CMPT 250 : Weeks 5 & 6 (Oct 3 to Oct 12)

1. UNSIGNED ADDITION (Continued)

To provide further improvement, the CPG is modified so that it does not actually compute a carry-out, c_{out}, but rather outputs two values:

$$G_0 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3$$

$$P_0 = p_0p_1p_2p_3$$

From these two values the carry-out is given by $G_0 + c_0P_0$.

Consider now the definition of carries c_4, c_8, and c_{12}. These are the carry inputs for the 2nd, 3rd and 4th modules of a 16 bit adder constructed from 4-bit CLAs.

$$c_4 = G_0 + c_0P_0$$

$$\begin{aligned} c_8 &= G_1 + c_4P_1 \\ &= G_1 + G_0P_1 + c_0P_0P_1 \end{aligned}$$

$$\begin{aligned} c_{12} &= G_2 + c_8P_2 \\ &= G_2 + G_1P_2 + G_0P_1P_2 + c_0P_1P_2 \end{aligned}$$

These equations are computed by a CPG whose input values are $G_0,...,G_2,P_0,...P_2,c_0$. These values in turn are the outputs from the 4-bit CLAs whose carry-out was replaced by G and P outputs. Therefore we can compute the carry-ins to each module more quickly by introducing a second level of CPGs as in the diagram: Analysis of this circuit shows that the longest path requires 8 levels of gates:

- one level to compute g_i,p_i
- two levels to compute G_i,P_i
- two levels to compute c_4,c_8,c_{12}
- two levels to compute the remaining carries
- one level to compute s_i as a function of c_i and p_i.

Thus we have reduced the delay by a factor of 4, from 32 gate levels to 8 gate levels in a 16 bit adder.

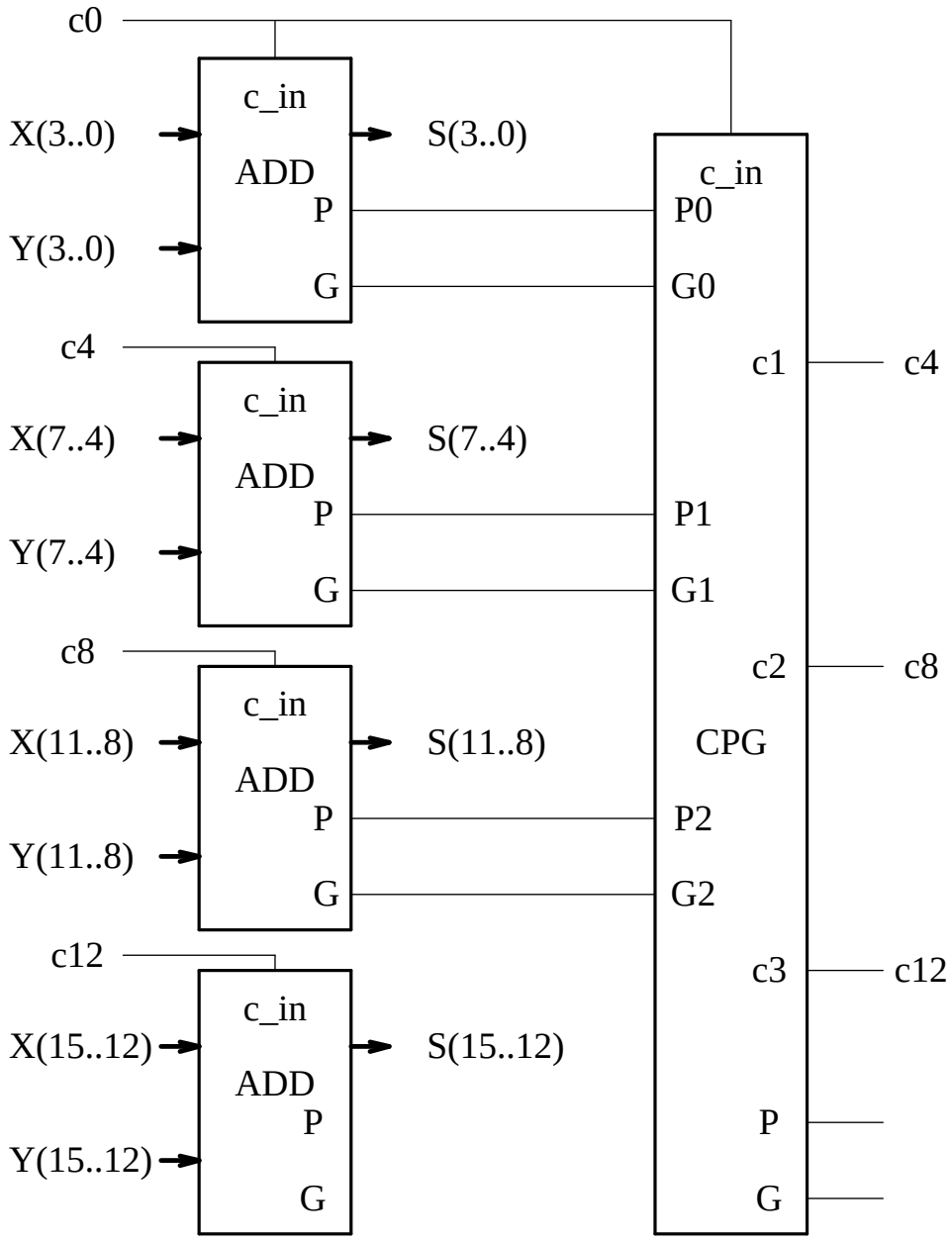

Figure 1-1: 16-bit addition with two levels of CPGs

2. BIT-VECTORS IN VHDL

The modelling of register behavior in VHDL may require the processing of bit vectors. In VHDL `bit_vector` is an unconstrained sequence of elements of type `bit`, and is available as a predefined datatype. However, usually one wants to constrain a `bit_vector` to a particular length, usually representing the size of a bus. Since `bit_vector` is already a `type`, a `subtype` declaration is required, along with a *range*. A range is a specification of a sequence of integer values, and in this instance is used to define the possible values a subscript may be assigned when indexing the bits of the `bit_vector`.

For example, a data subtype called `byte` can be declared by:

```
subtype byte is bit_vector(7 downto 0);
```

Here, the individual bits of any variable declared to be of type `byte` are indexed by subscripts 7 (the most significant bit) down to 0 (the least significant bit).

VHDL provides a looping construct, the `for statement` that permits one to access the bits of a `bit_string` subtype in an orderly way. The syntax of the `for` loop is:

```
for <index variable> in <range> loop
    statments defining the loop body
end loop;
```

Finally, VHDL provides `procedure` declarations to specify user-defined operations, and these can be used to specify operations involving `bit_vectors`. The syntax (simplified) for a `procedure` is:

```
procedure <procedure-name>(<parameter declaration>s)
local variable declarations
procedure body
end <procedure-name>;
```

A `<paramater declaration>` is similar to a `port` declaration, except that oridnary variables are declared rather than signals. It is also possible to include signals as parameters, but this will not be needed.

The syntax of a `<parameter declaration>` is as follows:

```
<parameter name> : <mode> <datatype>
```

Parameters passing values in to the procedure are defined to have mode `in`, while paramters returning values from the `procedure` are defined to have mode `out`. Variables serving both functions are given mode `inout`.

The following examples illustrate the implementation of a complementer and an incrementer behaviourally using `procedure declarations`.

Example 1: Complementer

```
entity COMPL is
   port(in_data: in byte;
        en: in bit;
        out_data: out byte);
end COMPL;

architecture behav of COMPL is
begin
compl_proc: process
   procedure complement(data_in: in byte,
                        data_out: out byte);
   variable i: integer;
   begin
     for i in 0 to 7 loop
        data_out(i) := not data_in(i);
     end loop;
   end complement;

   variable data: byte;
begin
   if en = '1' then
      complement(in_data,data);
      out_data <= data;
   else
      out_data <= in_data;
   end if;
   wait on in_data, en;
end process;
end behav;
```

Note that a temporary ordinary variable, `data` was used to avoid passing a signal variable as an argument for returning the complemented value, Although `in_data` is also a signal there are no "complications" associated with assigning signals to parameters of mode `in`. This is because input parameter values are passed using a "call by value" mechanism, whereas output parameters are linked via a "call by reference" mechanism.

Example 2: Incrementer

```
entity COUNTER is
   port(clk,clr: in bit;
        out_value: out byte);
end COUNTER;

architecture behav of COUNTER is
begin
   CTR_proc: process
   procedure increment(count inout: byte)
   variable i: integer;
   variable s,c: bit;
   begin
      c := '1';
      for i in 0 to 7 loop
         s := count(i) xor c;
         c := count(i) and c;
         count(i) := s;
      end loop;
   end increment;

   begin
      if clk = '1' then
         if clr = '1' then
            count := B"00000000";
         else
            increment(count);
         end if;
      out_value <= count;
      wait on clk;
   end process;
end behav;
```

In this example it was necessary to declare the parameter to be of mode `inout` since we are modifying the input parameter before returning it as an output parameter.

3. SEQUENTIAL CIRCUIT MULTIPLICATION

As was demonstrated previously, a significant amount of hardware is required for multiplying two 32 bit operands, the typical word size, using combinational logic. Multiplication can be implemented more simply, though less efficiently, with a sequential circuit based on repeated addition. The traditional algorithm consists of computing an n x 1 product, shifting it left one bit and adding it to an accumulated partial product. Alternatively, one can hold the n x 1 product fixed and shift the accumulated partial product to the right.

Both approaches are illustrated in the following example:

```
    0101    00000000
    0110    0000
    ----    --------
    0000    00000000
   0101     0101
  0101      --------
 0000       00101000
 -------    0101
00011110    --------
           00111100
           0000
           --------
           00011110
```

The second approach is defined formally in the ASM diagram on the next page. Note that if the format of the inputs is "unsigned binary" then the carry-out following addition should be used to replace the most significant bit when the number is shifted right.

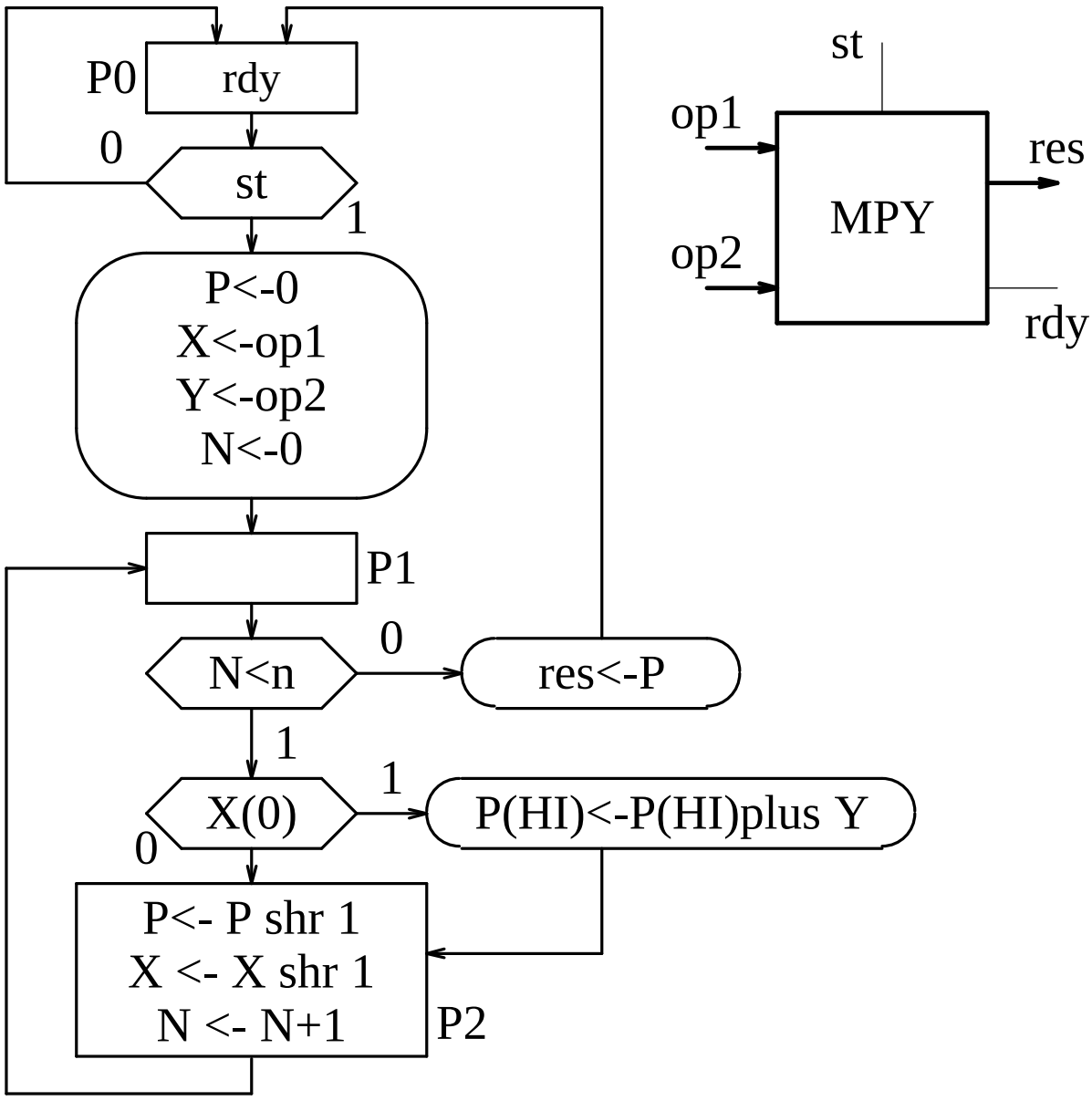

Figure 3-1: A sequential multiplier