Lecture 4 Computer Organization and Design   DKP/Fall 2019

____

Digital Logic

____

This is a conundrum. An introductory course on computer organization and design should focus on concepts rather than on engineering detail (I assume students are not focused on hardware design) and should explain the subject from a programmer's point of view and emphasize consequences for programmers. Normally, I would omit logic design altogether. But perhaps I can give you a taste of it without getting bogged down in details.

Logic operators are abstractions for specifying transformations of binary signals. The simplest logic circuits are _combinational_. This means that they do not operate with _memory_, or _state_. Combinational circuits may be abstracted as Boolean functions, which are familiar to us from our study of _sentential logic_ (also called _propositional calculus_).

The mathematical concept underlying combinational logic is that of _n-place Boolean function_. A Boolean function specifies what operation we want a combinational circuit to compute. In all cases, there are many implementations of a given Boolean function. To speak of this, we need terminology to specify Boolean functions, and means to describe particular implementations. Diagrams will describe implementations, and formulas from sentential logic will specify Boolean functions.

Physically, combinational circuits are constructed by wiring together some number of logic gates, which have names like 'NOT', 'AND', 'OR', 'XOR', 'NAND', 'NOR', and 'XNOR'. The most familiar gates correspond to familiar binary sentential connectives. Other gates correspond to slightly less familiar binary sentential connectives (e.g., '+', "downward arrow", and '|'). Of course, the most familiar sentential connectives are binary, while logic gates may have more than two inputs (and, to a lesser extent, more than one output). Multiple-input 'NAND', 'NOR', and 'XNOR' gates are _defined_ as the negations of multiple-input 'AND', 'OR', and 'XOR' gates, respectively. I will tell you the Boolean function that multiple-input 'XOR' gates compute in a moment.

An n-place _Boolean function_ is a map from {T, F}^n to {T, F}. For each 'n', there are 2^(2^n) n-place Boolean functions. It is convenient to identify a connective with the Boolean function it computes. This gives us 2^(2^n) n-ary connectives. We will look at all connectives with n <= 2.

There are two 0-place Boolean functions, 'T' and 'F'. The corresponding connectives---which are not representable as ASCII characters---are 'T' and 'inv. T'. There are four unary connectives, but only negation is of any interest. There are sixteen binary connectives, but only the last ten listed below are "really binary". I provide a correspondence table, which can be used for reference. I list all sixteen two-place Boolean functions and the connectives that correspond to them.

Digression: Are these sixteen binary connectives are sufficient to compute _all_ Boolean functions? Would it help to add, say, the ternary majority connective '#', which is true whenever a majority of its three arguments is true? No. We can compute everything we need from {'~', '/\'} or {'~', '\/'}, to give just two of several examples. Still, logicians enjoy naming some interesting (if not strictly necessary) ternary---and higher---connectives.

logic	name	gate	meaning	multiple input
T	true	nullary;	always true	
inv. T	false	nullary;	always false	
p	- -	- -	'p'	
q	- -	- -	'q'	
~p	not	NOT	not 'p'	
~q	not	NOT	not 'q'	
p /\ q	and	AND	'p' and 'q'	"all"
p \/ q	or	OR	'p' or 'q'	"any"
p --> q	conditional	- -	if 'p', then 'q'	
p <--> q	biconditional	XNOR	'p' iff 'q'	"not odd"
p <-- q	reversed conditional	- -	if 'q', then 'p'	
p + q	exclusive or	XOR	'p' or 'q', but not both	"odd"
p $\overset{|}{v}$ q	nor	NOR	neither 'p' nor 'q'	"not any"
p | q	nand	NAND	not both 'p' and 'q'	"not all"
p < q	- -	- -	(not 'p') and 'q'	
p > q	- -	- -	'p' and (not 'q')	

Remark: None of {'-->', '<--', '<', '>'} is associative or commutative.

We can pair Boolean functions/connectives with their negations:

<T, inv. T>; <p, ~p>; <q, ~q>; <p /\ q, p | q>; <p \/ q, p $\overset{|}{v}$ q>;

<p + q, p <--> q>; <p --> q, p > q>; <p <-- q, p < q>

In other terminology: <NOT, ident>; <AND, NAND>; <OR, NOR>; <XOR, XNOR>

Example: An electrical device with three inputs (clearly combinational).

```
         +--------+
x1 ---> |        |
x2 ---> |        | ---> f(x1,x2,x3)
x3 ---> |        |
         +--------+
```

To say that the device has no memory is to say that the present output level depends only on the present input levels (and not on the past history of the device).

Consider the two-input 'AND' gate:

```
        +--------+
p ---> |        |
        |  AND   | ---> p /\ q
q ---> |        |
        +--------+
```

We can label the output with a logical formula.

Consider the two-input 'OR' gate:

```
        +--------+
p ---> |        |
        |   OR   | ---> p \/ q
q ---> |        |
        +--------+
```

Now, consider the less familiar two-input 'NAND' gate:

```
        +--------+
p ---> |        |
        |  NAND  | ---> p | q
q ---> |        |
        +--------+
```

Finally, consider the unfamiliar two-input 'XNOR' gate:

```
        +--------+
p ---> |        |
        |  XNOR  | ---> p <--> q
q ---> |        |
        +--------+
```

You do not have to know all the logic-gate names or all the logic connectives; they will be provided to you as needed.

Here is the biggest circuit I am willing to draw today:

```
       +---------+
p ---> |         |
       |   AND   | ---> p /\ q ---+
q ---> |         |                |
       +---------+                |     +---------+
                                  +---> |         |
                                        |   OR    | ---> (p /\ q) \/ ~r
                                  +---> |         |
       +---------+                |     +---------+
       |         |                |
r ---> |   NOT   | ---> ~r -------+
       |         |
       +---------+
```

It should be intuitive that, just as every logical formula is tautologically equivalent to many other quite distinct logical formulas, so the same Boolean function can be implemented by many different circuits (i.e., different combinations of gates). For example, each fully parenthesized Boolean expression (wff) corresponds to a _unique_ tree of sentence symbols and connectives, and thus suggests a particular implementation of the Boolean function.

Whereas the process of converting a logic expression to a logic diagram, and then to an associated hardware realization, is trivial, obtaining a logic expression that leads to the best possible hardware circuit is not. For one thing, the definition of "best" changes depending on the technology and the implementation scheme being used (e.g., custom VLSI, programmable logic, or discrete gates), and on the design goals (e.g., area, time, power, and cost).

For another, the simplification process, if not done via automatic design tools, is not only cumbersome but also imperfect; for example, it might be based on minimizing the number of gates employed, without taking into account the time and energy implications of the wires that connect the gates.

I will not teach you any hand-minimization techniques for logic circuits, which I regard as obsolete.

Today, all digital design of processors and related hardware systems is done using a _hardware description language_, along with software tools for digital analysis and synthesis. The current specification language is VHDL.

The next step up from combinational logic is _sequential logic_, which we need to make registers, caches, and memories. You may think of a sequential circuit as a combinational circuit plus timed state elements.

The behavior of a combinational (memoryless) circuit depends only on its current inputs, not on past history. A sequential circuit, on the other hand, has a finite amount of memory whose content, determined by past inputs, affects the current output behavior.

Combinational circuits implement _Boolean functions_. Sequential circuits implement _finite-state machines_. In the simplest case, the machine stores the current state. When input is received, the current state is turned into the next state, and then output. State plays a big role in pipelines.

A state element has at least two inputs and one output. The required inputs are the data value to be written into the element, and the clock, which determines when the data value is written. The output from a state element provides the value that was written in an earlier clock cycle. The clock is used to determine when the state element should be written; a state element can be read at any time after it has been written. Note: there exist state elements without clocks, but we ignore them.

Because only state elements can store a data value, any collection of combinational logic must have its inputs come from a set of state elements, and its outputs written into a set of state elements. Below, 'd' = data.

Example:

```
                           ______________
     +-----------+        /              \        +-----------+
 d   |           |  d    /                \   d   |           |  d
---> | state     | ---> /  combinational   \ ---> | state     | --->
     | element 1 |      \  logic           /      | element 2 |
     |           |       \                /       |           |
     +-----------+        \______________/        +-----------+
           ^                                            ^
           |                                            |
         clock                                        clock
```

When a clock signal is received, the value of the data input to a state element is instantaneously stored in the state element. This value is stable until the next clock signal is received.

The state elements provide valid inputs to the combinational logic block. But the combinational logic itself is not instantaneous; rather, it needs time to settle. To ensure that the values written into the state element on the right are valid, the clock must have a long enough period so that all the signals in the combinational logic block stabilize, _after_ which the stable values can be stored into the receiving state element.

Finally, I will give a very brief sketch of how combinational logic may be used to implement a very simple (and slow) adder.

When two bits are added, the sum is a value in the range [0,2] that can be represented by a _sum bit_ and a _carry bit_. A circuit to compute both is known as a _half adder_. Let me use the symbol '+' to represent 'XOR' (exclusive or). That is, 'p + q' is equivalent to '(p \/ q) /\ ~(p /\ q)'.

A moment's reflection shows that a half adder can be implemented with an 'AND' gate and an 'XOR' gate, since sum = 'p + q' and carry = 'p /\ q'.

Here is a (possibly unnecessary) illustration of a half adder:

```
            +--------+
p --o-----> |        |
    |       |  AND   | ---> p /\ q = carry
q ----o---> |        |
    | |     +--------+
    | |
    | |     +--------+
    +-----> |        |
      |     |  XOR   | ---> p + q = sum
      +---> |        |
            +--------+
```

Two inputs are fed into each of two gates, giving us two outputs.

By adding a carry input to a half adder, we get a _full adder_.

```
         c_in
           |
           v
       +--------+
p ---> |        |
       |   FA   | ---> sum
q ---> |        |
       +--------+
           |
           v
         c_out
```

Here is a straightforward implementation of a full adder:

```
       +--------+
p ---> |        |
       |   HA   | ---> p /\ q ------+
q ---> |        |                   |
       +--------+                   |
           |                        |          +--------+
  p + q    |          +--------+    +--------> |        |
           +--------> |        |               |   OR   | ------------->
                      |   HA   | ------------> |        | (p /\ q) \/
c_in = r -----------> |        | (p + q) /\ r  +--------+ ((p + q) /\ r)
                      +--------+                          = c_out
                           |
                           v
                    (p + q) + r = sum
```

Abstractly, a full adder takes three inputs ('p', 'q', and 'r') and computes two Boolean functions as indicated. There are implementations other than the one shown, and other ways of naming the two Boolean functions.

Some of the other names are:

sum   = p + q + r |= =| p <--> q <--> r |= =| (+)^3 pqr.

c_out = pq \/ r(p + q) |= =| pq \/ pr \/ qr |= =| #pqr.

'(+)^3' is ternary addition modulo 2. '#' is the ternary majority connective.

Since the output labels---which both specify the Boolean functions the full adder computes and trace the circuit operation---are fully parenthesized, they indicate how an implementing circuit _was_ synthesized from simpler Boolean expressions. Thus, we could work backwards, starting with these expressions as the specification, and translating them directly into an implementing circuit.

Noting the reuse of 'p + q', we get a slightly different picture:

```
     ^               c           s = p + q + r
     |               ^
     |               |
     |        s      OR          c = (p /\ q) \/
     |        ^     / \              (r /\ (p + q))
time |        |    /   \
     |        +   AND   AND
     |       / \ / \   / \       s = (+)^3 pqr
     |      r   +   r p   q
     |         / \
     |        p   q              c = #pqr
```

A full adder, connected to a state element for holding the carry bit from one cycle to the next, functions as a _bit-serial_ adder. A _ripple-carry adder_, on the other hand, unfolds the sequential behavior of a bit-serial adder into space, using a cascade of 'k' full adders to add two k-bit numbers. There are a number of faster adders.

Once we have an adder, we can implement, say, a _counter_, and so on.

Digital-Logic Review

____

A combinational circuit with 'k' inputs and one output is a hardware realization of a k-place Boolean function.

```
         +---------+
  p ---> |         |
         |   AND   | ---> p /\ q
  q ---> |         |
         +---------+
```

For example, an 'AND' gate is a hardware realization of a simple two-place Boolean function. It is trivial to write out the truth table. The diagram is self-explanatory.

The output has been labeled with a logical formula. It completely defines the Boolean function realized by this circuit, which here is a single gate.

This logical formula is not unique: '(p /\ q) /\ (p \/ ~p)' represents the same Boolean function.

For each function, there is at least one formula that expresses it. Some are more concise. We are not responsible for transforming logical formulae into their "optimal" forms, whatever you might mean by this. Also, when we translate formulae into circuits, we are usually given a _vocabulary_ of what connectives, i.e., what gates, we are allowed to use. The simplest vocabulary is:

'/\', '\/', '~' ('AND', 'OR', 'NOT').

Suppose someone gives you information about a Boolean function's truth table.

For example, suppose someone says: Three-place Boolean function 'f' is true for '001' and '010', and false everywhere else.

You can write down immediately: ~p.~q.r + ~p.q.~r With pen and paper, you can more elegantly write:

$$\bar{p}\,\bar{q}\,r + \bar{p}\,q\,\bar{r}$$

i.e., explicitly list the triples that make this function true. This logical formula happens to be in _sum-of-products_ form.

Definition:: sum-of-products form: a Boolean sum of terms, each term being a Boolean product of variables and their complements.

This involves a change of notation. Products are simple concatenation, while sum is indicated by '+'. And complementation is indicated by writing a _bar_ over the variable.

To have one sum-of-products form doesn't mean it's the shortest one.

$$p\,q\,r + p\,\bar{q}\,r + p\,\bar{q}\,\bar{r} + \bar{p}\,\bar{q}\,r \;\; |= =| \;\; p\,r + p\,\bar{q} + \bar{q}\,r$$

We are not responsible for transforming the first formula into the second, a process often called _optimization_.

We are responsible for translating (reasonably nice) formulae directly into circuits.

Consider : $r + p\,q + \bar{p}\,\bar{q}$

This formula is _not_ fully parenthesized. We try:

```
       +---------+
p ---> |         |
       |  AND    | ---> p /\ q ---+
q ---> |         |                |
       +---------+                |                 +---------+
                                  +---------------> |         |
                                                    |   OR    | ---> p q +
                                                    |         |      _ _
       +---------+                                  +---------+     (p q + r)
p ---> |  NOT    | ---+                                  ^
       +---------+    |  +---------+  +---------+        |
                      +--|         |  |         |        |
                         |  AND    |--|   OR    |--------+
                      +--|         |  |         |
       +---------+    |  +---------+  +---------+
q ---> |  NOT    | ---+                    ^
       +---------+                         |
                                           |
r -----------------------------------------+
```

Here, the longest path is: ** 'NOT'; 'AND'; 'OR'; 'OR' **.

But equivalently:

```
       +---------+      +---------+
p ---> |         |      |         |
       |  AND    | ---> |   OR    | ---+
q ---> |         |      |         |    |
       +---------+      +---------+    |          +---------+
                             ^         |          |         |
                             |         +------->  |   OR    | ---> (p q + r) +
r ---------------------------+                    |         |
                                                                    _ _
       +---------+                                +---------+       p q
p ---> |  NOT    | ---+                                ^
       +---------+    |   +---------+                  |
                      +---|         |                  |
                          |  AND    |------------------+
                      +---|         |
       +---------+    |   +---------+
q ---> |  NOT    | ---+
       +---------+
```

Here, the longest path is: ** 'AND'; 'OR'; 'OR' **, which is clearly better (even though the 'NOT' gate is quite short in comparison to the others).

Example:

Consider p + q + r + s.

There are four inputs. How shall we lay out the 'OR' gates so that the circuit settles as rapidly as possible?

Since 'OR' gates take time to settle, we should construct the shallowest tree possible. Always, this is the most balanced tree possible. This will minimize the longest path from a leaf (variable or subcircuit) to the root (circuit output). On the left, the worst tree. On the right, the best tree.

```
   +                +         ^
  / \              / \        |
 p   +            /   \       |
    / \          +     +      | time
   q   +        / \   / \     |
      / \      p   q r   s    |
     r   s                    |
```

Exercise: What is the best layout for # pqr |= =| pq + pr + qr ?

De Morgan's Laws

____

~(p /\ q) |= =| (~p) \/ (~q)

~(p \/ q) |= =| (~p) /\ (~q)

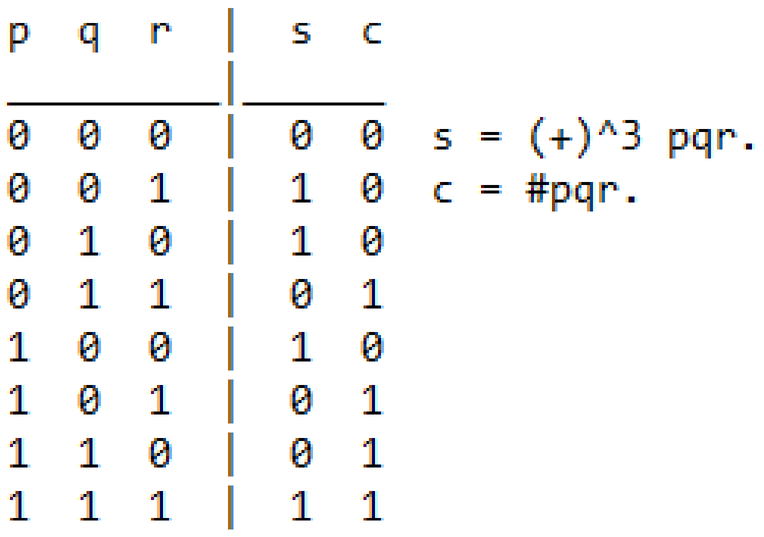
Computer Arithmetic

____

We start with addition and subtraction of binary integers.

Humans add binary numbers by acting as bit-serial adders. With a bit of practice, this becomes fairly automatic.

We can mathematically describe this process as repeatedly computing two 3-place Boolean functions, where the three arguments are the two bits 'p' and 'q' plus the carry-in bit 'r = c_in'. The two Boolean functions are 's' (sum) and 'c' (carry out). We can specify both functions by means of a table.



p	q	r	s	c
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

```
s = (+)^3 pqr.
c = #pqr.
```

Note: Efficient humans do _not_ add by doing table lookup.

Example:

```
    000111 =  7  (carries not shown)
  + 001101 = 13
    ------   --
    010100 = 20
```

To subtract, we compute a negation and then add.

Example:

```
    000111 =  7  (carries not shown)
  + 111010 = -6
    ------   --
  1|000001 =  1
```

The interpretation of bit patterns as decimal numbers obviously depends on the choice of semantics (natural number or two's complement), but the manipulation of bit patterns does not. There is one addition table above, not one table per semantics.

Computers do addition using registers. We can pretend our two examples take place in 6-bit registers. In the addition, there is no carry out from the result register. In the subtraction, there _is_ a carry out from the result register, but the bit pattern in the result register is entirely correct.

By definition, _overflow_ occurs when the result of an operation cannot be contained in the available hardware, in this case, a 6-bit register. When adding operands of opposite signs, overflow cannot occur.

The relationship between carry out and overflow is much simpler in natural-number semantics. Here, an n-bit register can store any natural number strictly less than 2^n. Therefore, if the true result of an addition is at least 2^n, then overflow has occurred. Moreover, in such a case, the addition algorithm will have produced a carry out.

Example:

```
    111111 = 63  (carries not shown)
  + 111111 = 63
    ------   --
  1|111110 = 62  (correct answer: 126)
```

Here, we have both carry out and overflow. Two's-complement semantics is slightly trickier.

What is the largest two's-complement integer that fits into a 6-bit register?

Answer: 2^5 - 1 = 31.

Therefore, we should be able to produce overflow by adding 16 to itself.

Example:

```
   010000 =  16  (carries not shown)
 + 010000 =  16
   ------
   100000 = -32
```

Even though there was no carry out from the result register, the correct answer (viz., 32) cannot fit into a 6-bit register---if we are using two's-complement semantics to interpret bit patterns. In natural-number semantics, of course, a 6-bit register can hold any natural number up to 63.

Returning to two's-complement semantics, we must say that overflow has occurred. The absence of a 7th bit means that overflow has occurred because the sign bit has been set with the _value_ of the result, rather than with the proper sign of the result. We have overwritten the sign bit.

Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This means a carry out has occurred into the sign bit.

A computer can easily have two separate add instructions, one suitable for general addition, with the hardware detecting and signaling overflow, and another suitable for memory-address calculation, with the hardware neither detecting nor signaling overflow. Memory addresses are natural numbers, so the rare overflow problems can be made the programmer's responsibility, saving a bit of work by the hardware.

Thus, that horrible-looking instruction

daddiu r1,r1,#-8

means "add integer -8 (a 16-bit signed _immediate_) to register 'r1', but ignore overflow". The initial 'd' is another story, of no particular interest. But there aren't two _addition algorithms_, only the two options of either paying attention to overflow or ignoring it, plus the obvious two options of how to interpret the resulting bit pattern as an integer. Since -8 is negative, it would be sign extended to give a 32-bit signed integer in two's complement. The addition would take place. But the sum, in the case of a memory-address calculation, would be interpreted with natural-number semantics. Again, programmer caution is advised.

Arithmetic in 16-bit registers can be shown with four hex digits.

Example:

```
  002c  (+44)
 +ffff  ( -1)
  ----   ---
  002b  (+43)
```

Multiplication

____

Multiplication is a bit trickier. There isn't one way to do it. The simplest to explain corresponds to what we learned in lower school (assume positive numbers):

```
- put multiplier in 32-bit register

- put multiplicand in 64-bit register

- initialize 64-bit product to zero

loop:   test lsb of multiplier

        if 1, add multiplicand to product

        shift multiplicand register 1-bit left

        shift multiplier register 1-bit right

        if not done, goto loop
```

Example:

```
    1101  (+13)  multiplicand
  x 1011  (+11)  multiplier
    ----
    1101
   1101
  0000
 1101
--------
10001111  (+143)  final result (product)
```

The simplest---but inefficient---hardware algorithm puts the multiplicand in a 64-bit register that is shifted left as each new result is added in. The sum is accumulated in a 64-bit product register. The multiplier goes in a 32-bit register that is shifted right to serially highlight each binary digit of the multiplier from right to left.

When the rightmost digit of the (currently shifted) multiplier is 1, we add the shifted multiplicand to the product (accumulator register). After this operation, we shift the multiplicand left and the multiplier right.

Now, the easiest way to multiply two signed numbers is to convert both operands to positive numbers, do traditional multiplication, and remember the signs.

A diagram may help:

```
     +--------------------+            We are multiplying two 32-bit
     |         |          |            numbers, but we are using
     |    multiplicand    |  64 bits   several 64-bit registers.
     |         |          |
     +--------------------+
               |
+--------+     |
|        |     |
|    +--------------------+
|    |                    |
|    |       adder        |  64 bits
|    |                    |
|    +--------------------+
|              |
|    +--------------------+            +--------------+
|    |                    |            |              |
|    |      product       |   <test>   |  multiplier  |  32 bits
|    |                    |            |              |
|    +--------------------+            +--------------+
|              |
+--------------+
```

The shifting is automatic. The test is primarily to determine if the shifted multiplicand is to be added to the product.

Overflow will occur if the (final) product is too big to fit into 32 bits.

We will not cover integer division or floating-point arithmetic.

Synthesis Exercises I

____

Definition: A set of connectives is _complete_ if they suffice to express an arbitrary Bollean function.

'(+)^3' is ternary addition modulo 2. '(+)^3 pqr' is true iff an odd number of 'p', 'q', and 'r', is true.

Exercise: Show that {'(+)^3', '/\', 'T', 'F'} is complete. (No proper subset is complete).

Exercise: Develop an argument to show that {'(+)^3', '~', 'T', 'F'} is not complete.

Exercise: Show {'|'} is complete.

Proof: ~p |= =| p | p.

                 p \/ q |= =| (~p) | (~q)

Since {'~', '\/'} is complete, and since both connectives can be simulated using only '|', {'|'} is complete. QED

Synthesis Exercises II

____

Logicians, and circuit designers, are concerned with 'completeness'. As defined earlier, a set of connectives is called _complete_ if an arbitrary Boolean function can be computed using only members of this set.

Well-known examples of complete sets are {'~', '/\'} and {'~', '\/'}. The synthesis game started by taking sets of connectives and asking if they were complete but may be played without reference to completeness.

1. Show that {'~', '-->'} is complete.

p --> q |= =| ~p \/ q

~p --> q |= =| p \/ q

2. Show that {'|'} is complete.

~p |= =| p | p

p \/ q |= =| ~p | ~q

3. 'I' is the ternary 'odd' connective. 'Ipqr' is true iff an odd number of its arguments is true.

Show that {'I', '/\', 'T', 'F'} is complete.

~p |= =| IpTF

4. 'P' is the ternary 'even' connective. 'Ppqr' is true iff an even number of its arguments is true.

Synthesize 'P' from {'~', '<-->'}.

Ppqr |= =| ~Ipqr

|= =| ~[(p + q) + r]

|= =| (p + q) <--> r

|= =| ~(p <--> q) <--> r

5. '2!' is the ternary 'precisely two' connective. '2!pqr' is true iff precisely two of its arguments are true. Show that {'2!', 'T'} is complete.

~p |= =| 2!pTT

p /\ q |= =| 2!pq(~T)

6. 'M' is the ternary 'minority' connective. 'Mpqr' is true iff a minority of its arguments is true. Show that {'M', 'F'} is complete.

~p |= =| MppF

p /\ q |= =| ~MpqF (#pqF)