Potentially sharable resources should be within one module. Sometimes, the synthesis tools determine how to share potentially common resources, such as adders. Whenever, there is potential for such sharing, all of the related logic should be contained in one module, or even better, within one procedural block. In general, keep closely related portions of the design within one module. For example, synthesis will usually produce one adder and one multiplexor for the design below, even though two adders are specified (Figure 10-7):

```
always@(A or B or C or sel)
  D = sel ? A+C : A+B;
```

- Separate modules that will have different synthesis strategies. Only one synthesis strategy can be used per module.
- Keep modules as small as possible consistent with the above guidelines. Synthesis has a better chance of optimizing small designs than large ones. However, don’t take this to the extreme. Partitioning each design so that each flip-flop and input logic is in its own module will make the design unnecessarily unwieldy.

10.3 Controller Design Styles

The remainder of this chapter contains examples that illustrate these design principles via examples. Two of the major high-level design styles discussed in previous chapters are used. The first and last examples use an explicit control design style. In this style, the control and datapath portions of the design are explicitly separated and joined via well identified control points. This is the simplest style to use, especially for designers who are just learning their trade. The other two examples use an implicit control style. In this style, the control flow is specified in the same blocks as the datapath and the control points are thus implicit within these blocks. This design style generally leads to more compact and elegant code but requires a more sophisticated understanding of the relationship between code and hardware.

In many cases the area and performance can turn out to be very similar irrespective of the design style, but not always.

10.4 Example of Explicit Style - Motion Estimator

This is a typical example of an ASIC function – a piece of hardware purpose built to accelerate a function that would be performed too slowly in even the fastest general purpose
microprocessor. The example is taken from multimedia hardware and would be found in a
digital movie production system designed to produce MPEG video, e.g. a digital cam-
corder, or videocam.

If recorded and transmitted in its raw format, TV-quality video would require about 6 Mb
of storage or transmission bandwidth for every second. To reduce this number, a number
of compression techniques are used, that when combined can reduce this number by a fac-
tor of four or more. One of these techniques is motion estimation.

Motion estimation is best illustrated by a simple example. Consider the two subsequent
video frames shown in Figure 10.8. In this case, the black image that is common to the
frames is moved one up and one to the right in the second frame, with respect to the first.
Thus, the second frame can be recorded as being the same as the first frame except for
having a motion vector of (1,1) between them. It takes a lot less storage to store this
motion vector than the raw video bandwidth for the second frame.

![Figure 10-8: Consecutive frames in which the second can be coded as a motion
vector applied to the first.](image)

Candidates for compression via motion estimation are found by doing a search on a set of
possible motion vectors between two subsequent frames and looking for a good match.
This is illustrated in Figure 10.9. Two frames are compared by taking each 16 pixel by 16
pixel reference block in the first frame and searching for a good match for it within a
search window in the second frame. (The term 'pixel' stands for picture element and is
essentially one ‘dot’ on the screen. It is encoded with color and intensity information.) In
this case, the search window in the second frame is a 31 x 31 window centered around the position of the reference block in the first frame.

Thus a total of (31-15) x (31-15) = 256 comparisons must be performed, one for each possible match of a candidate to the reference block, from the candidate block in the top left corner to the candidate block in the bottom right corner of the search window. For each possible comparison block (starting at position (i,j)), a distortion figure is calculated.

\[
D(i, j) = \sum_{m=0}^{15} \sum_{n=0}^{15} |r_{m,n} - s_{m+i,n+j}|
\]

Where \((i,j)\) is the position of the candidate block being evaluated. The index \((i,j)\) will take on values from \((0,0)\) to \((15,15)\), that being all possible top-left coordinates of the candidate blocks. If the value for \(D(i,j)\) is zero, then a perfect match has been found and the appropriate motion vector can be calculated. In practice, if \(D(i,j)\) is small enough, then the difference between the two frames is often still encoded as a motion vector, as the human eye will not notice the resulting small irregularities in the video picture.
Since the reference block position is at the center of the search window, a coordinate transformation is needed in order to obtain the actual motion vector. For example, the position of the (0,0) candidate block with respect to the reference block is (-8,-8). Thus if \(D(i,j)\) provides the best match, the potential motion vector is \((i-8, j-8)\).

Determining motion vectors requires high levels of performance. For each comparison, 256 additions have to be performed. For each reference block, there are 256 possible candidate blocks, and there might be as many as 4,096 reference blocks in each frame of video data. Thus over 268 million additions have to be done for each frame of data. At 30 frames per second, the required processing rate becomes over 8 billion additions per second, clearly beyond any current microprocessor. However, a motion estimator capable of performing at this rate can be custom built in a small amount of parallel logic.

One possible architecture for a motion estimator is shown in Figure 10.10 above and is taken from reference [BK96]. The architecture consists of three small memories (one for the reference block data, and one each for the left and right hand sides of the search window).
Managing Complexity--Large Designs

dow data), 16 processing elements (PEs), a comparator unit and a control unit. The idea is that each PE will produce a new distortion figure \( D(i,j) \) for candidate block \((i,j)\) each 256 clock cycles. To do this, the reference block data is continuously fed into the left hand PE and pipelined through the ‘D’ registers from left to right. On each clock cycle, a datum from the search window is subtracted from the current datum from the reference window. The sequence of data from the search window memories, together with the pipelining of data from the reference block memory, is cleverly arranged to make sure that all of the PEs are 100% busy for most of the time. The sequencing of data from the memories, and the current subtract being done in each PE is given in Table 10.1.

<table>
<thead>
<tr>
<th>clk cyc</th>
<th>Input R</th>
<th>Input S1</th>
<th>Input S2</th>
<th>PE-0</th>
<th>PE-1</th>
<th>…</th>
<th>PE-15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( R_{0,0} )</td>
<td>( S_{0,0} )</td>
<td></td>
<td>( R_{0,0} - S_{0,0} )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( R_{0,1} )</td>
<td>( S_{0,1} )</td>
<td></td>
<td>( R_{0,1} - S_{0,1} )</td>
<td>( R_{0,0} - S_{0,1} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( R_{0,2} )</td>
<td>( S_{0,2} )</td>
<td></td>
<td>( R_{0,2} - S_{0,2} )</td>
<td>( R_{0,1} - S_{0,2} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>15</td>
<td>( R_{0,15} )</td>
<td>( S_{0,15} )</td>
<td></td>
<td>( R_{0,15} - S_{0,15} )</td>
<td>( R_{0,14} - S_{0,15} )</td>
<td>( R_{0,1} - S_{0,15} )</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>( R_{1,0} )</td>
<td>( S_{1,0} )</td>
<td>( S_{0,16} )</td>
<td>( R_{1,0} - S_{1,0} )</td>
<td>( R_{0,15} - S_{0,16} )</td>
<td>( R_{0,1} - S_{0,16} )</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>( R_{1,1} )</td>
<td>( S_{1,1} )</td>
<td>( S_{0,17} )</td>
<td>( R_{1,1} - S_{1,1} )</td>
<td>( R_{1,0} - S_{1,1} )</td>
<td>( R_{0,2} - S_{0,17} )</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>31</td>
<td>( R_{1,15} )</td>
<td>( S_{1,15} )</td>
<td>( S_{0,31} )</td>
<td>( R_{1,15} - S_{1,15} )</td>
<td>( R_{1,14} - S_{1,15} )</td>
<td>( R_{1,0} - S_{1,15} )</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>240</td>
<td>( R_{15,0} )</td>
<td>( S_{15,0} )</td>
<td>( S_{14,16} )</td>
<td>( R_{15,0} - S_{15,0} )</td>
<td>( R_{14,15} - S_{14,16} )</td>
<td>( R_{14,1} - S_{14,16} )</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>255</td>
<td>( R_{15,1} )</td>
<td>( S_{15,1} )</td>
<td>( S_{13,31} )</td>
<td>( R_{15,15} - S_{15,15} )</td>
<td>( R_{15,14} - S_{15,15} )</td>
<td>( R_{15,0} - S_{15,15} )</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>256</td>
<td>…</td>
<td>( S_{15,16} )</td>
<td></td>
<td>( R_{15,15} - S_{15,16} )</td>
<td>( R_{15,1} - S_{15,15} )</td>
<td>( R_{16,0} - S_{15,16} )</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>270</td>
<td>…</td>
<td>( S_{15,30} )</td>
<td></td>
<td>( R_{15,15} - S_{15,30} )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE 10-1:** Memory Sequencing and pixel computation sequence for producing the first 16 distortions.

The sequence of operations is clearly shown in Table 10.1. The first PE starts computing \( D(0,0) \) on clock cycle 0 and finishes the computation on cycle 255. (Not shown is how it will start computing \( D(1,0) \) on cycle 256.) Similarly, PE-1 starts computing \( D(0,1) \) on cycle 1 and finishes it on cycle 256, and PE-15 starts computing \( D(0,15) \) on cycle 15 and finishes it on cycle 270.

Consider the performance of this structure. Note that it takes 15 cycles for all the PEs to become busy after the search is started. Thus, in this pipelined fashion, all 256 possible block comparisons can be completed in \( 16(16x16) + 15 = 4,111 \) cycles, and 4,096 reference blocks could be searched for in \( 4,096 \times 4,111 = 16,429,056 \). To achieve a frame rate of 30
frames per second, would require a clock frequency of almost 500 MHz. If the hardware could not be built to operate that fast, then two of these units could be used in parallel to achieve the required throughput at 250 MHz.

Datapath Design

Now that the algorithm, architecture, and performance of the design has been determined, it is now necessary to determine the detailed design. In this case, the micro-operation sequence is captured in the algorithm above. A useful approach is to first determine the detailed structure of the datapath elements, from that determine the control points, and finally design the controller to toggle the control points correctly. The details of the processing element were seen in Figure 10.10. It consists of an accumulator register and some input logic consisting of an absolute subtractor, a saturating adder and two multiplexors. A saturating adder is an absolute adder (no negative numbers) that never overflows. Instead of overflowing, it simply saturates at FF. A saturating adder is used because it would be undesirable if the distortion was 0 because two larger numbers were added causing an overflow and a zero result. Overflow in unsigned arithmetic can be detected by looking at the carry flag. The reference memory data pipeline is illustrated at the top of the processing element. A possible Verilog level design for the PE is given below:

```verilog
module PE (clock, R, S1, S2, S1S2mux, newDist, Accumulate, Rpipe);
input clock;
input [7:0] R, S1, S2;// memory inputs
input S1S2mux, newDist;// control inputs
output [7:0] Accumulate, Rpipe;
reg [7:0] Accumulate, AccumulateIn, Difference, Rpipe;
reg Carry;

always@(posedge clock) Rpipe <= R;
always@(posedge clock) Accumulate <= AccumulateIn;
always@(R or S1 or S2 or S1S2mux or newDist or Accumulate)
begin
\ capture behavior of logic
difference = R – S1S2mux ? S1 : S2;
if (difference < 0) difference = 0 - difference;
// absolute subtraction
{Carry,AccumulateIn} = Accumulate + difference;
if (Carry == 1) AccumulateIn = 8’hFF;// saturated
if (newDist == 1) AccumulateIn = difference;
// starting new Distortion calculation
end
endmodule
```

Motion estimator processing element (PE)

Note the following features in this design:
• There are no need for ‘else’ statements in the non-clocked procedural block. Every output is assigned for all possible input variations, so there are no implicit latches.
• The fact that the addition of two 8-bit numbers produces a 9-bit result is captured in the assignment of the add to \{Carry, AccumulateIn\}.

The comparator is a simple structure. It mainly consists of registers to keep track of the best distortion and associated motion vector detected so far; some initialization logic for these registers, and an input multiplexor. It can be described in Verilog as shown in the next box.

```
module Comparator (clock, CompStart, PEout, PEready, vectorX, vectorY, BestDist, motionX, motionY);
    input clock;
    input CompStart; // goes high when distortion calculations start
    input [8*16:0] PEout; // Outputs of PEs as one long vector
    input [15:0] PEready; // Goes high when that PE has a new distortion
    input [3:0] vectorX, vectorY; // Motion vector being evaluated
    output [7:0] BestDist; // Best Distortion vector so far
    output [3:0] motionX, motionY; // Best motion vector so far

    reg [7:0] BestDist, newDist;
    reg [3:0] motionX, motionY;
    reg newBest;

    always @(posedge clock)
        if (CompStart == 0) BestDist <= 8’hFF; //initialize to highest value
        else if (newBest == 1)
            begin
                BestDist <= newDist;
                motionX <= vectorX;
                motionY <= vectorY;
            end

    always @(BestDist or PEout or PEready)
        begin
            newDist = PEout [PEready*8+7 : PEready*8);
            if ((PEready == 0) || (start == 0)) newBest = 0; // no PE is ready
            else if (newDist < BestDist) newBest = 1;
            else newBest = 0;
        end

endmodule
```

Now that the datapath has been designed, the control points can be established. Though a designer would not normally do this, they are explicitly listed below to emphasize this point.
The natural control strategy for this design is counter-based, not FSM-based. There are a number of reasons for this choice:

- Every control signal is generated on a cyclic basis. E.g. PEready[0] goes high every 256 clock cycles.
- There are no inputs to the controller apart from a start signal. Since no other decisions have to be made, there is no need for a Finite State Machine.

When designing a controller, simplicity is often a key to success. In this case, the simplest controller design strategy is to build one counter and decode logic to it to toggle the various control lines. Just about any other alternative will involve multiple counters and/or shift register and will neither be as small nor as simple to design and understand as this alternative. The counter has to go high enough to calculate all of the distortion vectors. The Verilog for the controller is listed in the box below.

### Control points

The next example to be developed is a direct mapped, write-through cache store with sub-block placement [H&P94 p 396, 442] It will be tested with the parameterized size reduced to match the reduced size MIPS processor which is the example to follow.