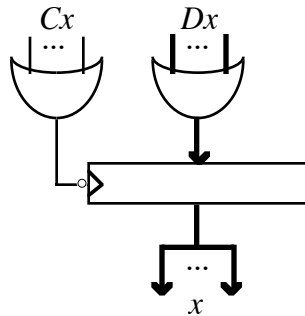


CSC258 Lecture Notes

High Level Circuit Design

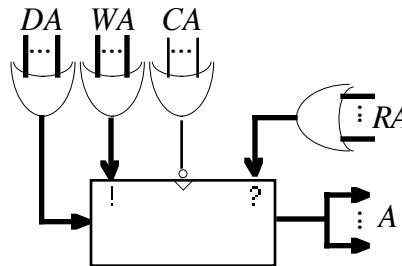
High level circuit design means circuit design by first writing a program in a high-level language (like C or Java), then automatically translating the program to a circuit. The automatic translator is sometimes called a “silicon compiler”. In these lecture notes we show one of many possible ways to translate programs to circuits.

For each variable in the program, we have a register. For example, if we have variable x in the program, then in the circuit we have



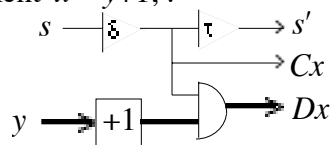
The wires labeled Cx (clock lines for x) and the wires labeled Dx (data lines for x) come from all places where variable x is assigned a value. When such a place is not actively making an assignment, we require that its Cx and Dx outputs be \perp because they are going into a \vee gate. The wires labeled x go to all places that use the value of variable x .

For each array in the program, we have a RAM. For example, if we have array A in the program, then in the circuit we have



The wires labeled DA (data for A), WA (writing address for A), and CA (clock for A) come from all places where an element of array A is being assigned a value. When such a place is not actively making an assignment, we require that its DA , WA , and CA outputs be \perp because they are going into a \vee gate. The wires labeled RA (reading address for A) come from all places that use the value of an element of array A . The wires labeled A go to all places that use the value of an element of array A . When such a place is not actively using array A , we require that its RA outputs be \perp because they are going into a \vee gate.

For each statement in the program we have a circuit with input s (start) and output s' (stop). A pulse on s starts the circuit working, and when it is finished, it sends out a pulse on s' . Here is the circuit for the assignment statement $x = y+1;$.

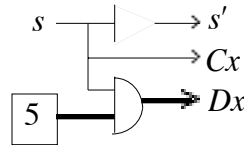


A pulse arriving on s goes through a delay δ that is just long enough to allow the incrementer to add 1 to the current value of variable y . This pulse then goes three places; it goes to a \wedge gate allowing the incremented value of y to go to the data input for variable x ; it also goes to the clock

input for x to cause x to latch onto the new value; and it goes through a delay τ that is just long enough for the new value of variable x to be latched, before signaling on s' that it is done.

An expression may have several operations and use the values of several variables. Its box needs input from all the variables used, and circuits for all the operations. That way we will have many adders and many subtractors, and so on, and the circuit will have maximum speed. Alternatively, an adder or other circuit can be shared among several expressions (by means of the function call circuitry which we present later), at the programmer's discretion.

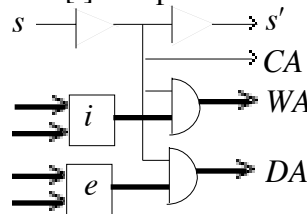
When the expression being assigned is a constant, there is a simplification. For example, the assignment $x = 5$; results in the circuit



where the box with 5 in it outputs the binary representation of 5, which is ...0000101, with 1s at bit positions 0 and 2. But there really is no point in taking 0s into the \wedge gates at Dx , so we really need only 2 wires coming from the 5 box and going through \wedge gates to Dx .

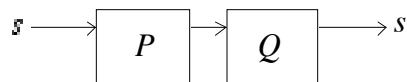
An expression may depend on an array element; if so, the reading address for that array element must be output from the expression circuit, conjoined with s , and routed to memory. There may be references to elements of several arrays, but at most one array element reference per array in the expression. If you want more than one element of the same array, you must break up the expression. For example, $x = A[i]+A[j]$; becomes $x = A[i]; x = x+A[j];$.

An array element assignment statement $A[i] = e$ produces the circuit



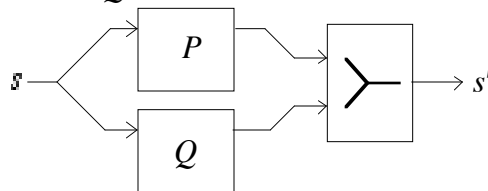
The unlabeled inputs to boxes i and e represent whatever variables are used in those two expressions.

When a statement P is followed sequentially by a statement Q in a program, the circuit for P is connected to the circuit for Q by connecting the s' output of P to the s input of Q .



The pulse that P sends out to say it is done is the pulse that starts Q .

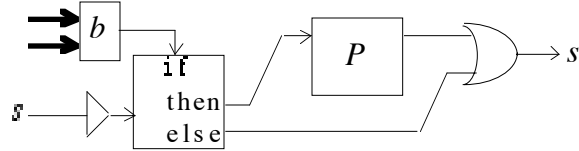
Most commonly used programming languages either do not have parallel composition, or have a parallel construct that is not easy to use. If a statement P is to be executed in parallel with a statement Q , the circuits for P and Q are connected like this:



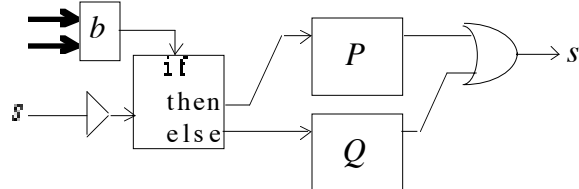
The start signal for the composition starts both P and Q at the same time. They may finish at different times. Each sends a pulse on its s' output when it finishes. These pulses go to a merge box, which emits a pulse only when it has received input pulses on both its inputs. Simultaneous access to different variables or arrays poses no problem. Even for the same variable, simultaneous

reads are no problem. But simultaneously reading and writing the same variable, or two simultaneous writes to the same variable, have unpredictable results.

The statement **if** (*b*) *P* produces the circuit

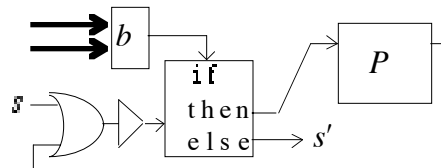


The unlabeled inputs to box *b* represent whatever variables are used in that expression. The statement **if** (*b*) *P* **else** *Q* produces the circuit

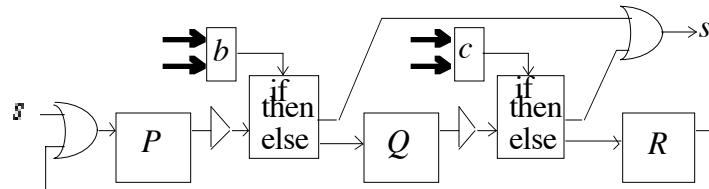


To make a circuit for a **switch** statement, the **if** circuit is generalized in the obvious way.

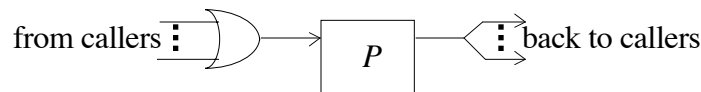
The circuit for **while** (*b*) *P* is as follows:



We can similarly construct a circuit for a loop whose exit condition comes at the end, or even in the middle. For example, **while**(1) {*P*; **if**(*b*) **break**; *Q*; **if**(*c*) **break**; *R*;} where *P*, *Q*, and *R* are any statements, produces the circuit

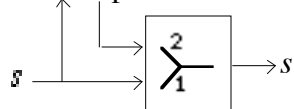


A procedure or void function or method is a unit of program that can be named, so that it can be called from several places, and return back to the place where it was called. Ignoring parameters for a moment, it has this circuit:



The calling points each become

to and from procedure



When a pulse arrives at a call, it is sent to the called procedure to start the procedure. It also goes into the 1-input of a 1-2-merge box. A 1-2-merge box emits a pulse only when it receives input pulses in a specific order: first into the 1-input, then into the 2-input. When the procedure finishes, it sends a pulse back to all calling points. When the pulse arrives at this calling point, it goes into the 2-input, causing the 1-2-merge box to emit a pulse. All those pulses that arrive at calling points that did not call the procedure go into the 2-input of the 1-2-merge box there, but there was no previous pulse in the 1-input there, so no pulse is emitted by that 1-2-merge box.

This implementation does not work for recursive calls in general, but it does work for tail-recursive calls. A parameter declaration can be treated exactly as though it were introducing a local variable

instead of a parameter, and parameter passing can be treated the same as assignment, at least for some kinds of parameters. For a function call that returns a result, we can again use the assignment statement circuit. There are better ways to do it, and there are many other language features we could cover, but perhaps that's enough to give you the main idea. Now let's put it all together with an example.

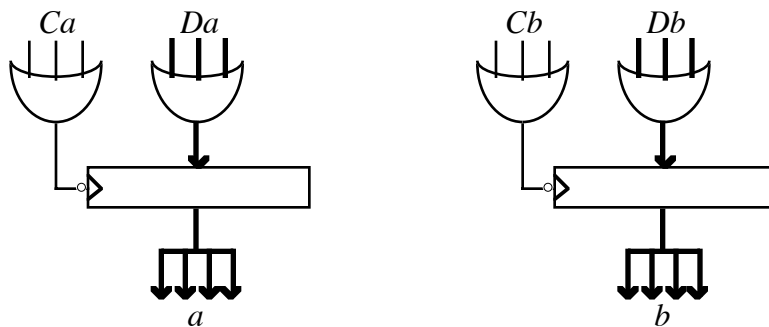
Here is a program in C; it should also be understandable to Java programmers.

```
int a, b;
```

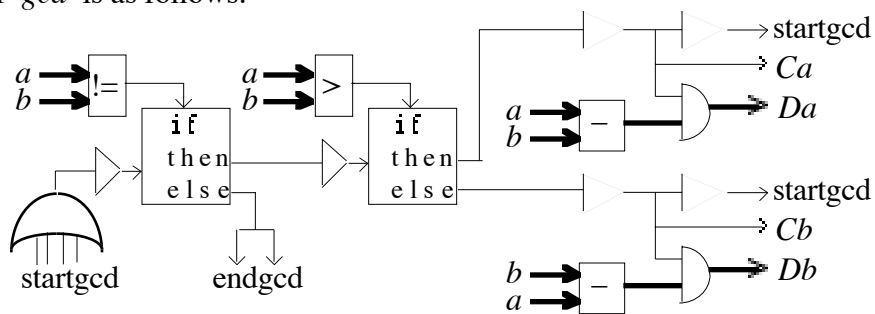
```
void gcd (void)
{ while (a!=b)
  if (a>b) a = a-b;
  else b = b-a; }
```

```
main ()
{ a = 3; b = 27; gcd ();
  a = 12; b = 30; gcd (); }
```

If a and b have positive integer values, then procedure gcd computes their greatest common divisor, and presents the answer as the value of both a and b . Variable a is assigned a value in 3 different places, and the value of variable a is used in 4 different places. Likewise for variable b . So here are their circuits.



The circuit for gcd is as follows:



The circuit for $main$ is a sequence of assignments and calls. Here is its circuit.

