1 When the Dog Bites, When the Bee Stings

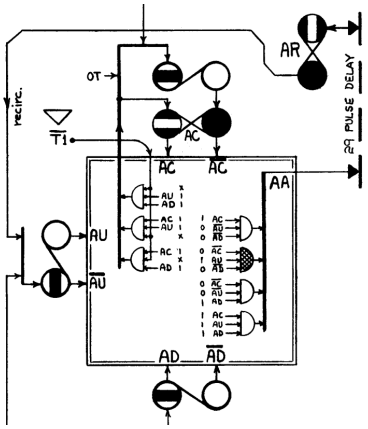
Converting non-inverting logic into inverting logic

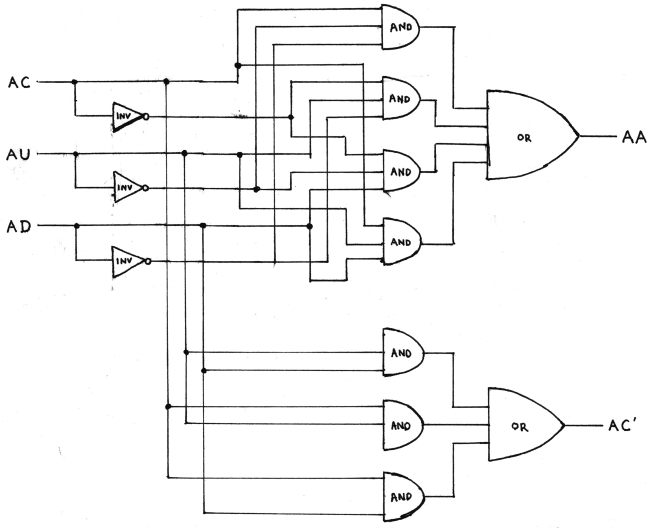
Several early computers from the 1950s, such as the Bendix G-15 and the Librascope LGP-30, used thermionic valves (vacuum tubes) and semiconductor diodes for their logical circuits. The first fully-transistorized computer, the IBM 608, used transistors and diodes.

An interesting property of diode logic is that it is *non-inverting*, which means that AND, OR, and NOT gates are the primitive elements, whereas NAND and NOR gates can only be made by inverting AND and OR.

By contrast, modern-day CMOS logic is *inverting;* NAND, NOR, and NOT are the primitive elements, and AND and OR can only be made by inverting NAND and NOR.

The circuit below is from the G-15. On the left is the original; on the right is the same circuit rewritten in modern notation. The way that logic is implemented in the G-15 constrains AND gates to come before OR gates,* so frequently the logic is naturally in a sum-of-product form. Using DeMorgan's law and other properties of Boolean algebra, translate this circuit from non-inverting logic to inverting logic while attempting to minimize the gate count. Can you guess what part of the machine this circuit is from?



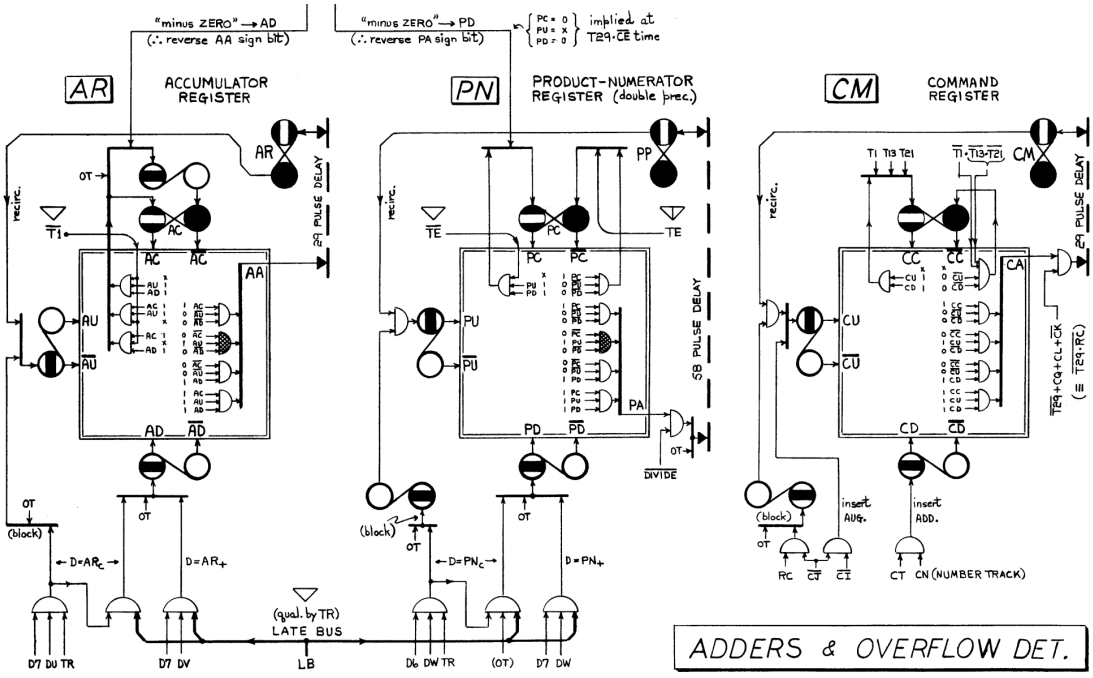


* Unless a buffer-inverter is used. AND and OR gates were made of diodes, but flip-flops and buffer-inverters required one or two vacuum tubes, which were physically larger and consumed a hundred times more power.

Solution

TODO

The circuit is from the adder; the bottom half of the schematic in which it appears is reproduced below.



2 I Didn't Write the Rest of These Problems...

Procedurally generating hardware

The following BSV function takes a doublet as a single argument and returns its reverse with the same type.

```
function Bit#(16) reverse16(Bit#(16) in);
    Bit#(16) rev;
    rev[ 0] = in[15];
    rev[ 1] = in[14];
    rev[ 2] = in[13];
    rev[ 3] = in[12];
    rev[ 4] = in[11];
    rev[ 5] = in[10];
    rev[ 6] = in[ 9];
    rev[ 7] = in[ 8];
    rev[ 8] = in[ 7];
    rev[ 9] = in[ 6];
    rev[10] = in[ 5];
    rev[11] = in[ 4];
    rev[12] = in[ 3];
    rev[13] = in[ 2];
    rev[14] = in[ 1];
    rev[15] = in[ 0];
    return rev;
endfunction
```

Write a new function that reverses input of any bit width. Its function signature should be

```
function Bit#(w) reverse(Bit#(w) in)
```

Solution

```
function Bit#(w) reverse(Bit#(w) in);
    Bit#(w) rev;
    for (Integer i = 0; i < valueOf(w); i = i+1) begin
        rev[i] = in[valueOf(w)-1-i];
    end
    return rev;
endfunction
```

3 ...So They Don't Have Snappy Titles

Judging the syntactic and typic correctness of BSV code

Six blocks of BSV code are shown below. Decide whether each block is syntactically and typically valid, and if it's not, recommend how it should be fixed. For the sake of this problem, assume that the function `add` exists, having the following function signature.

```
function Bit#(w) add(Bit#(w) a, Bit#(w) b, Bit#(1) c_in)
```

BLOCK 1

```
function Bit#(64) Add64(Bit#(64) a, Bit#(64) b);
    Bit#(64) Out = add(a, b, 0);
    return Out;
endfunction
```

VERDICT Invalid. Function names and variable names cannot be capitalized. Change `Add64` to `add64` and `Out` to `out`.

BLOCK 2

```
function Bit#(64) add64(Bit#(64) a, Bit#(32) b);
    return add(a, b, 0);
endfunction
```

VERDICT Invalid. The first two arguments to `add` must be of the same type. Emend line 2 to

```
    return add(a, signExtend(b), 0);
```

BLOCK 3

```
function UInt#(64) add64(UInt#(64) a, UInt#(64) b);
    return add(a, b, 0);
endfunction
```

VERDICT Invalid. Arguments `a` and `b` must be packed before being passed to `add` and the result must be unpacked before returning. Emend line 2 to

```
    return unpack(add(pack(a), pack(b), 0));
```

BLOCK 4

```
function Bool isZero(UInt#(8) a);
    return (a==0) ? 1 : 0;
endfunction
```

VERDICT Invalid. Literal `1` and `0` are of type `Bit#(1)`. Emend line 2 to

```
    return (a==0);                  or
    return (a==0) ? True : False;   or
    return unpack((a==0) ? 1 : 0);
```

BLOCK 5

```
function Bit#(32) shift32(Bit#(32) in, Bit#(5) shiftAmount);
    Bit#(32) out = in;
    for (Integer i = 0; i < shiftAmount; i = i+1) begin
        out = {1'b0, out[31:1]};
    end
    return out;
endfunction
```

VERDICT Invalid. Loop bounds must be known at compile time.

BLOCK 6

```
typedef enum {Red, Green, Blue} Color deriving {Bits, Eq};
function Color getColor(Bit#(2) a);
    return unpack(a);
endfunction
```

VERDICT Valid. Note, however, that the output is undefined when the input is `2'b11`.

4 "LOOKING TO HIRE QUESTION TITLE, FULL TIME..."

Parameterizing the reverse bit scanning function from Lab 3

In Lab 3 we wrote a function that computed the floor of the binary logarithm of its input (or equivalently, the index of the most-significant non-zero bit). Rewrite this function so that it works with inputs of arbitrary bit width.

Solution

```
function Bit#(TLog#(w)) floorLog(Bit#(w) in);
    Bit#(TLog#(w)) out = 0;
    for (Integer i = 0; i < valueOf(w); i = i+1) begin
        out = (in[i]==1) ? fromInteger(i) : out;
    end
    return out;
endfunction
```

5 "...CLEVER OR HUMOROUS IF POSSIBLE – $17/HR"

Writing a parameterized parity checker

Unlike integers, the parity of a bit vector usually refers to the oddness of the *sum* of its digits, so that the parity of a bit vector is `1` if it contains an odd number of `1`'s and `0` if it contains an even number of `1`'s.

Write a function that computes the parity of an n-bit vector, assuming that n is a power of 2. The propagation delay of your circuit should be $O(\log n)$.

Solution

```
function Bit#(1) parity(Bit#(n) in);
    Bit#(n) out = in;
    for (Integer s = valueOf(n)/2; s > 0; s = s/2) begin
        for (Integer i = 0; i < s; i = i+1) begin
            out[i] = out[2*i] ^ out[2*i+1];
        end
    end
    return out[0];
endfunction
```