
  title("* Microcode for Simple Traffic Light Controller\n") ;
  S(reset)
  S(byellow)
    always(OUT,YELLOWB);  // b yellow for 2 cycles
    always(OUT,YELLOWB);
    always(OUT,REDBOTH) ; // both red for 1 cycle
    always(OUT,GREENA) ;  // stay green for 5 cycles
    always(OUT,GREENA) ;
    always(OUT,GREENA) ;
    always(JUMP,await) ;
  S(await)
    iff(CARB,JUMP,ayellow) ; iff(NOTB,JUMP,await) ; next() ;
  S(ayellow)
    always(OUT,YELLOWA) ;  // stay yellow 2 cycles
    always(OUT,YELLOWA) ;
    always(OUT,REDBOTH) ;  // both red for 1 cycle
    always(OUT,GREENB) ;   // stay green for 3 cycles
    always(JUMP,bwait) ;
  S(bwait)
    iff(ELSE,JUMP,byellow) ; iff(NOT_A_AND_B,JUMP,bwait) ; next() ;
}
//----------------------------------------------------------------------
```

The first section of `#define` statements defines the five output values that are used in the main program. Next, the names of the four states (`await, byellow, bwait, and ayellow`) are declared as integers. The body of the program comes next. `S(byellow)` declares the `byellow` state – so we can jump to it later from the `bwait` state. This state will run for seven cycles. The first six cycles execute output instructions making the light on B go yellow for two cycles, then making both lights red for one cycle, and finally making the light green on A for three cycles. Finally the state ends in a JUMP command to state `await`. The `always` construct around each of these seven instructions indicates that they are to be executed regardless of the state of the input lines. Thus, each of these lines generates eight identical instructions to fill the eight slots selected by the eight input conditions.

State `await` comes next and consists of two jump instructions that are executed conditionally. In this state we jump to state `ayellow` if `CARB` is true and back to await if `NOTB`. Each of these `iff` constructs generates four jump instructions – for the four input combinations that correspond to the requested state of `CARB`. The `next()` ; construct increments the program counter to the next instruction – indicating the end of `iff` constructs for this instruction.

The `ayellow` and `bwait` states perform similar functions but with different conditions.

The assembled microcode in *hex file* format is shown below. Each line represents one instruction and starts with the ROM byte address for that line followed by the eight one-byte operations – all in hexadecimal. For example, the first line corresponds to the first instruction of the `byellow` sequence and corresponds to `always(OUT,YELLOWB)` – putting hex `0A` into each operation – an output operation to set the lights to the `YELLOWB` pattern. The next five lines are similar – containing `OUT` operations in all eight positions. I've added comments to each line showing the correspondence to the assembly program. However the hex file that gets loaded to program your ROMs should not contain comments.

The instruction at address 30 hex is the jump to `await`. The operation is 84 hex giving an opcode of 1 – `JUMP`, and an operand of 4 which specifies address 40 since a[2:0] come from the inputs and a[3] is always zero on a jump. The instruction at address 38 is not used – and hence left all zeros since `await` must start on an address with the low four bits zero. The instruction at `await`, hex 40, is the conditional branch. Operations in slots corresponding to CarB (which is in[0]) being one are 85 – which is a jump to the instruction at `ayellow`, hex 50, while operations for which CarB is zero are 84 – a jump back to `await` at hex 40. The rest of the code is similar.

```
0000 0A 0A 0A 0A 0A 0A 0A 0A    byellow: always(OUT,YELLOWB)
0008 0A 0A 0A 0A 0A 0A 0A 0A             always(OUT,YELLOWB)
0010 09 09 09 09 09 09 09 09             always(OUT,REDBOTH)
0018 21 21 21 21 21 21 21 21             always(OUT,GREENA)
0020 21 21 21 21 21 21 21 21             always(OUT,GREENA)
0028 21 21 21 21 21 21 21 21             always(OUT,GREENA)
0030 84 84 84 84 84 84 84 84             always(JUMP,await)
0038 00 00 00 00 00 00 00 00
0040 84 85 84 85 84 85 84 85    await:   iff(CARB,JUMP,ayellow) iff(NOTB,JUMP,await)
0048 00 00 00 00 00 00 00 00
0050 11 11 11 11 11 11 11 11    ayellow: always(OUT,YELLOWA)
0058 11 11 11 11 11 11 11 11             always(OUT,YELLOWA)
0060 09 09 09 09 09 09 09 09             always(OUT,REDBOTH)
0068 0C 0C 0C 0C 0C 0C 0C 0C             always(OUT,GREENB)
0070 88 88 88 88 88 88 88 88             always(JUMP,bwait)
0078 00 00 00 00 00 00 00 00
0080 80 88 80 80 80 88 80 80    bwait:   iff(ELSE,JUMP,byellow) iff(NOT_A_AND_B,JUMP,bwait)
```

The eight columns correspond to the eight input conditions – states of function, carA, carB as a binary number. For example, function=0, carA=1, carB=0 (010) corresponds to the third column. The first four columns are for function = 0 and the second four are for function = 1. Within each of these groups of four columns the first column is `~A&~B`, then `~A&B`, then `A&~B`, and finally `A&B`.

Prep Question 3 (Optional Challenge): You will notice that we waste a lot of ROM space repeating the same output command for a number of cycles. In effect we're using the program counter as a timer to count off the number of cycles the lights should be in a particular state. For our lab this is fine – we have lots of ROM space. However in some applications where microcode is tight, this isn't acceptable. Explain how to add a timer to our simple microengine so that it supports a third instruction – WAIT (in addition to JUMP and OUT). The opcode of the WAIT instruction is a count of the number of cycles to wait before proceeding to the next instruction. Draw a schematic, to the same level of detail as the drawings above, showing how you accomplish this.

The Lab

For the lab you will use the microcode sequencer we developed above to implement a universal traffic-light controller that implements two different light algorithms as selected by the function input bit. The codes and algorithms are:

0 – Exactly the same behavior as your machine from Lab 6 – you can add extra cycles where needed to fit into the restrictions of the sequencer.
1 – Like machine 00 except (a) the yellow lights must all last three cycles, (b) the green lights must each last at least five cycles, and (c) a yellow light must be followed by two cycles where both lights are red.

Prep Question 4: Generate a microcode hex file for your state machine design. You can either do this by hand or modify the C program I used to generate the microcode above. The program is attached as an appendix and available on the course web page. For these small programs, its probably faster to just assemble your program by hand. You should bring to the lab – your program in assembly language, a printout of your hex file, and a floppy disk (PC format) containing your hex file.

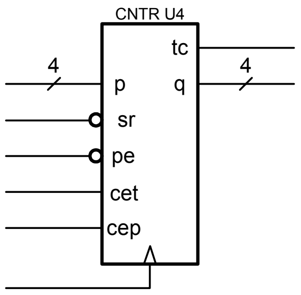
(Challenge) A prize will be awarded to the student who implements this code in the smallest number of microinstructions. To win the code must **exactly** implement the specified functionality. Ties will be broken by picking the code that has the fewest cycles around the loop when both A and B are true. Ties on both of these counts will be broken by choosing the student who gets their lab working first.

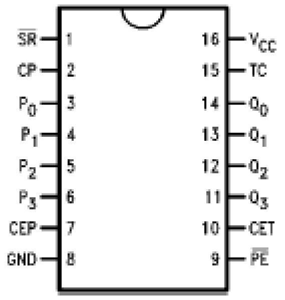
During the lab you will

1. Have the professor or TA check your wiring table from question 1 and your microcode from question 4.
2. Program your ROM with your microcode – the professor or TA will show you how to use the ROM burner to do this.

3. Wire up the circuit using your wiring table from question 1. Make sure not to route any wires over the ROM since may need to remove it for reprogramming.
4. Demonstrate operation of your microcoded engine.

Appendix 1 – The 74AC163 Binary Counter





The 74AC163 is a four-bit synchronous binary counter with parallel load and reset capability. The chip has four bits of state that are output on lines q[3:0]. This state can be reset, loaded from the p input, or incremented depending on the control inputs. Asserting reset (setting the ~sr pin low) clears q to zero on the next rising edge of the clock. Asserting parallel enable (setting the ~pe pin low when reset is not asserted) loads the four state bits from the p[3:0] input on the next rising edge of the clock. Finally, setting both count enables (cep and cet) high increments q by one on the next clock. When q reaches a count of 15, the tc output is asserted high. By connecting the tc output of one 74AC163 counter to the cep input of another 74AC163 counter (as we do in this lab), wider counters can be realized. The function of the control inputs are summarized in the following table.

~sr	~pe	cet	cep	q[3:0] on next clk	description
0	*	*	*	0000	reset
1	0	*	*	p[3:0]	load
1	1	1	1	q[3:0]+1 (mod 16)	increment
1	1	0	*	q[3:0]	hold
1	1	*	0	q[3:0]	hold

Appendix 2– C Program To Assemble Microprograms

The following C program expects a two subroutines called microcode0 and microcode1 with calls to S (shorthand for state), always, iff, and next and outputs a hex file containing the microcode for these two microprograms, one for each state of the function input. The names of all of the labels (states) in the microprograms must be declared as static integers.

```
//------------------------------------------------------------------------
// Simple microcode assembler
// W. J. Dally - 11/19/01
//------------------------------------------------------------------------
#include <Stdio.h>
#include <stdlib.h>
//------------------------------------------------------------------------
// defines for microcode assembler
//------------------------------------------------------------------------
#define MAXWORDS (1<<11)        // maximum size of microcode
//------------------------------------------------------------------------
```

```
#define PCSHIFT 3
#define CONDCOUNT 8
//------
#define S(x) state(&x)
//------
#define OUT 0
#define JUMP 1
//------
// conditions - four bits that define the truth table
//  3   2     1    0
// a&b a&~b ~a&b ~a&~b
#define CARB            0xa // 1010
#define CARA            0xc // 1100
#define NOTB            0x5 // 0101
#define NOTA            0x3 // 0011
#define A_AND_NOT_B     0x4 // 0100
#define NOT_A_AND_B     0x2 // 0010
#define A_AND_B         0x8 // 1000
#define NOT_A_AND_NOT_B 0x1 // 0001
#define A_OR_NOT_B      0xd // 1101
#define NOT_A_OR_B      0xb // 1011
#define A_OR_B          0xe // 1110
#define NOT_A_OR_NOT_B  0x7 // 0111
#define A_XOR_B         0x6 // 0110
#define A_EQUAL_B       0x9 // 1001
#define ALWAYS          0xf // 1111
//-------------------------------------------------------------------------
unsigned code[MAXWORDS]   ;       // microcode array
int pc = 0 ;                     // current program counter
int max_pc = 0 ;
char *t ;
int function ;
//-------------------------------------------------------------------------
void clear() {
  int i ;
  for(i = 0 ; i<MAXWORDS ; i++) code[i] = 0 ;
  max_pc = 0 ;
}
//-------------------------------------------------------------------------
void init() {
  pc = 0 ;
}
//-------------------------------------------------------------------------
void title(char *s) {
  t = s ;
}
//-------------------------------------------------------------------------
// align to even address and assign address to state
void state(int *state) {
  if(pc & 1) pc++ ; // align to even
  *state = pc ;
}
//-------------------------------------------------------------------------
// bump the pc
void next() {
  pc++ ;
  if(pc > max_pc) max_pc = pc ;
}
//-------------------------------------------------------------------------
// insert the operation into the cases that match the condition
void iff(int cond, int opcode, int operand) {
  int i ;
  int addr = pc<<PCSHIFT ;

  // adjust the operand - and test validity for jumps
  // can only jump to even addresses and LSB is dropped
  if(opcode == JUMP) {
    if(operand & 1)
      fprintf(stderr,"illegal jump target %02X at %04X \n",operand,pc) ;
    operand = operand >> 1 ;
  }
  // now insert the operation into the matching locations
  for(i = 0 ;i<CONDCOUNT;i++) {
    if(((i>>2) == function) && ((cond>>(i&3))&1))
      code[addr+i] = (opcode<<7)+operand ;
  }
}
//----------------------------------------------------------------------
// insert instruction in all eight cases - and increment pc.
void always(int opcode, int operand) {
  iff(ALWAYS, opcode, operand) ;
  next() ;
}
//----------------------------------------------------------------------
void print_hex() {
  int i, j ;
  //printf("%s",t) ;
  for(i=0;i<max_pc;i++) {
    printf("%04X ",i<<PCSHIFT) ;
    for(j=0;j<CONDCOUNT;j++) {
      printf("%02X ",code[(i<<PCSHIFT)+j]) ;
    }
    printf("\n") ;
  }
}
//----------------------------------------------------------------------
// defines for this program
//----------------------------------------------------------------------
// outputs are ag,ay,ar,bg,by,br
#define GREENA   0x21  // 10 0001
#define YELLOWA  0x11  // 01 0001
#define REDBOTH  0x09  // 00 1001
#define GREENB   0x0c  // 00 1100
#define YELLOWB  0x0a  // 00 1010
//----------------------------------------------------------------------
// microcode for function = 0
// traffic light controller with 2 cycle yellow, 1 cycle both red
// 5 cycle green on A, 3 cycle green on B, goes to ayellow on carb
// goes to byellow on car A or not car B
//----------------------------------------------------------------------
// state definitions
int await, byellow, bwait, ayellow ;
void microcode0() {
  S(byellow);
    always(OUT,YELLOWB);  // b yellow for 2 cycles
    always(OUT,YELLOWB);
    always(OUT,REDBOTH) ; // both red for 1 cycle
    always(OUT,GREENA) ;  // stay green for 5 cycles
    always(OUT,GREENA) ;
    always(OUT,GREENA) ;
    always(JUMP,await) ;
  S(await);
    iff(CARB,JUMP,ayellow) ; iff(NOTB,JUMP,await) ; next() ;
  S(ayellow) ;
    always(OUT,YELLOWA) ;  // stay yellow 2 cycles
    always(OUT,YELLOWA) ;
    always(OUT,REDBOTH) ;  // both red for 1 cycle
    always(OUT,GREENB) ;   // stay green for 3 cycles
    always(JUMP,bwait) ;
  S(bwait);
    iff(A_OR_NOT_B,JUMP,byellow) ; iff(NOT_A_AND_B,JUMP,bwait) ; next() ;
}
//----------------------------------------------------------------------
// microcode for function = 1
// state definitions only needed if different than above
// this doesn't do anything particularly useful
//----------------------------------------------------------------------
int start, doa, dob ;
void microcode1() {
  S(start);
    iff(CARA,OUT,GREENA) ; iff(NOTA,OUT,GREENB) ; next() ;
    iff(CARB,JUMP,dob) ; iff(NOTB,JUMP,doa) ; next() ;
  S(doa);
    always(OUT,YELLOWA) ;
    always(OUT,REDBOTH) ;
    always(OUT,GREENB) ;
    iff(A_AND_NOT_B,JUMP,start) ;
      iff(A_EQUAL_B,JUMP,doa) ;
      iff(NOT_A_AND_B,JUMP,dob) ;
      next() ;
  S(dob);
    always(OUT,YELLOWB) ;
    always(OUT,GREENA) ;
    always(JUMP,start) ;
}
//----------------------------------------------------------------------
void main(){
  clear() ;
  // first pass to build symbol table second pass to generate code
  // for function = 0 first
  function = 0 ; init() ; microcode0() ; init() ; microcode0() ;
  function = 1 ; init() ; microcode1() ; init() ; microcode1() ;
  print_hex() ;
}
//----------------------------------------------------------------------
```

This is the result of running the program

```
0000 0A 0A 0A 0A 0C 0C 21 21
0008 0A 0A 0A 0A 81 83 81 83
0010 09 09 09 09 11 11 11 11
0018 21 21 21 21 09 09 09 09
0020 21 21 21 21 0C 0C 0C 0C
0028 21 21 21 21 81 83 80 81
0030 84 84 84 84 0A 0A 0A 0A
0038 00 00 00 00 21 21 21 21
0040 84 85 84 85 80 80 80 80
0048 00 00 00 00 00 00 00 00
0050 11 11 11 11 00 00 00 00
0058 11 11 11 11 00 00 00 00
0060 09 09 09 09 00 00 00 00
0068 0C 0C 0C 0C 00 00 00 00
0070 88 88 88 88 00 00 00 00
0078 00 00 00 00 00 00 00 00
0080 80 88 80 80 00 00 00 00
```