Chapter 4
Control Unit Synthesis

Modern Electronic Design Automation tools have the capacity to synthesize the control unit from a finite state machine description, or even to extract and synthesize the control unit from a functional description of the complete circuit (Chap. 5). Nevertheless, in some cases the digital circuit designer can himself be interested in performing part of the control unit synthesis. Two specific synthesis techniques are presented in this chapter: command encoding and hierarchical decomposition [1]. Both of them pursue a double objective. On the one hand they aim at reducing the circuit cost. On the other hand they can make the circuit easier to understand and to debug. The latter is probably the most important aspect.

The use of components whose latency is data-dependent has been implicitly dealt with in Sect. 2.5. Some additional comments about variable-latency operations are made in the last section of this chapter.

4.1 Command Encoding

Consider the control unit of Fig. 2.6 and assume that *commands* is an m-bit vector, *conditions* a p-bit vector and *internal_state* an n-bit vector. Thus, the command generation block generates $m + 1$ binary function of $p + n$ binary variables. Nevertheless, the number s of different commands is generally much smaller than 2^m. An alternative option is to encode the s commands with a t-bit vector, with $2^t \geq s$. The command generation block of Fig. 2.6 can be decomposed into two blocks as shown in Fig. 4.1: the first one generates $t + 1$ binary functions of $p + n$ variables, and the second one (the command decoder) m binary functions of t binary variables.

A generic circuit-complexity measure is the number of bits that a memory (ROM) must store in order to implement the same functions. Thus, the complexity of a circuit implementing $m + 1$ functions of $p + n$ variables is

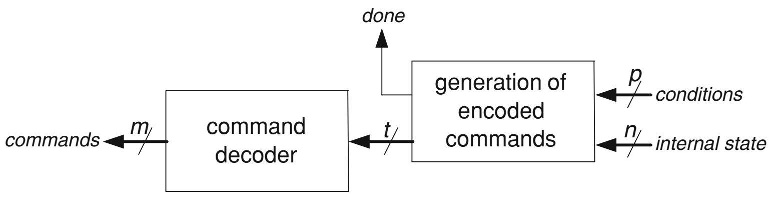

Fig. 4.1 Command encoding

$$(m+1)\cdot 2^{p+n}\text{bits}, \tag{4.1}$$

and the total complexity of two circuits implementing $t+1$ function of $p+n$ variables and m functions of t variables, respectively, is

$$(t+1)\cdot 2^{p+n}+m\cdot 2^{t}\text{bits}. \tag{4.2}$$

Obviously, this complexity measure only takes into account the numbers of outputs and inputs of the combinational blocks, and not the functions they actually implement.

Another generic complexity measure is the minimum number of LUTs (Chap. 1) necessary to implement the functions, assuming that no LUT is shared by two or more functions. If k-input LUTs are used, the minimum number of LUTs for implementing a function of r variables is

$$\lceil (r-1)/(k-1)\rceil\text{LUTs},$$

and the minimum delay of the circuit is

$$\lceil log_k r\rceil \cdot T_{LUT}$$

being T_{LUT} the delay of a k-input LUT.

The complexities corresponding to the two previously described options are

$$(m+1)\cdot \lceil (p+n-1)/(k-1)\rceil\text{LUTs} \tag{4.3}$$

and

$$(t+1)\cdot \lceil (p+n-1)/(k-1)\rceil + m\cdot \lceil (t-1)/(k-1)\rceil\text{LUTs}, \tag{4.4}$$

and the delays

$$\lceil log_k(p+n)\rceil \cdot T_{LUT} \text{ and } (\lceil log_k(p+n)\rceil + \lceil log_k t\rceil)\cdot T \tag{4.5}$$

Example 4.1
Consider the circuit of Sect. 2.5 (*scalar_product.vhd*, available at the Authors' web page). The commands consist of 26 bits: eight one-bit signals

`start_mult, load, en_xA, en_xB, en_zA, en_zB, en_R, sel_R`

and nine two-bit signals

`sel_p1, sel_p2, sel_a1, sel_a2, sel_sq, sel_xA, sel_xB, sel_zA, sel_zB.`

There are four binary conditions:

`start, k(m-1), mult_done, count < m-1`

and the finite-state machine has 40 states. Thus, $m = 26$, $p = 4$ and $n = 6$. Nevertheless, there are only $31 \ll 2^{26}$ different commands, namely

`nop, first, update, start1, end1, second, third, start4, end4, start5, end5, sixth, start7, end7, start8, end8, ninth, start10, end10, eleventh, twelfth, start13, end13, start14, end14, fifteenth, start16, end16, start17, end17, eighteenth,`

that can be encoded with $t = 5$ bits.

Thus, the complexities in numbers of stored bits (4.1 and 4.2) to be compared are

$$(m+1) \cdot 2^{p+n} = 27 \cdot 2^{10} = 27{,}648 \text{ bits}, \tag{4.6}$$

$$(t+1) \cdot 2^{p+n} + m \cdot 2^{t} = 6 \cdot 2^{10} + 26 \cdot 2^{5} = 6{,}976 \text{ bits}, \tag{4.7}$$

and the complexities in numbers of LUTs (4.3 and 4.4), assuming that 4-input LUTs are used, are

$$(m+1) \cdot \lceil (p+n-1)/3 \rceil = 27 \cdot \lceil 9/3 \rceil = 81 \text{ LUTS}, \tag{4.8}$$

$$(t+1) \cdot \lceil (p+n-1)/3 \rceil + m \cdot \lceil (t-1)/3 \rceil = 6 \cdot \lceil 9/3 \rceil + 26 \cdot \lceil 4/3 \rceil = 70 \text{ LUTs}. \tag{4.9}$$

The corresponding minimum delays (4.7) are

$$\lceil log_k(p+n) \rceil = \lceil log_4 10 \rceil = 2T_{LUT}, \tag{4.10}$$

$$(\lceil log_k(p+n) \rceil + \lceil log_k t \rceil) \cdot T_{LUT} = \lceil log_4 10 \rceil + \lceil log_4 5 \rceil = 4T_{LUT}. \tag{4.11}$$

The second complexity measure (number of LUTs) is surely more accurate than the first one. Thus, according to (4.8–4.11), the encoding of the commands hardly reduces the cost and increases the delay. So, in this particular case, the main advantage is clarity, flexibility and ease of debugging, and not cost reduction.

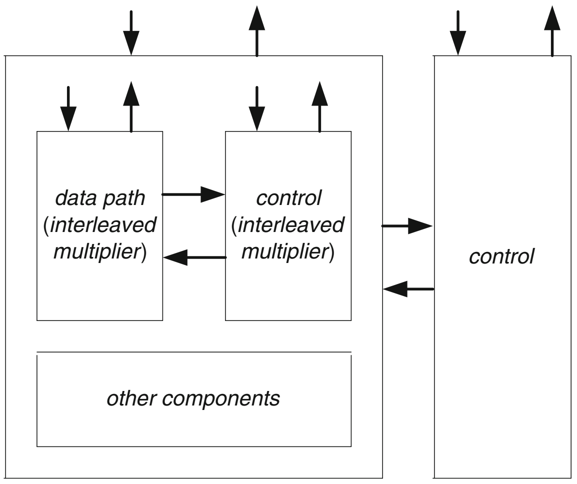
data path (interleaved multiplier)
control (interleaved multiplier)
other components
control

Fig. 4.2 Hierarchical circuit

4.2 Hierarchical Control Unit

Complex circuits are generally designed in a hierarchical way. As an example, the data path of the *scalar product* circuit of Sect. 2.5 (Fig. 2.18) includes a *polynomial adder* (XOR gates), a *classic squarer* and an *interleaved multiplier*, and the latter in turn consists of a data path and a control unit (Fig. 4.2). This is a common strategy in many fields of system engineering: hierarchy improves clarity, security, ease of debugging and maintenance, thus reducing development times.

Nevertheless, this type of hierarchy based on the use of previously defined components does not allow for the sharing of computation resources between several components. As an example, one of the components of the circuit of Sect. 2.5 is a polynomial adder, and the *interleaved multiplier* also includes a polynomial adder. A slight modification of the operation scheduling, avoiding executing field multiplications and additions at the same time, would allow to use the same polynomial adder for both operations. Then, instead of the architecture of Fig. 4.2, a conventional (flat) structure with a data path including only a polynomial adder could be considered. In order to maintain some type of hierarchy, the corresponding control unit could be divided up into a main control unit, in charge of controlling the execution of the main algorithm (*scalar product*) and a secondary control unit, in charge of controlling the execution of the interleaved multiplication.

Consider another, simpler, example.

Example 4.2
Design a circuit that computes

$$z = \sqrt{x^2 + y^2}.$$

The following algorithm computes *z*:

```
a := $x^2$;
b := $y^2$;
c := a + b;
z := square_root(c);
```

A first solution is to use three components: a squaring circuit, an adder and a square rooting circuit, for example that of Sect. 2.1. The corresponding circuit would include two adders, one for computing *c*, and the other within the *square_root* component (Fig. 2.3). Another option is to substitute, in the preceding algorithm, the call to *square_root* with the corresponding sequence of operations. After scheduling the operations and assigning registers to variables, the following algorithm is obtained:

```
1: r := $x^2$;
2: s := $y^2$;
3: c := r + s;
4: r := 0; s := 1;
5: r := r+1;
6: s := s + 2·r + 1;
7: greater := signb(s-c);
8: if greater = 0 then r := r+1; goto 6; else z := r; end if;
```

This algorithm can be executed by the data path of Fig. 4.3.

In order to distinguish between the main algorithm and the square root computation, the control unit can be divided up as shown in Fig. 4.4. A command decoder (Sect. 4.1) is used. There are eight different commands, namely

```
nop, r := $x^2$, s := $y^2$, c := r + s, (r, s) := (0, 1),
r := r+1, s := s + 2·r + 1, greater := signb(s - c);
```

that are encoded with three bits. The following process describes the command decoder:

```
command_decoder: PROCESS(command)
BEGIN
  CASE command IS
    WHEN "000" => sel_sq <= '0'; sel_a1 <= '0';
      sel_a2 <= "00"; cy_in <= '0'; sel_r <= '0';
      sel_s <= '0'; load <= '0'; en_r <= '0';
      en_s <= '0'; en_c <= '0'; en_signb <= '0';
    WHEN "001" => sel_sq <= '0'; sel_a1 <= '0';
      sel_a2 <= "00"; cy_in <= '0'; sel_r <= '0';
```

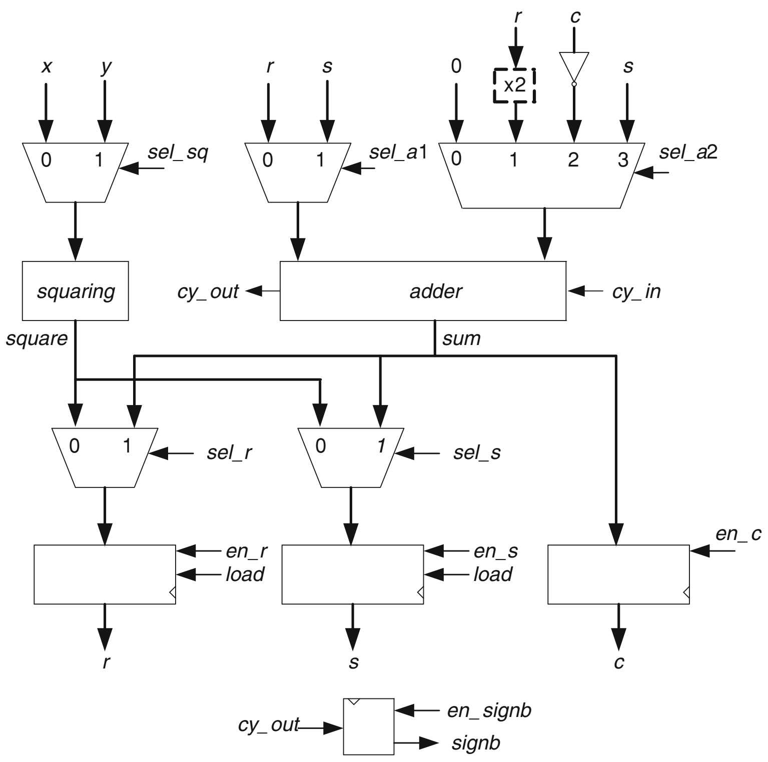

Fig. 4.3 Data path

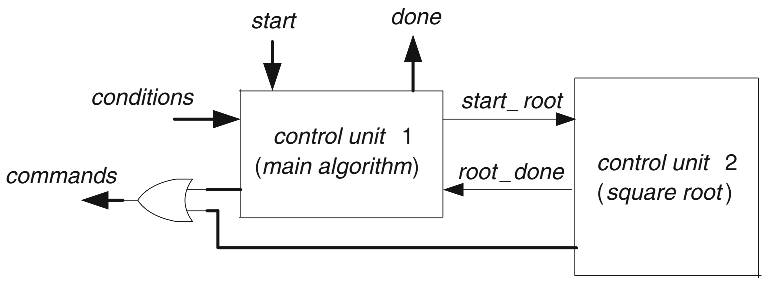

Fig. 4.4 Hierarchical control unit

```
      sel_s <= '0'; load <= '0'; en_r <= '1';
      en_s <= '0'; en_c <= '0'; en_signb <= '0';
    ...
  END CASE;
END PROCESS;
```

The two control units communicate through the *start_root* and *root_done* signals. The first control unit has six states corresponding to a “wait for start” loop, four steps of the main algorithm (operations 1, 2, 3, and the set of operations 4–8), and an “end of computation” detection. It can be described by the following process:

```
control_unit1:
PROCESS(clk, reset, current_state1, start, root_done)
BEGIN
  CASE current_state1 IS
    WHEN 0 =>
      command1 <= "000"; start_root <= '0'; done <= '1';
    WHEN 1 => IF start = '0' THEN command1 <= "000";
       ELSE command1 <= "001"; END IF;
       start_root <= '0'; done <= '1';
    WHEN 2 =>
      command1 <= "010"; start_root <= '0'; done <= '0';
    WHEN 3 =>
      command1 <= "011"; start_root <= '0'; done <= '0';
    WHEN 4 =>
      command1 <= "000"; start_root <= '1'; done <= '0';
    WHEN 5 =>
      command1 <= "000"; start_root <= '0'; done <= '0';
  IF reset = '1' THEN current_state1 <= 0;
  ELSIF clk'EVENT AND clk = '1' THEN
    CASE current_state1 IS
      WHEN 0 =>
        IF start = '0' THEN current_state1 <= 1; END IF;
      WHEN 1 =>
        IF start = '1' THEN current_state1 <= 2; END IF;
      WHEN 2 => current_state1 <= 3;
      WHEN 3 => current_state1 <= 4;
      WHEN 4 => current_state1 <= 5;
      WHEN 5 =>
        IF root_done = '1' THEN current_state1 <= 0; END IF;
    END CASE;
  END IF;
END PROCESS;
```

The second control unit has five states corresponding to operations 4, 5, 6, and 7, and “end of root computation” detection:

```
control_unit2: PROCESS(clk, reset, current_state2,
  start_root, signb)
BEGIN
  CASE current_state2 IS
    WHEN 0 => IF start_root = '0' THEN command2 <= "000";
      ELSE command2 <= "100"; END IF; root_done <= '0';
    WHEN 1 => command2 <= "101"; root_done <= '0';
    WHEN 2 => command2 <= "110"; root_done <= '0';
    WHEN 3 => command2 <= "111"; root_done <= '0';
    WHEN 4 =>
      IF signb = '0' THEN
        command2 <= "101"; root_done <= '0';
      ELSE command2 <= "000"; root_done <= '1'; END IF;
  END CASE;

  IF reset = '1' THEN current_state2 <= 0;
  ELSIF clk'EVENT AND clk = '1' THEN
    CASE current_state2 IS
      WHEN 0 => IF start_root = '1' THEN
          current_state2 <= 1; END IF;
      WHEN 1 => current_state2 <= 2;
      WHEN 2 => current_state2 <= 3;
      WHEN 3 => current_state2 <= 4;
      WHEN 4 => IF signb = '0' THEN current_state2 <= 2;
        ELSE current_state2 <= 0; END IF;
    END CASE;
  END IF;
END PROCESS;
```

The code corresponding to *nop* is 000, so that the actual command can be generated by ORing the commands generated by both control units:

```
command <= command1 OR command2;
```

A complete VHDL model *example*4_1.*vhd* is available at the Authors' web page.

Comments 4.1

- This technique is similar to the use of procedures and functions in software generation.
- In the former example, the dividing up of the control unit was not necessary. It was done only for didactic purposes. As in the case of software development, this method is useful when there are several calls to the same procedure or function.

- This type of approach to control unit synthesis is more a question of clarity (well structured control unit) and ease of debugging and maintenance, than of cost reduction (control units are not expensive).

4.3 Variable-Latency Operations

In Sect. 2.3, operation scheduling was performed assuming that the computation times t_{JM} of all operations were constant values. Nevertheless, in some cases the computation time is not a constant but a data-dependent value. As an example, the latency t_m of the field multiplier *interleaved_mult.vhd* of Sect. 2.5 is dependent on the particular operand values. In this case, the scheduling of the operations was done using an upper bound of t_m. So, an implementation based on this schedule should include "wait for t_m cycles" loops. Nevertheless, the proposed implementations (*scalar_product.vhd* and *scalar_product_DF2.vhd*) are slightly different: they use the *mult_done* flag generated by the multiplier. For example, in *scalar_product_DF2.vhd* (Sect. 2.5), there are several sentences, thus:

```
wait until mult_done = '1';
```

In an implementation that strictly respects the schedule of Fig. 2.14, these particular sentences should be substituted by constructions equivalent to

```
wait for tm cycles;
```

In fact, the pipelined circuit of Fig. 3.6 (*pipeline_DF2.vhd*) has been designed using such an upper bound of t_m. For that, a generic parameter *delta* was defined and a signal *time_out* generated by the control unit every *delta* cycles. On the other hand, the self-timed version of this same circuit (Example 3.4) used the *mult_done* flags generated by the multipliers.

Thus, in the case of variable-latency components, two options could be considered: a first one is to previously compute an upper bound of their computation times, if such a bound exists; another option is to use a *start-done* protocol: *done* is lowered on the *start* positive edge, and raised when the results are available. The second option is more general and generates circuits whose average latency is shorter. Nevertheless, in some cases, for example for pipelining purpose, the first option is better.

Comment 4.2
A typical case of data-dependent computation time corresponds to algorithms that include *while* loops: some iteration is executed as long as some condition holds true. Nevertheless, for unrolling purpose, the algorithm should be modified and the *while* loop substituted by a *for* loop including a fixed number of steps, such as *for i*

in 0 *to* $n-1$ *loop*. Thus, in some cases it may be worthwhile to substitute a variable-latency slow component by a constant-latency fast one.

An example of a circuit including variable-latency components is presented.

Example 4.3

Consider again Algorithm 2.3, with the schedule of Fig. 2.17, so that two finite field multipliers are necessary. Assume that they generate output flags $done_1$ and $done_2$ when they complete their respective operations. The part of the algorithm corresponding to $k_{m-i} = 0$ can be executed as follows:

```
cycle 1: start $x_A \cdot z_B$; start $x_B \cdot z_A$; $z_B := x_A + z_A$;
cycle 2: $z_B := z_B^2$;
cycle 3: $z_B := z_B^2$;
wait until ($done_1$ and $done_2$) = 1;
cycle i: $R := x_A \cdot z_B$; $x_B := x_B \cdot z_A$;
cycle i+1: start $x_A \cdot z_A$; start $R \cdot x_B$; $z_A := R + x_B$;
wait until ($done_1$ and $done_2$) = 1;
cycle j: $x_A := x_A \cdot z_A$; $x_B := R \cdot x_B$; $z_A := z_A^2$;
cycle j+1: start $x_P \cdot z_A$; $R := x_A^2$;
wait until $done_1$ = 1;
cycle k: $x_A := x_P \cdot z_A$;
cycle k+1 : $x_A := z_B$; $z_A := R$; $x_B := x_A + x_B$ ; $z_B := z_A$ ;
```

The following VHDL model describes the circuit:

```
mod_f_multiplier1: interleaved_mult PORT MAP (
  A => mult1a, B => mult1b, clk => clk, reset => reset,
  start => start_mult1, Z => product1, done => done1);
mod_f_multiplier2: interleaved_mult PORT MAP (
  A => mult2a, B => mult2b, clk => clk, reset => reset,
  start => start_mult2, Z => product2, done => done2);
a_squarer: classic_squarer PORT MAP (
  a => square_in, c => square);
done12 <= done1 AND done2;
PROCESS
BEGIN
  LOOP
    WAIT UNTIL start = '0'; WAIT UNTIL start = '1';
    xA <= one; zA <= zero; xB <= xP; zB <= one;
    start_mult1 <= '0'; start_mult2 <= '0'; done <= '0';
    sync;
```

```
FOR i in 1 TO m LOOP
  IF reset = '1' THEN EXIT; END IF;
  IF k(m-i) = '0' THEN
    zB <= xA XOR zA; mult1a <= xA; mult1b <= zB;
    start_mult1 <= '1'; mult2a <= xB; mult2b <= zA;
    start_mult2 <= '1'; sync;
    start_mult1 <= '0'; start_mult2 <= '0'; sync;
    square_in <= zB; sync;
    zB <= square; sync;
    square_in <= zB; sync;
    zB <= square; sync;
    WAIT UNTIL done12 = '1';
    R <= product1; xB <= product2; sync;
    zA <= R XOR xB; mult1a <= xA; mult1b <= zA;
    mult2a <= R; mult2b <= xB; start_mult1 <= '1';
    start_mult2 <= '1'; sync;
    start_mult1 <= '0'; start_mult2 <= '0'; sync;
    WAIT UNTIL done12 = '1';
    square_in <= zA; xA <= product1; xB <= product2;
    sync;
    zA <= square; sync;
    square_in <= xA; mult1a <= xP; mult1b <= zA;
    start_mult1 <= '1'; sync;
    R <= square; start_mult1 <= '0'; sync;
    WAIT UNTIL done1 = '1';
    xA <= product1; sync;
    xB <= xA XOR xB; xA <= zB; zA <= R; zB <= zA; sync;
  ELSE

      --same operations interchanging A and B
      ....
    END IF;
  END LOOP;
  done <= '1'; sync;
  END LOOP;
END PROCESS;
```

A complete model *unbounded_DF.vhd* is available at the Authors' web page. Other implementations, using latency upper bounds and/or pipelining or self-timing, are proposed as exercises.

4.4 Exercises

1. Design a circuit that computes $z = (x_1 - x_2)^{1/2} + (y_1 - y_2)^{1/2}$ with a hierarchical control unit (separate square rooter control units, see Example 4.2).
2. Design a 2-step self-timed circuit that computes $z = (x_1 - x_2)^{1/4}$ using two square rooters controlled by a *start/done* protocol.
3. Design a 2-step pipelined circuit that computes $z = (x_1 - x_2)^{1/4}$ using two square rooters, with a *start* input, whose maximum latencies are known.
4. Consider several implementations of the scalar product circuit of Sect. 2.5, taking into account Comment 2.2. The following options could be considered:

- hierarchical control unit;
- with an upper bound of the multiplier latency;
- pipelined version;
- self-timed version.

Reference

1. De Micheli G (1994) Synthesis and optimization of digital circuits. McGraw-Hill, New York