Exploring DCT Implementations

Gaurav Aggarwal
Daniel D. Gajski

Technical Report UCI-ICS-98-10
March, 1998

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425.
Phone: (714) 824-8059

gaurav@ics.uci.edu
gajski@ics.uci.edu

Abstract

The Discrete Cosine Transform (DCT) is used in the MPEG and JPEG compression standards. Thus, the DCT component has stringent timing requirements. The high performance which is required cannot be achieved by a sequential implementation of the algorithm. In this report, we explore different optimization techniques to improve the performance of the DCT. We discuss various pipelining options to further reduce the latency. We present a transformation of the algorithm that reduces the memory requirements and hence, reduces the cost of the implementation. We also describe RT-level implementations of the sequential, pipelined and memory optimized designs.

Contents

List of Figures

List of Tables

Exploring DCT Implementations

Gaurav Aggarwal Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425.
Phone: (714) 824-8059

Abstract

The Discrete Cosine Transform (DCT) is used in the MPEG and JPEG compression standards. Thus, the DCT component has stringent timing requirements. The high performance which is required cannot be achieved by a sequential implementation of the algorithm. In this report, we explore different optimization techniques to improve the performance of the DCT. We discuss various pipelining options to further reduce the latency. We present a transformation of the algorithm that reduces the memory requirements and hence, reduces the cost of the implementation. We also describe RT-level implementations of the sequential, pipelined and memory optimized designs.

1 Introduction

In recent years, a considerable amount of research has focussed on image compression. Compression plays a significant role in image/signal processing and transmission. Discrete Cosine Transform (DCT) is a type of transform coding that has a better compressional capability for reducing bit-rate as compared to other techniques like predictive or transform coding [1]. DCT is part of the JPEG (Joint Photographic Expert Group) and the MPEG (Motion Picture Expert Group) compression standards. Recently, DCT has been proposed as a component in the HDTV (high-definition television) standard that might replace the NTSC [2].

There are strict requirements on the performance of the DCT and the IDCT (Inverse Discrete Cosine Transform) components since they are part of the JPEG and MPEG encoders/decoders. The timing constraint on the DCT component can be computed from the MPEG standard [3]. Each picture in the MPEG standard consists of 720×480 pixels which implies that there are 1350 macroblocks/picture since each block is 16×16 pixels of the display. The picture rate is 30 frames/sec and hence, the MPEG decoder must process 40500 macroblocks/sec. Each macroblock consists of four 8×8 illuminance blocks and two 8×8 chrominance blocks. Thus, the rate is 40500×6=243000 blocks/sec. This translates into a timing constraint of 4115.22 ns/block. To be on the safe side, we impose a timing constraint of 4100 ns/block on our implementations for the DCT component.

The report is organized as follows. The *formal specification* of the DCT algorithm and a software implementation in C is given in Section 2. The sequential hardware implementation and optimizations like *loop unrolling*, *chaining* and *multicycling* are described in Section 3. The different *pipelining* options are discussed in Section 4. We then present a *memory optimized* algorithm and the pipelined version of this algorithm in Section 5. We compare the different optimization and pipelining techniques in Section 6. We finally conclude our exploratory study in Section 7.

2 Specification of the DCT

The generic problem of compression is to minimize the bit rate of the digital representation of signals like an image, a video stream or an audio signal. Many applications benefit when signals are available in the compressed form. Discrete Cosine Transform (DCT) is a way of transforming the signal from spatial domain to frequency domain which can then be compressed using some algorithm like run-length encoding [5]. The details of its use as a *lossy compression* algorithm are given in [1, 4].

In this section, we discuss the fundamentals of the Discrete Cosine Transform and give the formal definition of the algorithm. In Section 2.1 we give the mathematics and show how the algorithm can be decomposed into two matrix multiplications. This is followed by the C source code to compute the DCT in Section 2.2.

2.1 Mathematical Specification

As discussed above, DCT is a function that converts a signal from the spatial domain to the frequency domain. In this report, we primarily look at the two-dimensional transform that takes an image that has been digitized into pixels as its input. Most of the theory and implementation remains the same when it is used with other signals such as audio.

The formal specification of the 2-D DCT operation is as follows [4].

$$F_{uv} = \frac{c(m)c(n)}{4} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \left[f_{mn}\, cos\frac{(2m+1)u\pi}{2N} cos\frac{(2n+1)v\pi}{2N} \right]$$

where:

u, v = discrete frequency variables such that $(0 \leq u, v \leq N-1)$

f_{mn} = gray level of pixel at (m, n) in the $N \times N$ image $(0 \leq m, n \leq N-1)$

F_{uv} = coefficient of point (u, v) in spatial frequency domain

$c(0) = 1/\sqrt{2}$ and $c(m) = 1$ for $m \neq 0$.

In typical designs (like the MPEG standard), the image is sub-divided into 8×8 blocks of pixels. We also use a value of $N = 8$ in this example. Furthermore, let $CosBlock$ be a 8×8 matrix defined by

$$CosBlock_{un} = round(factor * (\frac{1}{8} cos\frac{(2n+1)u\pi}{16}))$$

An important property of the cosine transform is that the two summations are separable. Thus, it can be shown that

$$OutBlock = CosBlock \times InBlock \times CosBlock^T$$

where $InBlock$ is the input 8×8 block of image, f. $OutBlock$ is the output matrix in the frequency domain F and $CosBlock$ is defined above. The DCT can, thus, be modeled as two 8×8 matrix multiplications. These matrix multiplications (MM) can be serialized in time.

$$TempBlock = InBlock \times CosBlock^T \quad \text{(MM1)}$$
$$OutBlock = CosBlock \times TempBlock \quad \text{(MM2)}$$

The DCT transformation can then be modeled as two processes. The first process completes the first matrix multiplication and generates the 8×8 $TempBlock$ matrix. The results of this matrix multiplication is then used by the second process that generates the final output matrix, $OutBlock$. Both processes have an internal copy of the $CosBlock$ matrix.

2.2 DCT in C

DCT can be computed in software by doing two matrix multiplications. The code to compute the DCT in **C** is given below. The incoming image, $InBlock$, is an 8×8 array of integers and the DCT is in the frequency domain, $OutBlock$, again is an 8×8 array.

```
int   CosBlock[8][8];

void MatrixMult (int a[][8], int b[][8],
                 int c[][8]) {
   register int   i, j, k;

   for (i=0; i<8; i++)
      for (j=0; j<8; j++) {
         c[i][j] = 0;
         for (k=0; k<8; k++)
            c[i][j] += a[i][k] * b[k][j];
      }
}

void Transpose (int a[][8], int b[][8]) {
   register int   i, j;

   for (i=0; i<8; i++)
      for (j=0; j<8; j++)
         b[j][i] = a[i][j];
}

void DCT (int InBlock[][8],
          int OutBlock[][8]) {
   int  TempBlock[8][8], CosTrans[8][8];
   Transpose(CosBlock, CosTrans);

   /* TempBlock = InBlock * CosBlock^T */
   MatrixMult(InBlock,CosTrans,TempBlock);

   /* OutBlock = CosBlock * TempBlock  */
   MatrixMult(CosBlock,TempBlock,OutBlock);
}
```

Each matrix multiplication is a triple-nested loop. First the transpose of the CosBlock matrix is calculated and then the InBlock is multiplied with this transpose. The CosBlock is then multiplied with the result of the first multiplication. The code is obviously sub-optimal and several optimizations are possible. However, it has been used here to provide an unambiguous and simple definition to the DCT problem.

3 Design Space Explorations

The DCT component can be designed in a large number of ways. Each design incurs varying performance in terms of area and delay. In this report, we explore some of the options and discuss some common optimizations that can be used to speed up the design without incurring large area penalties.

We first tabulate the speed and cost parameters for component from our RT-level library in Section 3.1. We start the design exploration with a *sequential* design in Section 3.2. We then discuss some optimizations beginning with *loop unrolling* in Section 3.3 followed by *chaining* in Section 3.4. Then in Section 3.5 we describe multicycling.

3.1 RT-level Library Components

During our design exploration for the DCT we will implement the algorithm using register transfer level (RTL) components like registers, counters, adders, multipliers, multiplexers and so forth. These components are taken from a RTL library that maps these components to their gate level equivalents. The library also stores the delay and cost parameters associated with each component. The delay parameter is the critical path (in ns) of the component. The cost parameter is the area cost in number of transistors required for the component.

We list the components used for implementing the different designs in Table 1. This library is described in [6]. However, we have scaled down the delays of all components by a factor of 10 since technology improvements have increased the speed of gates [7]. The delay of an inverter is now 0.1 ns compared to 1 ns used by the library of [6]. The delay in the second column gives the worst case delay from input to output for a single signal change. The delay of pipelined components is represented by the delay of the longest stage. The delay of storage components is the average of read and write times. The third column gives the number of transistors required to implement each component. These numbers are based upon the cost incurred by the basic gates (nand, nor and inverter) as discussed in [6]. This RTL library will be used to determine the performance of the various designs discussed in following sections.

Table 1: Parameters for RTL components

Component	*Delay in* ns	*Cost in* trans
16 bit selector	0.4	224
32 bit selector	0.4	448
16 bit CLA adder	2.1	1074
32 bit CLA adder	2.9	2148
8 bit multiplier	5.6	3562
8 bit multiplier 2 stage pipe	3.5	4210
8 bit multiplier 4 stage pipe	2.7	5218
16 bit multiplier	8.8	11220
16 bit multiplier 2 stage pipe	5.4	12624
16 bit multiplier 4 stage pipe	3.5	15036
8 bit register	0.4	256
9 bit register	0.4	272
16 bit register	0.4	512
9 bit counter	2.5	414
64×16 RAM	3.5	6144
64×8 ROM	3.5	2048

3.2 Sequential Design

The DCT can be implemented as a sequential design in which only one operation is done during each clock cycle. This design is the slowest since there is no concurrency in execution of operations. However, this design is a good starting point for exploring different design alternatives. It is naturally developed from the software specification and has a simple controller.

As pointed out before the DCT consist of two matrix multiplications which are computationally identical. The sequential design does not attempt to do these matrix multiplications together. Thus, we discuss only one of matrix multiplication with the understanding that both the multiplications are done in the same manner. The Algorithmic State Machine (ASM)

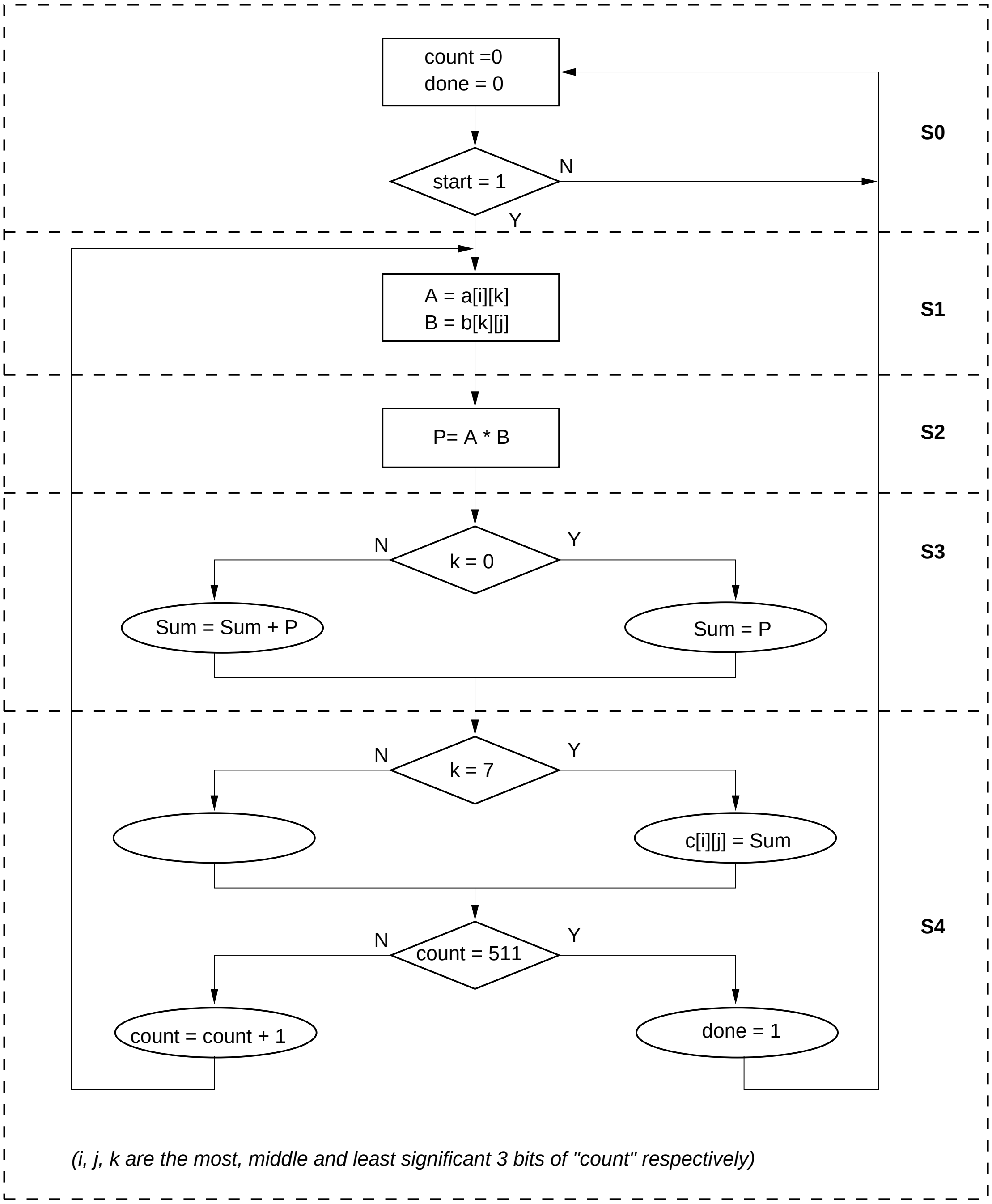

Figure 1: ASM chart for sequential matrix multiplication

chart [8] for an 8×8 sequential matrix multiplication is shown in Figure 1.

Sequential matrix multiplication can be implemented using RT-level components as shown in Figure 2. This design implements a single matrix multiplication and can be extended for implementing the actual DCT algorithm which consists of two serialized matrix multiplications.

A behavioral model of the sequential design in VHDL is given in Appendix A. The test bench for verifying the design is given in Appendix C. The structural model of DCT is comprised of a controller and a datapath. The schematics and VHDL model are given in Appendix B. Note that Figure 23 in Appendix B gives the complete datapath while Figure 2 only describes a single matrix multiplication. However, the basic datapath remains the same. The complete DCT datapath has an extra memory, and the input to the `A` register comes from a multiplexer since the source is `InBlock` memory during the first matrix multiplication and `TempBlock` memory during the second matrix multiplication. Furthermore, Figure 2 includes the controller while Figure 23 in the Appendix does not include the controller. Figure 2 is used as an example and for calculating the hardware costs of the design.

The ASM chart for the sequential matrix multiplication design can be partitioned into 4 states. Each state corresponds to a clock cycle. The clock period is, then, determined by the maximum delay in any of the states. Thus, from our RT-level component library, we can determine that the required clock period will be $8.8 + 0.4 = 9.2$ns (as determined by the slowest state that has the multiplier). This leads us to calculate the time required for the entire DCT computation as follows. Note that each loop has 512 iterations and hence the total number of iterations is 512×2=1024.

# states	= 4
clock period	= 9.2 ns
# iterations	= 1024
Latency	= #states×clock×#iterations
	= 4×9.2×1024 = 37683.2 ns

The cost of the design in terms of transistors can be calculated using the cost values of the RTL components from Table 1. The cost of the datapath components is as follows: `CosBlock`=2048, `A`=512, `B`=512, `multiplier`=11220, `P`=1024, `Sum`=1024, `adder`=2148, `selector`=448, `TempBlock`=6144, `counter`=414. Thus, the total is 25494 transistors. We do not include the storage requirements for the `InBlock` and `OutBlock` matrices. These matrices are the input and output of the DCT component and may be accessed using RAMs or FIFOs in sequential or burst modes. The controller has a 4 bit register (128), a decoder (168) and some gates (102). The total is 398 transistors. Thus, the design is clearly data-dominated since there is an order of magnitude difference between the number of transistors required for the datapath and the number for controller. We, therefore, ignore the controller cost from consideration. Hence, the cost of the sequential design is 25494≈25K transistors (since the transistor cost is not exact, and we are interested only in comparing designs, we round off the cost to 1000 transistors).

It takes 37683.2 ns for the sequential design to compute the DCT for an 8×8 input image block. This design is obviously too slow. We next explore different optimizations techniques to reduce the latency of the design.

3.3 Loop Unrolling

A design often spends most of the computation inside a loop. Such designs can typically be speeded up by *unrolling* the loop n of times. This implies that the loop is modified so that n iterations in the loop of the original design are now done in 1 iteration of the loop. Thus, the total number of iterations in the loop of the design go down by $\frac{1}{n}$ (with appropriate boundary conditions). The obvious requirement for unrolling the loop is availability of n times the hardware since the n iterations which were done sequentially in the original loop are now done concurrently.

Consider unrolling the inner-most loop of the sequential matrix multiplication by 2. The ASM chart for *2-unrolled* is shown in Figure 3. Two values each from the `A` and the `B` matrices (`A[i][k], A[i][k+1]` and `B[k][j], B[k+1][j]` respectively) are read concurrently. In the next clock cycle, `A[i][k]` is multiplied by `B[k][j]` and `A[i][k+1]` by `B[k+1][j]` concurrently. It is clear from the ASM chart that each iteration of the loop does double the computation and the number of iterations is reduced by half since `count` is incremented by 2.

The effect of unrolling the loop is an increase in the hardware requirements. Twice the number of registers, multipliers and adders are required. If the loop

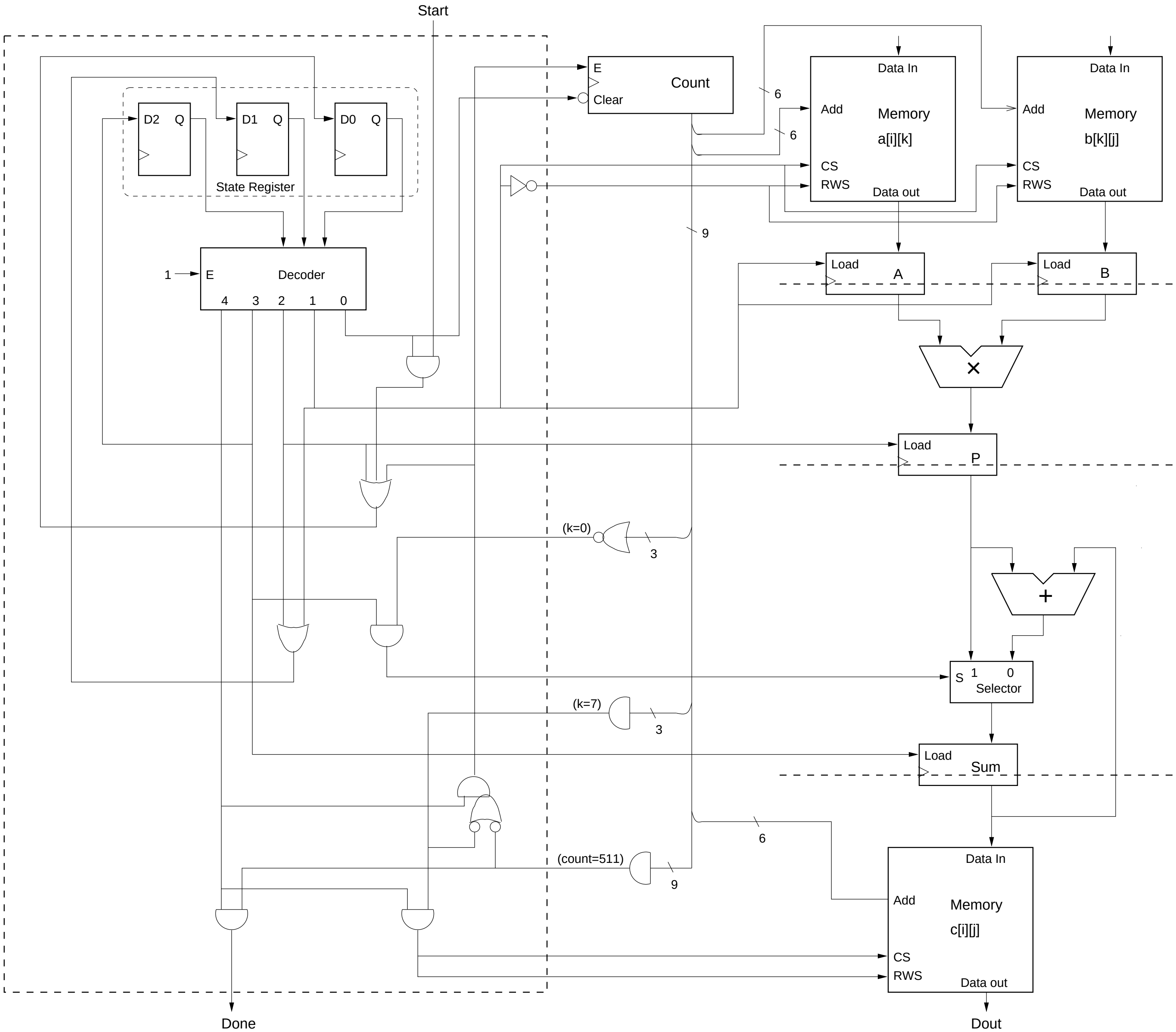

Figure 2: Design for sequential matrix multiplication

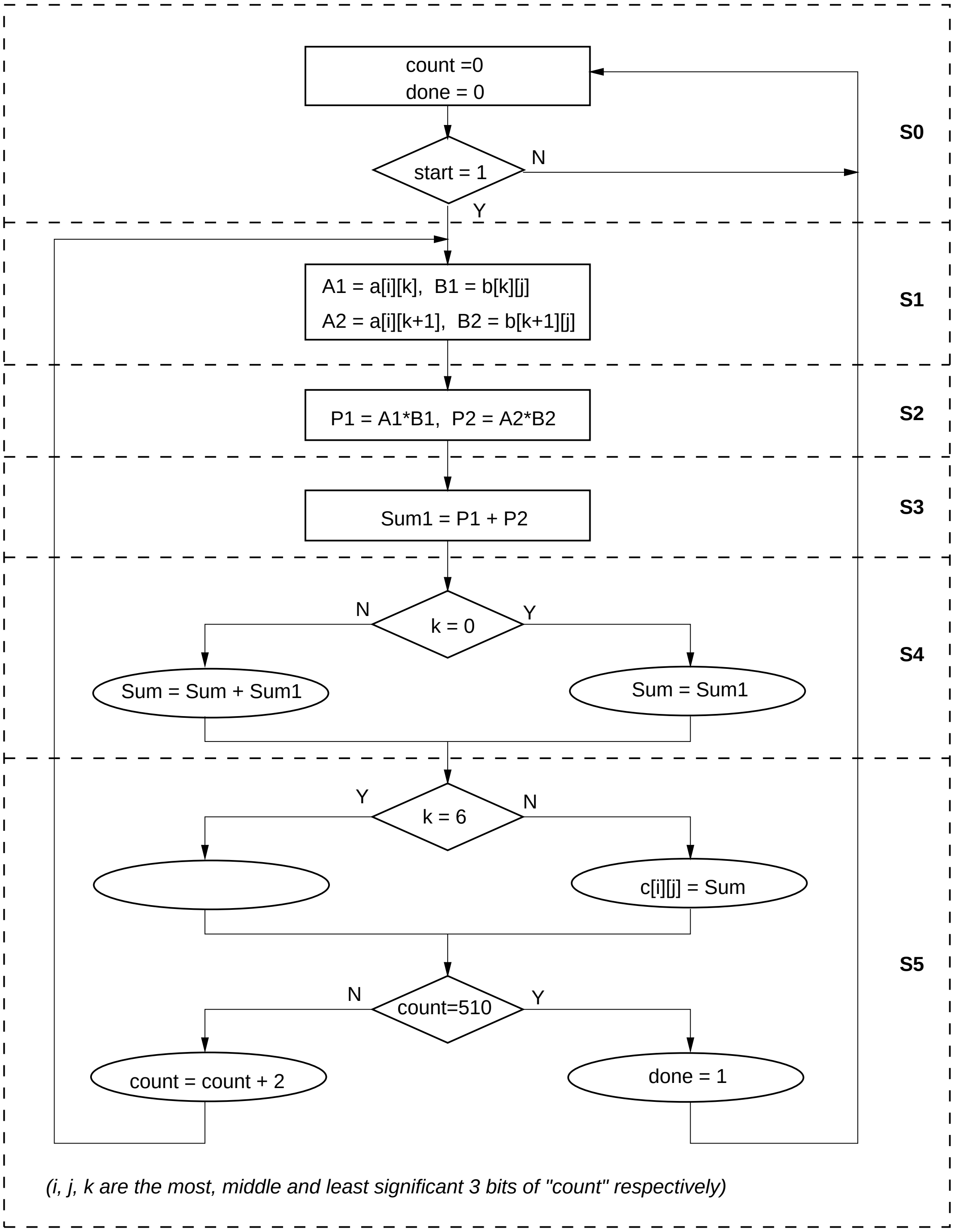

Figure 3: ASM chart for 2-unrolled matrix multiplication

includes memory accesses (as is the case here), unrolling a loop will also increase the bandwidth requirements for the memory and multi-port memories will have to be used. The datapath for a *2-unrolled* matrix multiplication design is shown in Figure 4. The hardware requirements for the loop unrolled implementation increases as shown in the figure. A dual-port memory is required to read two data values in the same clock cycle. In addition, two multipliers, two adders and four registers are required. Consequently, the hardware cost increases, as evaluated below.

The number of states in the loop increase to 5 compared to 4 in the sequential non-optimized design, as shown in the ASM chart of Figure 3. However, the number of iterations required is reduced by half. Thus, each matrix multiplication requires 256 iterations instead of 512. The total number of iterations for DCT is then, 256×2=512. The clock period is the same as that for the sequential design. It is $8.8 + 0.4 = 9.2$ns (determined by the slowest state that has the multiplier). The latency of the design can then be computed as follows.

$$\begin{aligned}
\#\ \text{states} &= 5 \\
\text{clock period} &= 9.2\ \text{ns} \\
\#\ \text{iterations} &= 512 \\
\text{Latency} &= \#\text{states}\times\text{clock}\times\#\text{iterations} \\
&= 5\times 9.2\times 512 = 23552\ \text{ns}
\end{aligned}$$

The additional cost of the 2-unrolled design can be computed as follows: `dual-port RAM`=8092-6144, `A2`=512, `B2`=512, `multiplier`=11220, `P2`=1024, `adder`=2148, `Sum1`=1024. Thus, the total cost for the 2-unrolled design is 43882≈44K transistors.

3.4 Chaining

It is almost never the case that the delays of states in a design be identical. The delays are, most often, not even nearly equal. However, the clock period is equal to the worst register-to-register delay. The worst register-to-register delay path goes through the slowest functional unit and hence, other faster functional units use only a part of the clock period and are idle for the remaining. This clearly slows down the design and is inefficient since some units sit idle.

A common technique to reduce this wastage is *chaining* of two or more functional units. Consecutive states with functional units whose total delay is comparable to the maximum delay (the clock period) can be combined together into one state. This has the effect of reducing the number of states in the loop and hence, improving performance. It is not necessary to keep the clock period same. It may be possible to chain states even if the cumulative delay is more than the original clock period if this still leads to a net improvement in the performance which is determined by both the number of states and the clock period (Latency = #states × clock × #iterations).

The basic idea behind chaining is shown in Figure 5. The multiplier in Figure 5(a) is the slowest component. States `S3` and `S4` take much less time and are idle for a part of the clock duration since the delay of an adder is less than that of a multiplier. The two states can be chained into one state as shown in Figure 5(b). The number of states decreases by one and the register between the two states is removed. The controller is also modified. The net effect is an improvement in the performance.

Chaining can be done in the 2-unrolled design from Section 3.3. The ASM chart for the *unrolled* and *chained* design is shown in Figure 6. The number of states is reduced from 5 in the only-unrolled design to 4 because of chaining. The clock cycle remains 9.2 ns since two additions and selection can be done within this time period. The ASM chart is for a single matrix multiplication. The latency of a DCT component that is 2-unrolled and chained can be computed as follows. Note that each loop has 256 iterations and hence the total number of iterations is $256 \times 2 = 512$.

$$\begin{aligned}
\#\ \text{states} &= 4 \\
\text{clock period} &= 9.2\ \text{ns} \\
\#\ \text{iterations} &= 512 \\
\text{Latency} &= \#\text{states}\times\text{clock}\times\#\text{iterations} \\
&= 4\times 9.2\times 512 = 18841.6\ \text{ns}
\end{aligned}$$

Chaining only reduces a single register (1024 transistors) from the design. Hence, it does not reduce the cost by a large margin. The cost for the 2-unrolled chained design is consequently 41K transistors.

3.5 Multicycling

In the previous section, we discussed how operation may be chained so as to reduce the number of states in the iteration loop if the delays of the states are not nearly equal. Another possible alternative is to split the longer state into 2 or more states. This is called *multicycling* because the operation now takes multiple

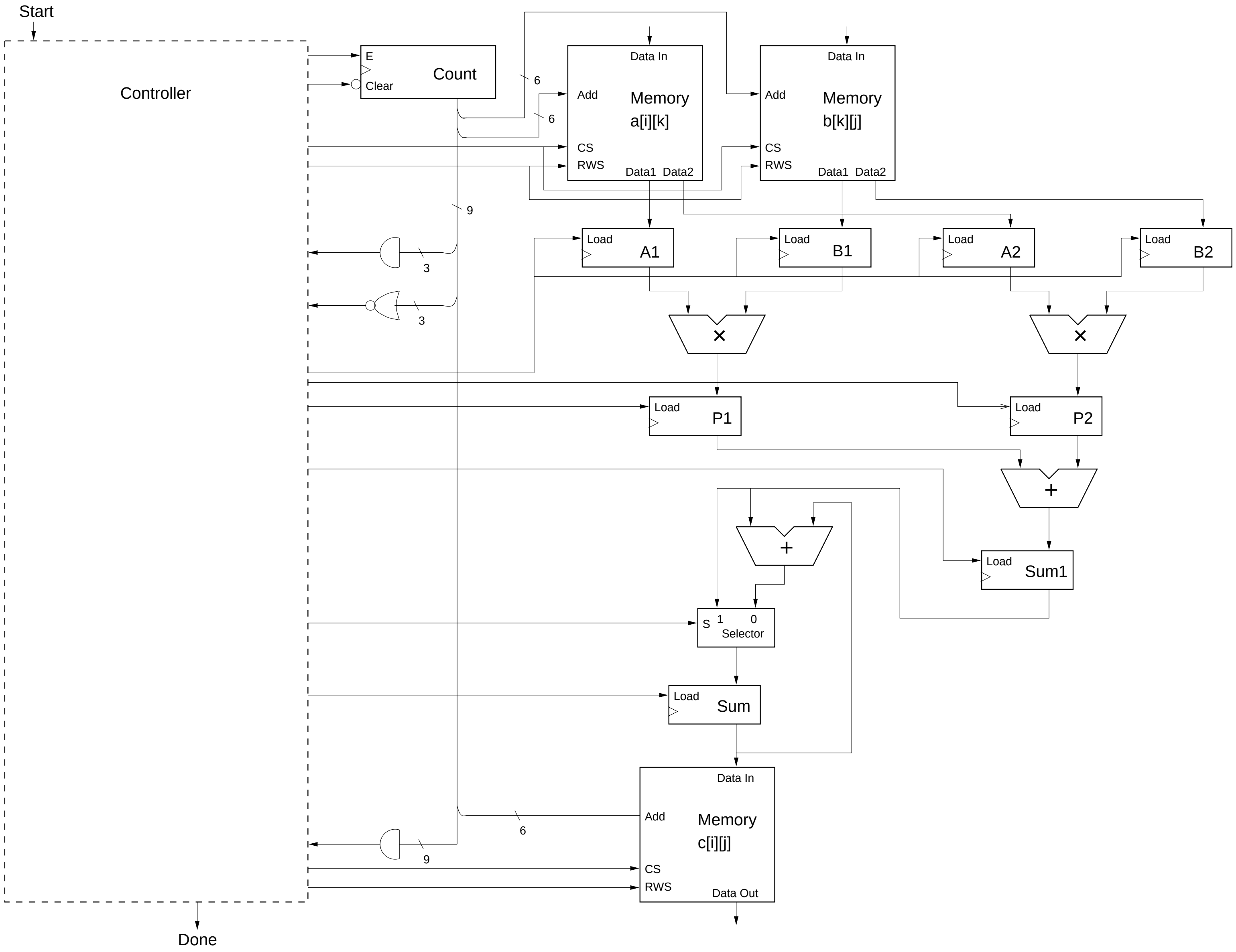

Figure 4: Datapath for 2-unrolled matrix multiplication

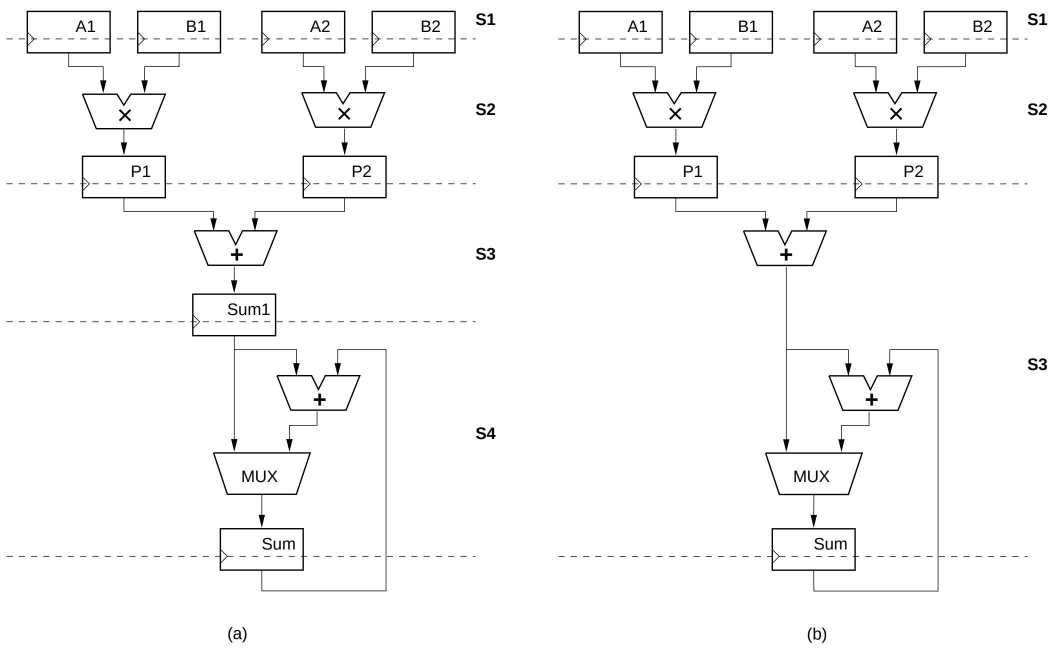

Figure 5: Chaining *short* operations: (a) before chaining (b) after chaining

clock cycles to complete. Even though the number of states increases as a result of multicycling, there can still be an advantage due to the decrease in the clock period.

Multicycling is useful because it decreases the clock rate which may be based on other system parameters. It can be used together with loop unrolling and chaining. Multicycling may be useful even when it does not lead to a large reduction in the clock period if the number of states is large. This is because the increase in number of states is more than offset by even the small reduction in the clock period.

The ASM chart for an unrolled, multicycled design is shown in Figure 7. The multiplier has been multicycled into 2 states `S2` and `S3` since multiplication the slowest operation. Note that multicycling does not required a change in the datapath of the design. Only the controller needs to be modified. An extra state is added in the controller and the output of the multiplier is latched one clock cycle later. Thus, the design can be operated at a clock period of 4.6 ns (= 9.2÷2) since the multiplier gets two cycles for completion. This leads us to compute the overall latency of the DCT design as follows. Note that each loop has 256 iterations and hence the total number of iterations is $256 \times 2 = 512$.

# states	= 6
clock period	= 4.6 ns
# iterations	= 512

$$\text{Latency} = \#\text{states}\times\text{clock}\times\#\text{iterations}$$
$$= 6\times 4.6\times 512 = 14131.2 \text{ ns}$$

The cost of the multicycled design remains the same as that for a 2-unrolled design, i.e., 42K transistors.

4 Pipelining Alternatives

In the previous sections, we optimized the design by techniques such as *loop unrolling*, *chaining* and *multicycling*. This improved the performance of the DCT design. Further improvement is possible using the standard technique of *pipelining*. In the following sections, we try to improve the performance by pipelining the DCT core which consists of the two matrix multiplications as given in lines 59–110 of Appendix A.

There can different levels of pipelining. We describe process pipelining in Section 4.1. We then look at loop pipelining in Section 4.2 and functional unit pipelining in Section 4.3. In Section 4.4 we describe a design in which the second matrix multiplication is started before the first one is completed. Finally, in Section 4.5

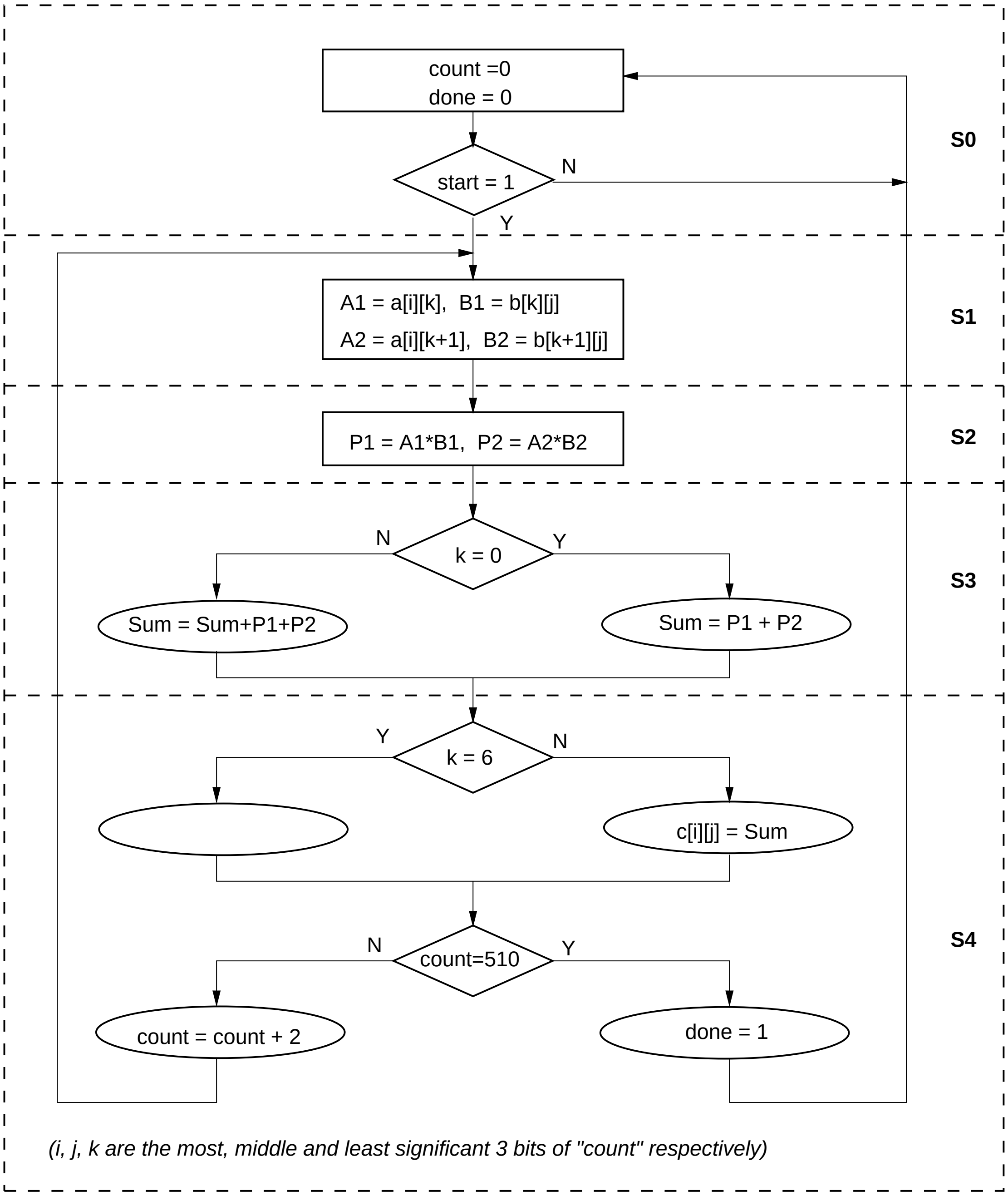

Figure 6: ASM chart for unrolled and chained matrix multiplication

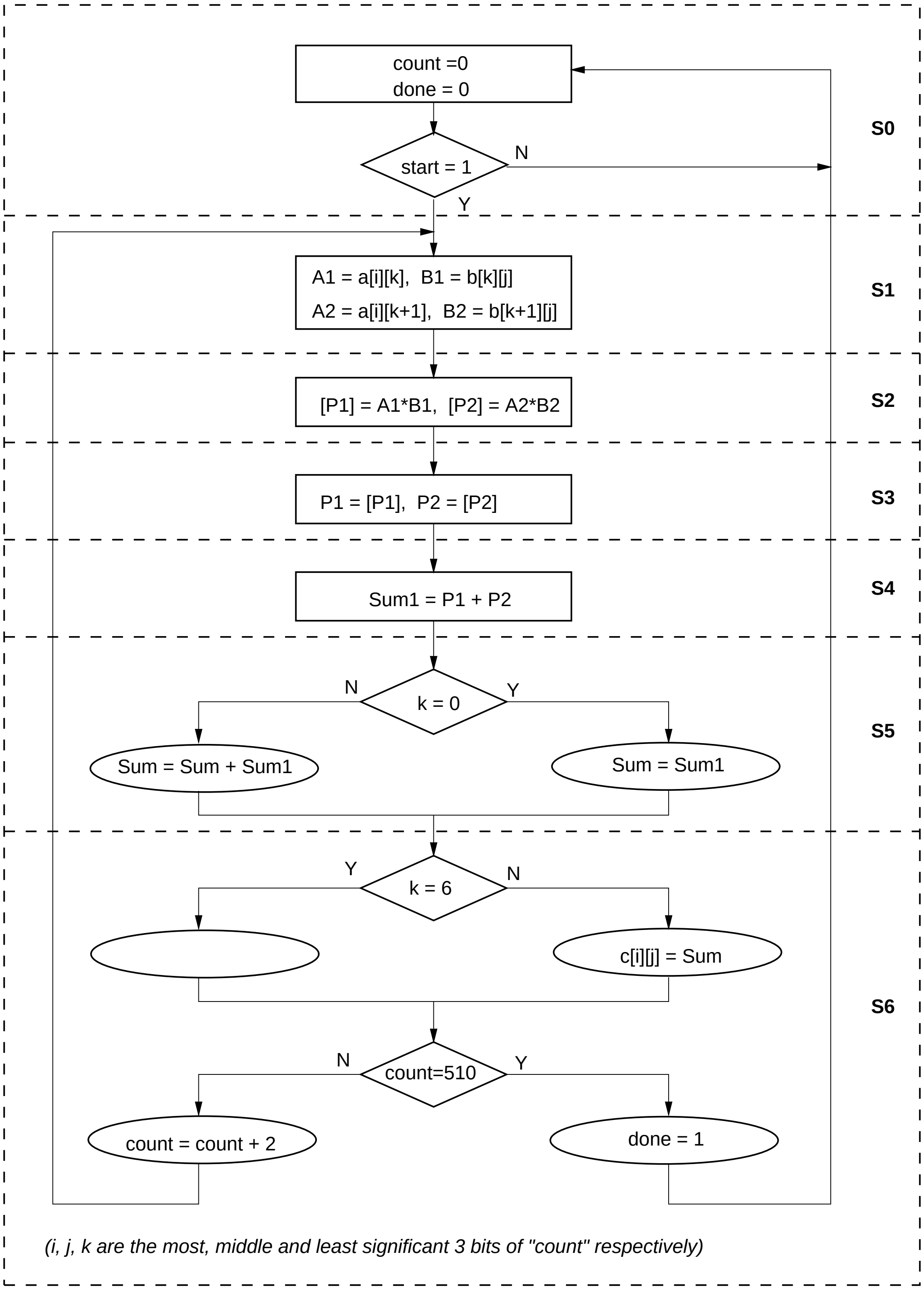

Figure 7: ASM chart for unrolled, multicycled matrix multiplication

we present a complete design using a *"distributed controller"* model.

4.1 Process Pipelining

The DCT algorithm consists of two matrix multiplications serialized in time. In the sequential algorithm computation of the DCT on a new image block starts only after both matrix multiplications have been completed. Only one of the processes is active at a time. However, both processes can be made to operate concurrently on different sets of data. The first process performs the first matrix multiplication and generates the `TempBlock` matrix. The second process uses this `TempBlock` and performs the second matrix multiplication. Concurrently with this, the first process starts computing on a new image block. The two processes are called the *pipeline stages* as shown in Figure 8.

```
for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A := In(i, k); B := Cos(j, k);
      P := A * B;
      if (k=0) then
        Sum := P;
      else
        Sum := Sum + P;
      end if;
      if (k=7) then
        Temp(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;
```

Stage 1

Stage 2

```
for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A := Temp(k, j); B := Cos(i, k);
      P := A * B;
      if (k=0) then
        Sum := P;
      else
        Sum := Sum + P;
      end if;
      if (k=7) then
        Out(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;
```

Figure 8: The two stages in process pipelining

Process pipelining requires that both stages (the two processes) are active together. Thus, same hardware resources may not be used for both matrix multiplications. The design consequently requires twice the number of multipliers, adders, registers and selector logic. In addition, two memories are required for storing the `TempBlock` matrix as shown in Figure 9. In one iteration of DCT computation, stage 1 writes into RAM 1 and stage 2 reads from RAM 2. In the next iteration, the memories are switched and stage 1 writes into RAM 2 while stage 2 reads from RAM 1.

The throughput of the process pipelined design is half that of the the non-pipelined sequential design, i.e., 18841.6 ns. The cost of the design increases as suggested by Figure 9. The cost is 50K transistors since the entire sequential design is duplicated.

4.2 Loop Pipelining

Process pipelining was able to improve the performance but incurred a large hardware cost since it requires double the number of functional units and two memories. An alternative is to pipeline the loop itself. In the sequential design, an iteration of the loop begins after the previous finishes. Only state in the loop is active at a time and the others are idle. The loop can be pipelined by starting an iteration of the loop every clock. The states of the loop are now called **stages**. Registers latch the intermediate results between the stages. Such a loop pipelined design is shown in Figure 10.

Loop pipelining incurs very little additional cost. Since the two matrix multiplications are serialized in time, same hardware resources may be used for both loops and thus, hardware does not have to be doubled as was the case in process pipelining. All stages are not active from the start. In the first clock cycle, only the first stage is active. In the next, the first two stages are active; the first stage works on data set 1 while the second stage works on previous data set 0. This continues till all the four stages become active. This is *pipeline filling* and the pipeline is *flushed* similarly as shown in Figure 11.

It is difficult to describe a pipelining using the original ASM chart. The Extended ASM chart shows only the state in which all pipeline stages are active. The filling and flushing states are not shown in the ASM chart but they shall be executed. An Extended ASM chart for the loop pipelined design is shown in Fig-

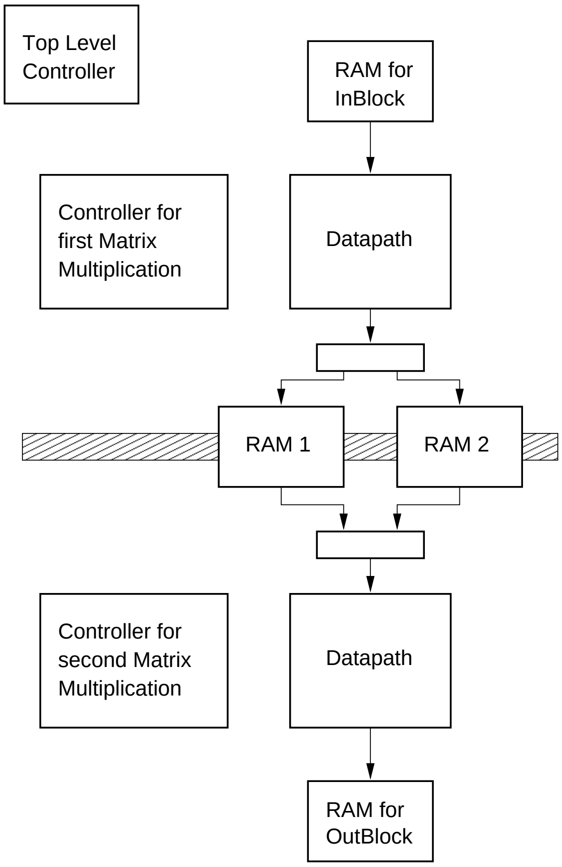

Figure 9: Overview of process pipelined datapath

```
for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A := In(i, k); B := Cos(j, k);        Stage 1
      P := A * B;                           Stage 2
      if (k=0) then
        Sum := P;
      else
        Sum := Sum + P;
      end if;                               Stage 3
      if (k=7) then                         Stage 4
        Temp(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;

for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A := Temp(k, j); B := Cos(i, k);      Stage 1
      P := A * B;                           Stage 2
      if (k=0) then
        Sum := P;
      else
        Sum := Sum + P;
      end if;                               Stage 3
      if (k=7) then                         Stage 4
        Out(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;
```

Figure 10: The stages in loop pipelining

Clock Cycles →	1	2	3	4	5		511	512	513	514	515	516						...				1029	1030
Stage1 (A=In[i][k]; B=Cos[j][k])	0	1	2	3	4	...	510	511															
Stage2 (P=A×B)		0	1	2	3	...	509	510	511														
Stage3 (Sum=Sum+P)			0	1	2	...		509	510	511													
Stage4 (Temp[i][j]=Sum)				0	1	...	..		509	510	511												
Stage1 (A=Cos[i][k]; B=Temp[k][j])												0	1	2	3	4	...	510	511				
Stage2 (P=A×B)													0	1	2	3	...	509	510	511			
Stage3 (Sum=Sum+P)														0	1	2	...		509	510	511		
Stage4 (Out[i][j]=Sum)															0	1	...	..		509	510	511	

← Matrix Multiplication 1 → ← Matrix Multiplication 2 →

Figure 11: Timing diagram for loop pipelining

ure 12. Thus, the assumption is that in the first clock cycle, Stage 1 will be executed, in the next clock cycle, Stage 1 and 2 will be executed and so forth. In order words, filling and flushing are implicit in the Extended ASM chart.

The pipeline can also be described using a State Action Table (SAT) [8]. A SAT can be used to describe the filling and flushing states also, as shown in Figure 13. The SAT can be used for implementing the controller of the loop pipelined design as shown in Figure 14. Each stage works on a different set of data and hence, four counters are required. Comparison with Figure 2 shows that there is very little extra hardware cost. The controller has more number of states and three extra counters are required.

The timing diagram lets us compute the performance of the pipelined design as follows. Each loop requires 3 cycles for filling the pipelining and 3 for flushing the pipelining. All states are active for 509 cycles. Thus, each loop takes $3 + 509 + 3 = 515$ clock cycles. The clock period remains the same at 9.2 ns.

$$\begin{aligned}
\#\text{ states} &= 1 \\
\text{clock period} &= 9.2 \text{ ns} \\
\#\text{ iterations} &= 515 + 515 \\
\text{Performance} &= \#\text{states} \times \text{clock} \times \#\text{iterations} \\
&= 1 \times 9.2 \times 1030 = 9476 \text{ ns}
\end{aligned}$$

The loop pipelined design incurs an extra cost because of the registers for the counter as shown in Figure 14. Each register costs 272 transistors from Table 1. Thus, the cost of the loop pipelined design is 26K transistors.

4.3 Functional Unit Pipelining

Some functional units may be much slower than the other components in a design. In such cases, it might be possible to improve the performance by pipelining the functional unit. In the DCT design, the multiplier is the slowest functional unit and can be pipelined. The pipelined multiplier is divided into 4 stages and a new data set can be instantiated every clock cycle. The latency of the multiplier essentially remains the same (it may increase because of the partition and the intermediate latches) but the throughput increases. A faster clock can be used because each stage is shorter than the complete multiplier. The stages with a pipelined multiplier are shown in Figure 15.

```
for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A:=In(i, k); B:=Cos(j, k);        Stage 1
                                        Stage 2
                                        Stage 3
       P := A * B;                      Stage 4
                                        Stage 5
      if (k=0) then                     Stage 6
        Sum := P;
      else
        Sum := Sum + P;
      end if;
      if (k=7) then                     Stage 7
        Temp(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;
```

```
for i in 0 to 7 loop
  for j in 0 to 7 loop
    for k in 0 to 7 loop
      A:=Temp(k, j); B:=Cos(i, k);      Stage 1
                                        Stage 2
                                        Stage 3
       P := A * B;                      Stage 4
                                        Stage 5
      if (k=0) then                     Stage 6
        Sum := P;
      else
        Sum := Sum + P;
      end if;
      if (k=7) then                     Stage 7
        Temp(i, j) := Sum;
      end if;
    end loop;
  end loop;
end loop;
```

Figure 15: The stages in functional unit pipelining

It is important to note that functional unit pipelining requires availability of pipelined functional units in the RTL-library. The other optimizations were done using existing RTL components. Functional unit pipelining also requires availability of data every *data introduction interval* of the unit. Thus, there must be enough computations that can be done using the pipelined functional unit. If this is not possible, then the design may be loop pipelined. In our case, there

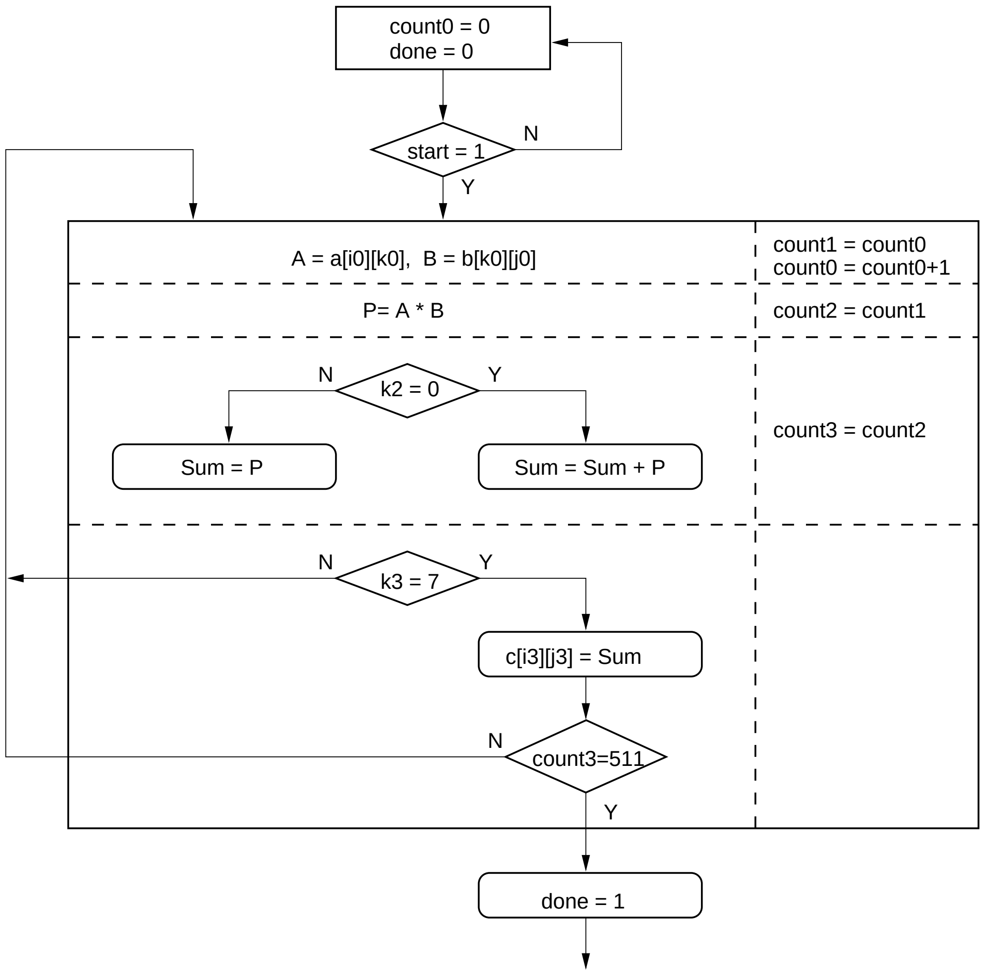

Figure 12: ASM chart for process pipelining

PRESENT STATE	NEXT STATE CONDITION, STATE			CONTROL AND DATAPATH ACTIONS CONTROL, ACTIONS	
S0 (000)	Start=0, Start=1,	S0 S1	000 001		count0=0 done=0
S1 (001)		S2	010		A=a[i0][k0] B=b[k0][j0] count1=count0 count0=count0+1
S2 (010)		S3	011		P=A*B count2=count1 A=a[i0][k0] B=b[k0][j0] count1=count0 count0=count0+1
S3 (011)		S4	100	k2=0, k2≠0,	Sum=P Sum=Sum+P count3=count2 P=A*B count2=count1 A=a[i0][k0] B=b[k0][j0] count1=count0 count0=count0+1
S4 (100)	count0≠511, count0=511,	S4 S5	100 101	k3=7, k2=0, k2≠0, count0≠511,	c[i3][j3]=Sum Sum=P Sum=Sum+P count3=count2 P=A*B count2=count1 A=a[i0][k0] B=b[k0][j0] count1=count0 count0=count0+1
S5 (101)		S6	110	k3=7, k2=0, k2≐0,	c[i3][j3]=Sum Sum=P Sum=Sum+P count3=count2 P=A*B count2=count1
S6 (110)		S7	111	k3=7, k2=0, k2≐0,	c[i3][j3]=Sum Sum=P Sum=Sum+P count3=count2
S7 (111)		S0	000	k3=7,	c[i3][j3]=Sum done=1

D0 = (Start AND S0) OR S2 OR ((count0=511) AND S4) OR S7
D1 = S1 OR S2 OR S5 OR S6
D2 = S3 OR S4 OR S6 OR S7

Figure 13: State Action Table for loop pipelining

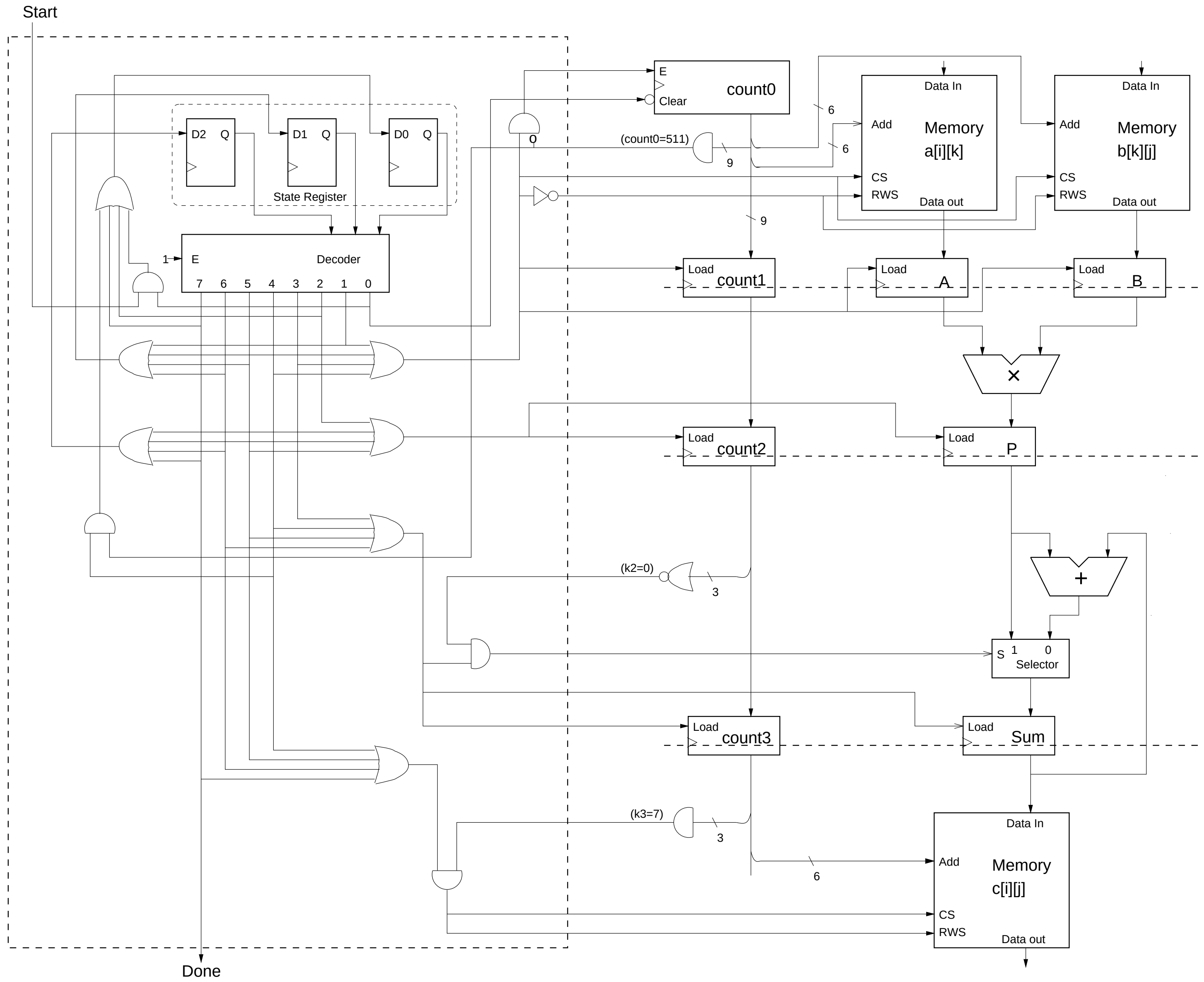

Figure 14: Design for pipelined matrix multiplication

is only multiplication in the loop and hence, we need to pipeline the loop also.

The pipe for each matrix multiplication now consists of 7 stages as shown in Figure 15. The delay of a multiplier with 4 stages is 3.5 ns from Table 1. Thus, the clock period is 3.5+0.4=3.9 ns. Each loop requires 6 cycles for filling the pipelining and 6 for flushing the pipelining. All states are active for 506 cycles. Thus, each loop takes $6 + 506 + 6 = 518$ clock cycles.

# states	=	1
clock period	=	3.9 ns
# iterations	=	518 + 518
Performance	=	#states×clock×#iterations
	=	1×3.9×1036 = 4040.4 ns

The pipelined multiplier is more costly than the non-pipelined multiplier. It uses 15036 transistors as opposed to 11220 transistors used by the non-pipelined multiplier. Thus, the net cost of the design is 30K transistors.

4.4 Long pipe with both matrix multiplications

In all the previous examples, the second matrix multiplication was started after the first matrix multiplication was completed. Even though both the matrix multiplications were performed concurrently in the process pipelined design, yet they operated on different image blocks. The first process generated the entire `TempBlock` matrix and the second process then performed the second matrix multiplication on this matrix. However, both multiplications can be started together, if the two matrix multiplications are reversed.

In the current algorithm, the second process reads the `TempBlock` matrix in a *column-wise* manner while the first process generates the matrix in a *row-wise* manner. Thus, we can change the order of matrix multiplications as follows.

$$TempBlock = CosBlock \times InBlock \quad \text{(MM1)}$$
$$OutBlock = TempBlock \times CosBlock^T \quad \text{(MM2)}$$

It takes 64 iterations of the first multiplication loop to generate a row (8 values) of the `TempBlock` matrix and it takes 64 iterations of the second matrix multiplication to consume a row of the `TempBlock` matrix. Thus, the second process can be started after the first process completes 64 iterations. In this way, both the processes will be active concurrently. Each process can, in addition, be loop pipelined since it does not incur additional costs. With these changes, the DCT design consists of a long pipe whose timing diagram is shown in Figure 16.

The timing diagram lets us compute the performance of the pipelined design. The first loop requires 3 cycles for filling the pipelining. The second matrix multiplication is started after the first 64 iterations are complete. It then takes 512+3 more clock cycles to finish the DCT. Thus, the total number of clock cycles required is $3 + 64 + 512 + 3 = 582$.

# states	=	1
clock period	=	9.2 ns
# iterations	=	582
Latency	=	#states×clock×#iterations
	=	1×9.2×582 = 5354.4 ns

The cost of the design increases since both the loops are active at the same time. Thus, the datapath is doubled as compared to just loop pipelining (Section 4.2). The total cost is then 52K transistors.

4.5 Pipelined Design with *Distributed* Controller

A typical hardware design consists of a datapath and a controller as shown in the design for a sequential design in Figure 2. However, the number of states in the state transition function of a pipelined design is large because of the filling-up and flushing of the pipeline stages. Thus, the FSM inside the controller gets unwieldy and large as in the design for the loop pipelined design, shown in Figure 14. We next present a design that uses a *distributed* model for the controller which results in a much simpler design.

The controller design complexity can be reduced by having a separate controller for each pipeline stage. Since each controller controls just one pipeline stage, it is only 1-bit wide and can be implemented using a *D flip-flop* or an *SR latch*. A stage is active and computes whenever the corresponding flip-flop is *set*. The 1-bit single state controllers are themselves connected as shown in the schematic in Figure 17.

Initially, all the flip flops are *reset*. Computation starts by loading '1' into the flip flop for `Stage 0`.

Clock Cycles →	1	2	3	4	..	..	65	66	67	68	69		..	..		511	512	513	514	515	...	...	578	579	580	581	582
Stage1 (A=Cos[i][k]; B=In[k][j])	0	1	2		..		64	65	66			..	..			510	511										
Stage2 (P=A×B)		0	1	2		..		64	65				..				510	511									
Stage3 (Sum=Sum+P)			0	1		..		63	64	65			..				509	510	511								
Stage4 (Temp[i][j]=Sum)				0		..		62	63	64	65			..				509	510	511							
Stage1 (A=Temp[i][k]; B=Cos[j][k])										0	1	2		..	..		444	445		...			510	511			
Stage2 (P=A×B)											0	1				...		444	445		...			510	511		
Stage3 (Sum=Sum+P)												0	1				...		444			...		509	510	511	
Stage4 (Out[i][j]=Sum)													0	1				...		444			...		509	510	511

← Matrix Multiplication 1 →

← Matrix Multiplication 2 →

Figure 16: Timing diagram when both processes are started

Then every clock, this '1' *token* is passed to the next flip flop. Every clock one more pipeline stage becomes active. This is the filling up of pipeline. When the computation is over, the SR latch is *reset*. This '0' *token* is then passed to the next stages and they stop computing progressively. This is the flushing of the pipeline.

A distributed control design requires a flip flop for each pipeline stage. Thus, a minimum of n flip flops are required if there are n stages in the pipeline. A single controller will need $\lceil log_2 k \rceil$ flip flops where k is the number of states. In a pipe with n stages, there would be $n-1$ states for pipeline filling, 1 for the full pipe and $n-1$ for pipeline flushing. Therefore, the total number of states, $k = n - 1 + 1 + n - 1 = 2n - 1$. The distributed controller design, thus, uses less number of flip flops, has minimal *next state logic* as shown in Figure 17 and is simpler to design. In such a design, each pipeline stage can be modeled as a Finite State Machine with a Datapath (FSMD) [8].

5 Memory Optimization

In the previous sections, we did explorations using the algorithm presented in Section 2.1. This algorithm performs a matrix multiplication on `InBlock` and `CosBlock` and generates the 8×8 `TempBlock` matrix. This `TempBlock` matrix is used for the second matrix multiplication. However, in most signal processing applications like video and speech processing, memory occupies more than 50% of the chip area [9]. In these type of applications, the chip area can be reduced more effectively with memory optimizations than with just datapath optimizations. We next present an algorithm that does not store the entire matrix and uses only 1 word compared to the 64 words required by the earlier algorithm.

The entire `TempBlock` need not be stored in a memory if each value of the matrix is consumed as soon as it is produced. Thus, the two matrix multiplications have to be interleaved. Each `TempBlock` element is used for 8 elements of the `OutBlock` matrix. Hence, each `TempBlock` value is multiplied with the corresponding `CosBlock` values and added to the partial sums in the `OutBlock` matrix. Thus, at any time the `OutBlock` only has partial sums. Every time a new `TempBlock` value is computed, it is used to update the corresponding column as shown in Figure 18(b). The first matrix multiplication loop produces the element at (i, j) of the `TempBlock` matrix. This value is multiplied with the i^{th} column of `CosBlock` matrix, i.e. elements at $(0, i), (1, i), \ldots, (7, i)$. This generates the partial sums for the j^{th} column of `OutBlock` matrix. Note that every time all `TempBlock` elements in j^{th} column update the j^{th} column of `OutBlock` matrix. The complete VHDL behavioral model is given in Appendix D.

This algorithm requires only 1 word for storing a `TempBlock` element since it is consumed as soon as it is produced. However, the number of memory accesses increases since the `OutBlock` stores partial sums and these must be read and then written back into. However, this does not decrease performance since the accesses are in different clock cycles and hence, the clock period does not have to be increased. The performance can then be calculated as follows.

\# states = (4×8)+(4×8)=64
clock period = 9.2 ns

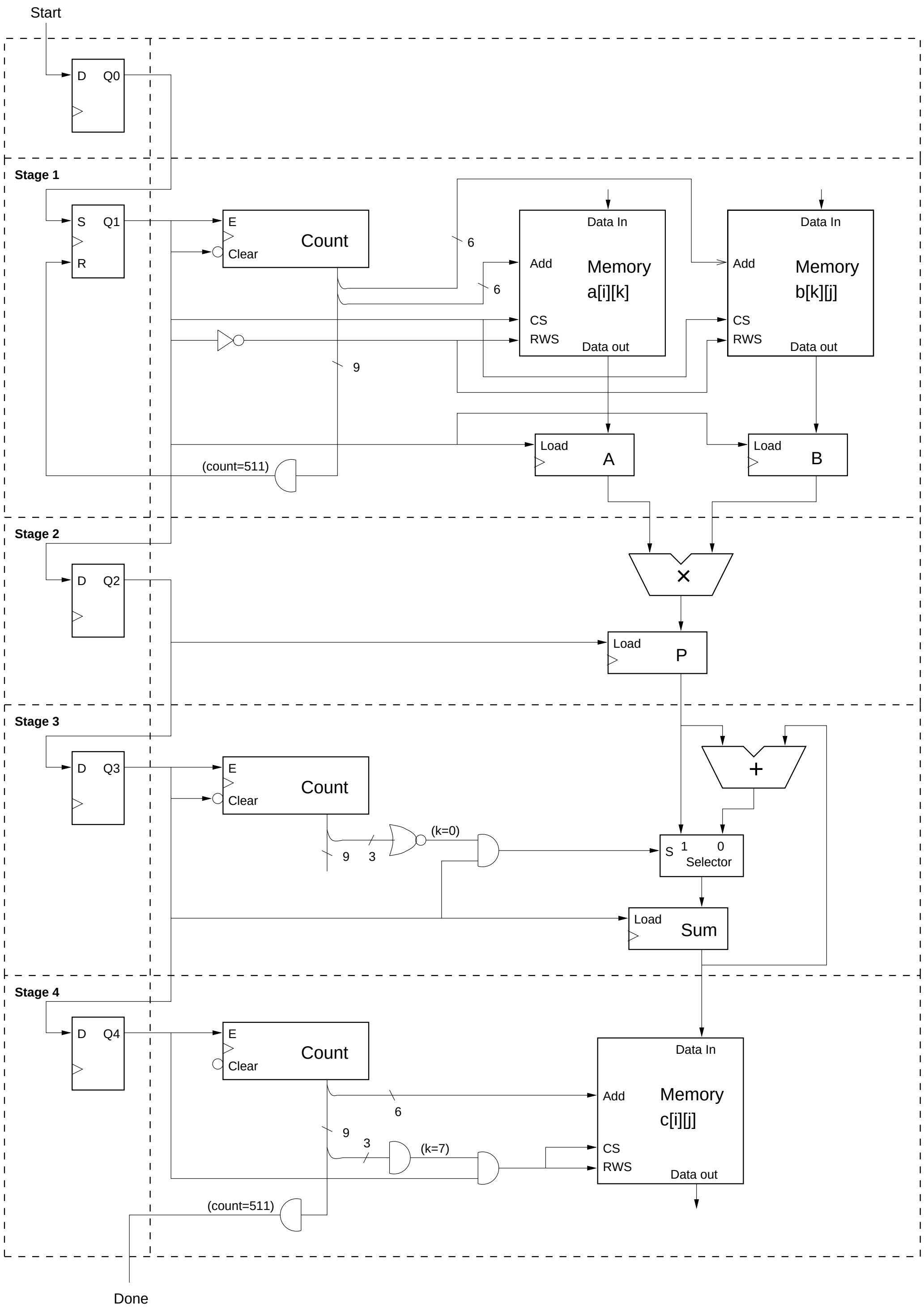

Figure 17: One FSMD for each pipelined stage

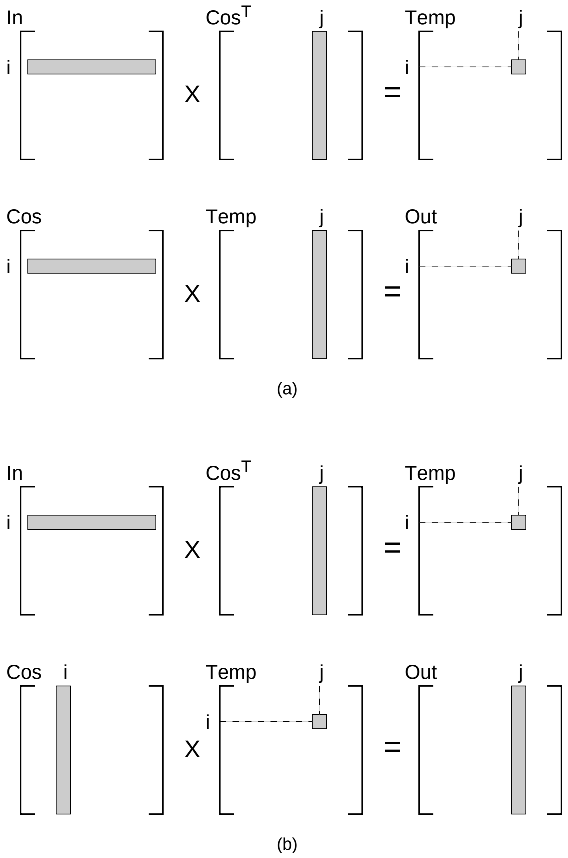

Figure 18: Memory Optimization: (a) Original algorithm (b) Memory optimized algorithm

$$\begin{aligned}\#\text{ iterations} &= 64\\ \text{Performance} &= \#\text{states}\times\text{clock}\times\#\text{iterations}\\ &= 64\times 9.2\times 64 = 37683.2\text{ ns}\end{aligned}$$

Thus, the performance is the same as for the most sequential algorithm presented in Section 3.2 but we have been able to decrease the memory requirements from 64 words to a single word. Thus, the cost of this design is 19K transistors.

5.1 Loop and FU Pipelining

The performance of the memory optimized algorithm can be improved using the techniques used for improving the performance of the sequential algorithm (as discussed in Sections 3.3, 3.4 and 3.5). In addition, the design can be pipelined just like the sequential design (as discussed in Sections 4.2 and 4.3). However, *Loop unrolling* incurs extra hardware cost. Pipelining, on the other hand, improves the performance with little overheads. Since the techniques are similar, we just discuss loop and functional unit pipelining for the memory optimized algorithm.

The loop in the memory optimized algorithm can be pipelined to improve performance as discussed in Section 4.2. In addition, the same design can use a pipelined multiplier which reduces the clock period and, hence, improves the performance. The DCT core is coded in lines 60–98 of Appendix D. There are 8×8=64 iterations of the inner loops on variable `k`. We pipeline these two loops into eight stages as shown in Figure 19. The multiplier has two stages.

```
for i in 0 to 7 loop
   for j in 0 to 7 loop
      for k in 0 to 7 loop
         A:=In(i,j); B:=Cos(j,k);        Stage1
                                         Stage2
            P:=A×B;
                                         Stage3
         if (k=0) then                   Stage4
            sum:=P;
         elsif (k=7) then
            temp:=sum + P;
         else
            sum:=sum + P;
         end if;
      end loop;

      for k in 0 to 7 loop
         C:=Out(k,j); D:=Cos(k,i);       Stage5
                                         Stage6
            prod:=d×temp;
                                         Stage7
         if (i=0) then                   Stage8
            Out(k,j):= prod;
         else
            Out(k,j):= prod + C;
         end if;
      end loop;
   end loop;
end loop;
```

Figure 19: The stages in loop and functional unit pipelined design

The filling up of the eight pipelining stages takes more clock cycles than suggested by the loop pipelining example in Section 4.2. The second loop can be started only after eight iterations of the first loop has been completed because the second loop requires the `temp` value calculated by the first loop (in line 75 of Appendix D). It takes 8 multiplications and additions to generate a `temp` value. Thus, the stages of the sec-

Clock Cycles →	1	2	3	4	5		9	10	11	12	13	14	15			512	513	514	515	516		523	524	525	526
Stage1 (A:=In(i,j); B:=Cos(j,k))	0	1	2	..		...	8	9	10		..				....	511					....				
Stage2 (tempP:=A×B)		0	1			...	7	8	9	10		..			....	510	511				....				
Stage3 (P := tempP)			0	1		...		7	8	9	10		..		....	509	510	511			....				
Stage4 (sum := sum + P)				0	1	...			7	8	9	10		..	....	508	509	510	511		....				
Stage5 (C:=Out(k,j); D:=Cos(k,i))										0	1	2			....	500	501	502	503	504	....	511			
Stage6 (tempProd := D×temp)											0	1			....	499	500	501	502	503	....	510	511		
Stage7 (prod := tempProd)												0	1		....	498	499	500	501	502	....	509	510	511	
Stage8 (Out(k,j) := prod+C)													0	1	....	497	498	499	500	501	....	508	509	510	511

Figure 20: Timing diagram for pipelined memory optimized algorithm

ond loop are delayed by 8 iterations of the first loop. It will take 11 clock cycles to perform eight iterations of the first loop since 3 clock cycles are required for filling the pipeline. The pipeline flushing is similar. The timing diagram for the loop and multiplier pipelined design is shown in Figure 20.

The timing diagram lets us compute the performance of the pipelined design as follows. It takes 11 clock cycles to generate the first `temp` value. The stages for second loop take another 3 clock cycles for filling. Finally, it takes 512 clock cycles for completing all the iterations of the second loop. Hence, the total number of clock cycles is $11 + 3 + 512 = 526$. The clock period is $5.4 + 0.4 = 5.8$ ns since the delay of a two-stage pipelined multiplier is 5.4 ns from Table 1.

$$
\begin{aligned}
\#\ \text{states} &= 1\\
\text{clock period} &= 5.8\ \text{ns}\\
\#\ \text{iterations} &= 526\\
\text{Performance} &= \#\text{states}\times\text{clock}\times\#\text{iterations}\\
&= 1\times 5.8\times 526 = 3050.8\ \text{ns}
\end{aligned}
$$

The pipelined design uses extra registers for storing the count value. Thus, the cost increases to 21K transistors. The complete behavioral model for the pipelined memory optimized algorithm is given in Appendix D.

6 Comparison of optimization techniques

We looked at some commonly used optimization techniques like *loop unrolling*, *chaining* and *multicycling* in Section 3. We explored pipelining options in Section 4. Finally, we presented a memory optimized algorithm in Section 5. We now compare the various optimization and pipelining techniques. Table 2 gives a summary of the performance of the non-pipelined, pipelined and memory optimized designs.

Table 2: Comparison of optimization techniques

design \ *parameter*	*Latency (ns)*	*Cost (trans)*
Sequential	37684 ns	25K
2-Unrolled	23552 ns	44K
Unrolled and Chained	18842 ns	41K
Unrolled and Multicycled	14131 ns	42K
Process Pipelined	18842 ns	50K
Loop Pipelined	9476 ns	26K
Loop and FU Pipelined	4040 ns	30K
Both loops together	5354 ns	52K
Memory Optimized	37683 ns	19K
Pipelined Mem Optimized	3051 ns	21K

Table 2 shows the performance and cost of different implementations for the DCT. There is a wide range of latency times and costs of the different designs. A pipelined design must be used to meet the timing constraint of 4100 ns given in Section 1. Furthermore, a pipelined multiplier has to be used along with loop pipelining since loop pipelining alone cannot meet the stringent latency requirements. There are two designs that meet the timing constraints as shown in Table 2. The memory optimized is the most efficient in terms of performance and cost.

7 Conclusion

In this report, we presented the formal definition of the Discrete Cosine Transform and a sequential implementation for it using RTL components. We described some commonly used optimization techniques like *loop unrolling*, *chaining* and *multicycling* to reduce the latency of the sequential design. We also explored various levels of pipelining like *process pipelining*, *loop pipelining* and *functional unit pipelining* to further improve the performance without incurring extra hardware cost. We also described a pipelined implementation using a *"distributed controller"* which reduced the complexity of the control-path.

We presented a memory optimized algorithm to further reduce the hardware costs. We then described a pipelined implementation of the memory optimized algorithm. A comparison of the different techniques showed that pipelining is required for meeting the timing constraints of the DCT component. The large spectrum of performance to cost tradeoffs is a good starting point for further optimizations during high level synthesis.

References

[1] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic Publishers, 1997.

[2] Ching-Te Chiu and K.J. Ray Liu. "Real-Time Parallel and Fully Pipelined Two-Dimensional DCT Lattice Structures with Application to HDTV Systems", *IEEE Transactions on CAS for Video Technology*, Vol 2, No 1, March 1992.

[3] Didler Le Gall. "MPEG: A Video Compression Standard for Multimedia Applications", *Communications of the ACM 34*, 1994.

[4] K. R. Rao and P. Yip. *Discrete Cosine Transform: Algorithms, Advantages, Applications*, Academic Press, Inc. 1990.

[5] James F. Blinn, "What's the Deal with the DCT?" *IEEE Computer Graphics and Applications*, July 1993.

[6] Wenwei Pan, Peter Grun and Daniel Gajski. "Behavioral Exploration with RTL Library", *Technical Report, UCI-ICS-#96-34*, July 1996.

[7] LCB 500K, *Preliminary Design Manual*, LSI Logic Inc., June 1995.

[8] Daniel D. Gajski. *Principles of Digital Design*, Prentice Hall, 1997.

[9] Francky Catthoor, Werner Geurts, Hugo De Man. "Loop Transformation Methodology for Fixed-rate Video, Image and Telecom Processing Applications", Proceedings of the *International Conference on Application Specific Array Processors*, San Francisco, CA, 1994.

[10] Zainalabedin Navabi. *VHDL: Analysis and modeling of Digital Systems*, McGraw-Hill, 1993.

A Behavioral model of Sequential Design

In this appendix we give the detailed behavioral model of the most sequential design for the DCT component. VHDL [10] was used for modeling and simulation.

```
---------------------------------------------------
-- DCT component
-- compute the transform for a 8x8 image block
-- sequential algorithm without any optimizations
--
-- Gaurav Aggarwal; December 20, 1997.
---------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity dct is
    port ( clk   : in  std_logic;
           start : in  std_logic;
           din   : in  integer;
           done  : out std_logic;
           dout  : out integer);
end dct;


architecture behavioral of dct is
begin
    process
        type memory is array (0 to 7, 0 to 7) of integer;

        variable InBlock, TempBlock, OutBlock : memory;
        variable A, B, P, Sum : integer;
        variable CosBlock : memory :=
          ((88,  122,  115,  103,  88,   69,   47,   24),
           (88,  103,   47,  -24, -88, -122, -115,  -69),
           (88,   69,  -47, -122, -88,   24,  115,  103),
           (88,   24, -115,  -69,  88,  103,  -47, -122),
           (88,  -24, -115,   69,  88, -103,  -47,  122),
           (88,  -69,  -47,  122, -88,  -24,  115, -103),
           (88, -103,   47,   24, -88,  122, -115,   69),
           (88, -122,  115, -103,  88,  -69,   47,  -24));
    begin

        -----------------------------
        -- wait for the start signal
        -----------------------------
        wait until start = '1';
        done <= '0';

        ----------------------------------
        -- read input 8x8 block of pixels
        ----------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
```

```
                wait until clk = '1';
                InBlock(i, j) := din;
            end loop;
        end loop;

        ------------------------------------
        -- TempBlock = InBlock * CosBlock^T
        ------------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    A := InBlock(i, k);
                    B := CosBlock(j, k);
                    wait until clk='1';

                    P := A * B;
                    wait until clk='1';

                    if (k = 0) then
                        Sum := P;
                    else
                        Sum := Sum + P;
                    end if;
                    wait until clk='1';

                    if (k = 7) then
                        TempBlock(i, j) := Sum;
                    end if;
                    wait until clk='1';
                end loop;
            end loop;
        end loop;

        -----------------------------------
        -- OutBlock = CosBlock * TempBlock
        -----------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                for k in 0 to 7 loop
                    A := TempBlock(k, j);
                    B := CosBlock(i, k);
                    wait until clk='1';

                    P := A * B;
                    wait until clk='1';

                    if (k = 0) then
                        Sum := P;
                    else
                        Sum := Sum + P;
                    end if;
                    wait until clk='1';

                    if (k = 7) then
                        OutBlock(i, j) := Sum;
```

```
                    end if;
                    wait until clk='1';
                end loop;
            end loop;
        end loop;


        ------------------------
        -- give the done signal
        ------------------------
        wait until clk = '1';
        Done <= '1';

        ------------------------------
        -- output the computed matrix
        ------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                Dout <= OutBlock(i, j);
            end loop;
        end loop;
        Done <= '0';
    end process;
end behavioral;
```

B Structural model of Sequential Design

The schematics for the structural model have been captured using the *Synopsys Graphical Environment* (sge) tools. The top-level model of DCT comprises of a datapath and a controller as shown in Figure 21.

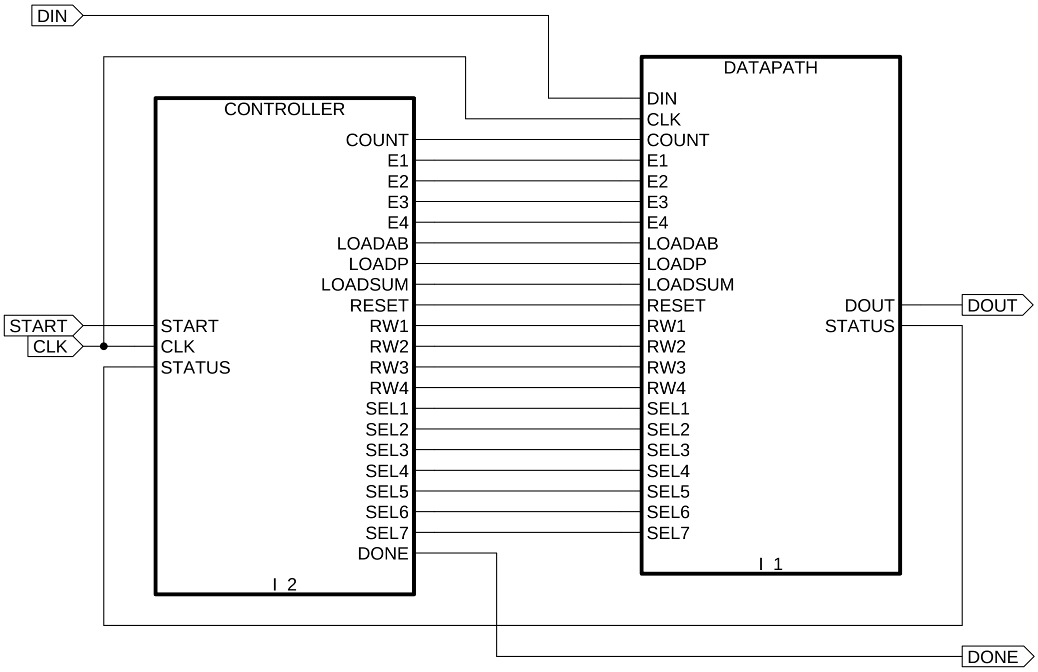

Figure 21: Schematic of Sequential DCT

The controller is a Mealy type finite state machine. The schematic is shown in Figure 22.

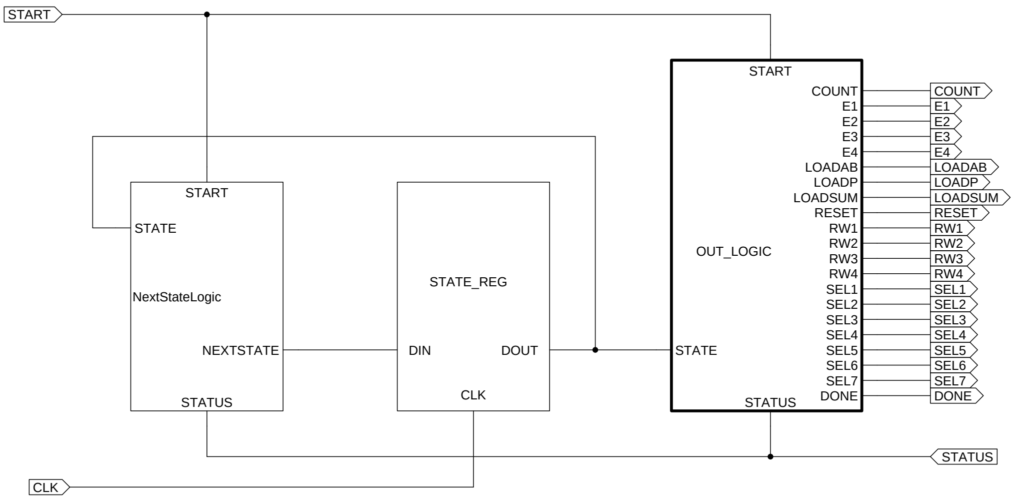

Figure 22: Schematic of Controller for Sequential DCT

The datapath comprises of register-level components. The schematic is shown in Figure 23.

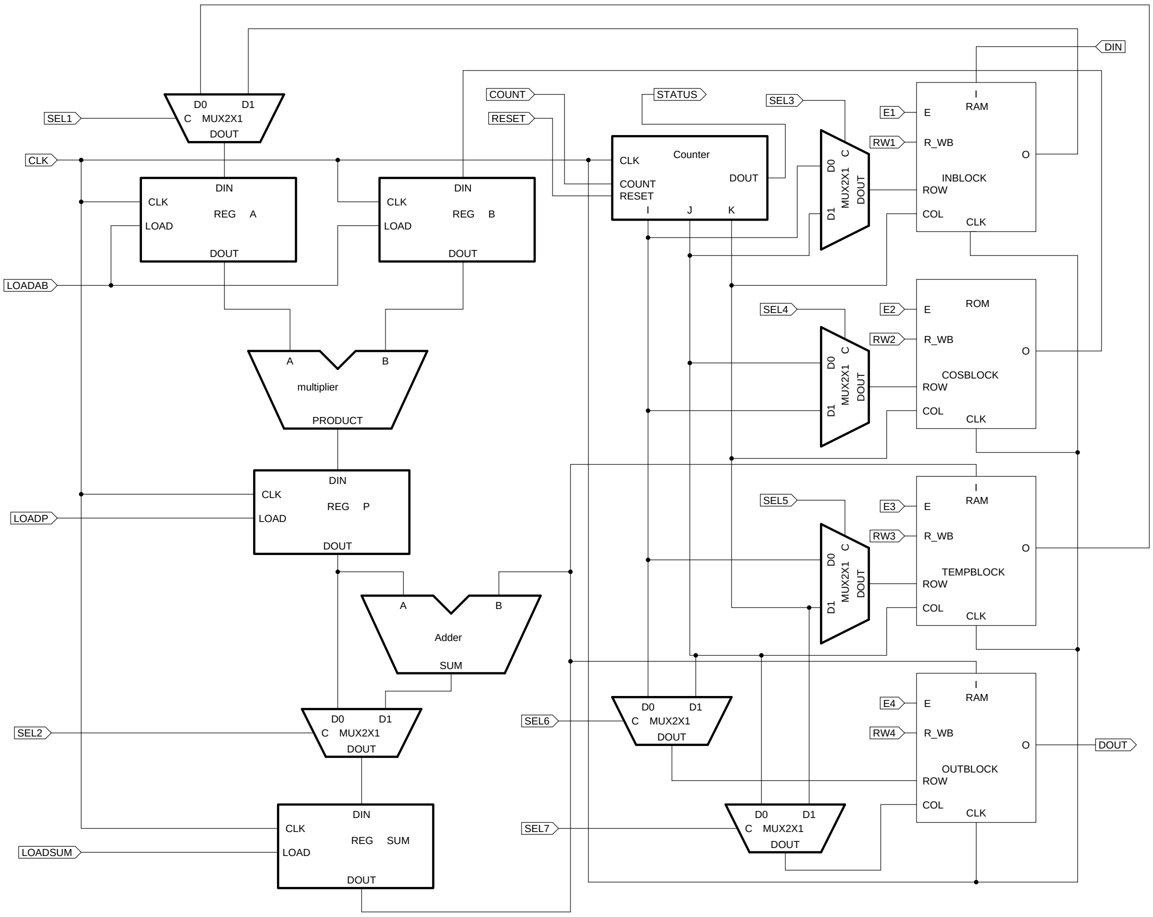

Figure 23: Schematic of Datapath for Sequential DCT

B.1 VHDL code for datapath

The datapath for the sequential design consists of RT-level components. We next list the netlist for the datapath which is shown in Figure 23.

```
-- VHDL Model Created from SGE Schematic datapath.sch -- Jan 13 10:39:31 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;
   use work.components.all;

entity DATAPATH is
```

```
      Port (     CLK : In    std_logic;
               COUNT : In    std_logic;
                 DIN : In    integer;
                  E1 : In    std_logic;
                  E2 : In    std_logic;
                  E3 : In    std_logic;
                  E4 : In    std_logic;
              LOADAB : In    std_logic;
               LOADP : In    std_logic;
             LOADSUM : In    std_logic;
               RESET : In    std_logic;
                 RW1 : In    std_logic;
                 RW2 : In    std_logic;
                 RW3 : In    std_logic;
                 RW4 : In    std_logic;
                SEL1 : In    std_logic;
                SEL2 : In    std_logic;
                SEL3 : In    std_logic;
                SEL4 : In    std_logic;
                SEL5 : In    std_logic;
                SEL6 : In    std_logic;
                SEL7 : In    std_logic;
                DOUT : Out   integer;
              STATUS : Out   integer );
end DATAPATH;

architecture SCHEMATIC of DATAPATH is

   signal     N_22 : integer;
   signal     N_20 : integer;
   signal     N_21 : integer;
   signal      N_1 : integer;
   signal      N_2 : integer;
   signal      N_3 : integer;
   signal      N_4 : integer;
   signal      N_5 : integer;
   signal      N_7 : integer;
   signal      N_8 : integer;
   signal      N_9 : integer;
   signal     N_10 : integer;
   signal     N_11 : integer;
   signal     N_12 : integer;
   signal     N_13 : integer;
   signal     N_14 : integer;
   signal     N_16 : integer;
   signal     N_18 : integer;
   signal     N_19 : integer;

   component COUNTER
      Port (     CLK : In    std_logic;
```

```
              COUNT : In    std_logic;
              RESET : In    std_logic;
               DOUT : Out   integer;
                  I : Out   integer;
                  J : Out   integer;
                  K : Out   integer );
    end component;

    component ROM
       Port (    CLK : In    std_logic;
                 COL : In    integer;
                   E : In    std_logic;
                R_WB : In    std_logic;
                 ROW : In    integer;
                   O : Out   integer );
    end component;

    component RAM
       Port (    CLK : In    std_logic;
                 COL : In    integer;
                   E : In    std_logic;
                   I : In    integer;
                R_WB : In    std_logic;
                 ROW : In    integer;
                   O : Out   integer );
    end component;

    component MUX2X1
       Port (      C : In    std_logic;
                  D0 : In    integer;
                  D1 : In    integer;
                DOUT : Out   integer );
    end component;

    component MULTIPLIER
       Port (      A : In    integer;
                   B : In    integer;
             PRODUCT : Out   integer );
    end component;

    component REG
       Port (    CLK : In    std_logic;
                 DIN : In    integer;
                LOAD : In    std_logic;
                DOUT : Out   integer );
    end component;

    component ADDER
       Port (      A : In    integer;
                   B : In    integer;
```

```
                 SUM : Out   integer );
   end component;

begin

   I_19 : COUNTER
      Port Map ( CLK=>CLK, COUNT=>COUNT, RESET=>RESET, DOUT=>STATUS,
                 I=>N_20, J=>N_21, K=>N_18 );
   COSBLOCK : ROM
      Port Map ( CLK=>CLK, COL=>N_18, E=>E2, R_WB=>RW2, ROW=>N_4,
                 O=>N_14 );
   OUTBLOCK : RAM
      Port Map ( CLK=>CLK, COL=>N_19, E=>E4, I=>N_1, R_WB=>RW4,
                 ROW=>N_16, O=>DOUT );
   TEMPBLOCK : RAM
      Port Map ( CLK=>CLK, COL=>N_21, E=>E3, I=>N_1, R_WB=>RW3, ROW=>N_5,
                 O=>N_2 );
   INBLOCK : RAM
      Port Map ( CLK=>CLK, COL=>N_18, E=>E1, I=>DIN, R_WB=>RW1, ROW=>N_3,
                 O=>N_8 );
   I_18 : MUX2X1
      Port Map ( C=>SEL3, D0=>N_20, D1=>N_21, DOUT=>N_3 );
   I_17 : MUX2X1
      Port Map ( C=>SEL4, D0=>N_21, D1=>N_20, DOUT=>N_4 );
   I_16 : MUX2X1
      Port Map ( C=>SEL5, D0=>N_20, D1=>N_18, DOUT=>N_5 );
   I_14 : MUX2X1
      Port Map ( C=>SEL7, D0=>N_21, D1=>N_18, DOUT=>N_19 );
   I_15 : MUX2X1
      Port Map ( C=>SEL6, D0=>N_20, D1=>N_21, DOUT=>N_16 );
   I_1 : MUX2X1
      Port Map ( C=>SEL2, D0=>N_10, D1=>N_13, DOUT=>N_11 );
   I_2 : MUX2X1
      Port Map ( C=>SEL1, D0=>N_2, D1=>N_8, DOUT=>N_22 );
   I_3 : MULTIPLIER
      Port Map ( A=>N_7, B=>N_12, PRODUCT=>N_9 );
   SUM : REG
      Port Map ( CLK=>CLK, DIN=>N_11, LOAD=>LOADSUM, DOUT=>N_1 );
   P : REG
      Port Map ( CLK=>CLK, DIN=>N_9, LOAD=>LOADP, DOUT=>N_10 );
   B : REG
      Port Map ( CLK=>CLK, DIN=>N_14, LOAD=>LOADAB, DOUT=>N_12 );
   A : REG
      Port Map ( CLK=>CLK, DIN=>N_22, LOAD=>LOADAB, DOUT=>N_7 );
   I_8 : ADDER
      Port Map ( A=>N_10, B=>N_1, SUM=>N_13 );

end SCHEMATIC;

configuration CFG_DATAPATH_SCHEMATIC of DATAPATH is
```

```

   for SCHEMATIC
      for I_19: COUNTER
         use configuration WORK.CFG_COUNTER_BEHAVIORAL;
      end for;
      for COSBLOCK: ROM
         use configuration WORK.CFG_ROM_BEHAVIORAL;
      end for;
      for OUTBLOCK, TEMPBLOCK, INBLOCK: RAM
         use configuration WORK.CFG_RAM_BEHAVIORAL;
      end for;
      for I_18, I_17, I_16, I_14, I_15, I_1, I_2: MUX2X1
         use configuration WORK.CFG_MUX2X1_BEHAVIORAL;
      end for;
      for I_3: MULTIPLIER
         use configuration WORK.CFG_MULTIPLIER_BEHAVIORAL;
      end for;
      for SUM, P, B, A: REG
         use configuration WORK.CFG_REG_BEHAVIORAL;
      end for;
      for I_8: ADDER
         use configuration WORK.CFG_ADDER_BEHAVIORAL;
      end for;
   end for;

end CFG_DATAPATH_SCHEMATIC;
```

B.2 VHDL code for Next State Logic

The datapath in the structural model is a netlist of RT-level components from a library. The controller is a finite state machine that consists of a state register and the next state logic. In this section, we give the VHDL code listing for the *Next State Logic* component which is shown in Figure 22.

```
-- VHDL Model Created from SGE Symbol nsl.sym -- Jan 13 10:41:09 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;
   use work.components.all;

entity NSL is
    generic ( Delay   : Time := 5 ns);
    Port ( START     : In  std_logic;
           STATE     : In  STATE_VALUE;
           STATUS    : In  integer;
           NEXTSTATE : Out STATE_VALUE);
end NSL;

architecture BEHAVIORAL of NSL is
begin
```

```
    process (State, Start, Status)
        variable  Count : integer;
        variable  NewState : STATE_VALUE := S1;
    begin
        Count := STATUS;

        case State is
          when S1 =>
            if (Start = '1') then
                NewState := S2;
            else
                NewState := S1;
            end if;

          when S2 =>
            if (Count = 63) then
                NewState := S3;
            else
                NewState := S2;
            end if;

          when S3 =>
            NewState := S4;

          when S4 =>
            NewState := S5;

          when S5 =>
            NewState :=  S6;

          when S6 =>
            if (Count = 511) then
                NewState := S7;
            else
                NewState := S3;
            end if;

          when S7 =>
            NewState := S8;

          when S8 =>
            NewState := S9;

          when S9 =>
            NewState :=  S10;

          when S10 =>
            if (Count = 511) then
                NewState := S11;
            else
                NewState := S7;
            end if;

          when S11 =>
            if (Count = 63) then
```

```
                NewState := S1;
            else
                NewState := S11;
            end if;
        end case;
        NextState <= NewState after Delay;
    end process;
end BEHAVIORAL;

configuration CFG_NSL_BEHAVIORAL of NSL is
   for BEHAVIORAL
   end for;

end CFG_NSL_BEHAVIORAL;
```

B.3 VHDL code for Output Logic

We next list the VHDL code for the output logic which reads in the current state of the controller and gives the corresponding control signals to the datapath. The Output Logic component is shown in Figure 22.

```
-- VHDL Model Created from SGE Symbol out_l.sym -- Jan 13 10:42:19 1998

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;
   use work.components.all;

entity OUT_L is
      generic( Delay : TIME := 5 ns);
      Port (   START : In    std_logic;
               STATE : In    STATE_VALUE;
              STATUS : In    integer;
               COUNT : Out   std_logic;
                  E1 : Out   std_logic;
                  E2 : Out   std_logic;
                  E3 : Out   std_logic;
                  E4 : Out   std_logic;
              LOADAB : Out   std_logic;
               LOADP : Out   std_logic;
             LOADSUM : Out   std_logic;
               RESET : Out   std_logic;
                 RW1 : Out   std_logic;
                 RW2 : Out   std_logic;
                 RW3 : Out   std_logic;
                 RW4 : Out   std_logic;
                SEL1 : Out   std_logic;
                SEL2 : Out   std_logic;
                SEL3 : Out   std_logic;
                SEL4 : Out   std_logic;
                SEL5 : Out   std_logic;
                SEL6 : Out   std_logic;
                SEL7 : Out   std_logic;
                DONE : Out   std_logic);
```

```
end OUT_L;

architecture BEHAVIORAL of OUT_L is
begin
    process (State, Start, Status)
        variable Counter : unsigned(8 downto 0);
        variable VarDone : STD_LOGIC;
        variable VarCW   : CONTROL_WORD;
        variable i, j, k : integer;

        procedure DefaultCW (CW : out CONTROL_WORD) is
        begin
            -- Control Signals for Muxes
            CW.Sel1 := '0';
            CW.Sel2 := '0';
            CW.Sel3 := '0';
            CW.Sel4 := '0';
            CW.Sel5 := '0';
            CW.Sel6 := '0';
            CW.Sel7 := '0';

            -- Control Signals for Registers
            CW.LoadAB := '0';
            CW.LoadP := '0';
            CW.LoadSum := '0';

            -- Control Signal for Counter
            CW.Count := '0';
            CW.Reset := '0';

            -- Control Signals for Memories
            CW.E1 := '0';
            CW.RW1 := '0';
            CW.E2 := '0';
            CW.RW2 := '0';
            CW.E3 := '0';
            CW.RW3 := '0';
            CW.E4 := '0';
            CW.RW4 := '0';
        end DefaultCW;

        procedure OutputCW (CW : in CONTROL_WORD) is
        begin
            Sel1 <= CW.Sel1 after Delay;
            Sel2 <= CW.Sel2 after Delay;
            Sel3 <= CW.Sel3 after Delay;
            Sel4 <= CW.Sel4 after Delay;
            Sel5 <= CW.Sel5 after Delay;
            Sel6 <= CW.Sel6 after Delay;
            Sel7 <= CW.Sel7 after Delay;

            -- Control Signals for Registers
            LoadAB <= CW.LoadAB after Delay;
            LoadP <= CW.LoadP after Delay;
            LoadSum <= CW.LoadSum after Delay;
```

```

            -- Control Signal for Counter
            Count <= CW.Count after Delay;
            Reset <= CW.Reset after Delay;

            -- Control Signals for Memories
            E1 <= CW.E1 after Delay;
            RW1 <= CW.RW1 after Delay;
            E2 <= CW.E2 after Delay;
            RW2 <= CW.RW2 after Delay;
            E3 <= CW.E3 after Delay;
            RW3 <= CW.RW3 after Delay;
            E4 <= CW.E4 after Delay;
            RW4 <= CW.RW4 after Delay;
        end OutputCW;

    begin
        Counter := int_to_uvec(STATUS, 9);
        if (Counter(0) /= 'U' and Start /= 'U') then
            i := CONV_INTEGER(Counter(8 downto 6));
            j := CONV_INTEGER(Counter(5 downto 3));
            k := CONV_INTEGER(Counter(2 downto 0));

            DefaultCW(VarCW);

            case State is
              when S1 =>
                -- Counter := "000000000";
                VarCW.Reset := '1';
                VarCW.Count := '0';
                VarDone := '0';

              when S2 =>
                if (Counter = 63) then
                    -- Counter := "000000000";
                    VarCW.Reset := '1';
                    VarCW.Count := '0';
                else
                    -- Counter := Counter + 1;
                    VarCW.Reset := '0';
                    VarCW.Count := '1';
                end if;
                -- InBlock( j, k ) :=  Din;
                VarCW.Sel3 := '1';
                VarCW.E1 := '1';
                VarCW.RW1 := '0';

              when S3 =>
                -- A := InBlock(i, k);
                VarCW.LoadAB := '1';
                VarCW.Sel1 := '1';

                VarCW.Sel3 := '0';
                VarCW.E1 := '1';
                VarCW.RW1 := '1';
```

```
          -- B := COSBlock(j, k);
          VarCW.Sel4 := '0';
          VarCW.E2 := '1';
          VarCW.RW2 := '1';

        when S4 =>
          -- P := A * B;
          VarCW.LoadP := '1';

        when S5 =>
          if (k = 0) then
              -- Sum := P;
              VarCW.LoadSum := '1';
              VarCW.Sel2 := '0';
          else
              -- Sum := P + Sum;
              VarCW.LoadSum := '1';
              VarCW.Sel2 := '1';
          end if;

        when S6 =>
          if (k = 7) then
              -- TempBlock(i, j) := Sum;
              VarCW.Sel5 := '0';
              VarCW.E3 := '1';
              VarCW.RW3 := '0';
          end if;
          if (Counter = 511) then
              -- Counter := "000000000";
              VarCW.Reset := '1';
              VarCW.Count := '0';
          else
              -- Counter := Counter + 1;
              VarCW.Reset := '0';
              VarCW.Count := '1';
          end if;

        when S7 =>
          -- A := TempBlock(k, j);
          VarCW.LoadAB := '1';
          VarCW.Sel1 := '0';
          VarCW.E3 := '1';
          VarCW.RW3 := '1';
          VarCW.Sel5 := '1';

          -- B := COSBlock(i, k);
          VarCW.Sel4 := '1';
          VarCW.E2 := '1';
          VarCW.RW2 := '1';

        when S8 =>
          -- P := A * B;
          VarCW.LoadP := '1';

```

```
              when S9 =>
                if (k = 0) then
                    -- Sum := P;
                    VarCW.LoadSum := '1';
                    VarCW.Sel2 := '0';
                else
                    -- Sum := P + Sum;
                    VarCW.LoadSum := '1';
                    VarCW.Sel2 := '1';
                end if;

              when S10 =>
                if (k = 7) then
                    -- OutBlock(i, j) := Sum;
                    VarCW.Sel6 := '0';
                    VarCW.Sel7 := '0';
                    VarCW.E4 := '1';
                    VarCW.RW4 := '0';
                end if;
                if (Counter = 511) then
                    -- Counter := "000000000";
                    VarCW.Reset := '1';
                    VarCW.Count := '0';

                    VarDone := '1';
                else
                    -- Counter := Counter + 1;
                    VarCW.Reset := '0';
                    VarCW.Count := '1';
                end if;

              when S11 =>
                VarDone := '0';
                if (Counter = 63) then
                    -- Counter := "000000000";
                    VarCW.Reset := '1';
                    VarCW.Count := '0';
                else
                    -- Counter := Counter + 1;
                    VarCW.Reset := '0';
                    VarCW.Count := '1';
                end if;

                -- Dout <=  OutBlock(j, k);
                VarCW.Sel6 := '1';
                VarCW.Sel7 := '1';
                VarCW.E4 := '1';
                VarCW.RW4 := '1';
            end case;
            OutputCW(VarCW);
            Done <= VarDone;
        end if;
    end process;
end BEHAVIORAL;
```

```
configuration CFG_OUT_L_BEHAVIORAL of OUT_L is
   for BEHAVIORAL
   end for;
end CFG_OUT_L_BEHAVIORAL;
```

C Test Bench for the DCT models

The test bench feeds the DCT component with an input 8×8 image block that consists of a black-and-white "sandwich" pattern. The expected transform values have been computed using a C program.

```
--------------------------------------------------
-- Test Bench for DCT component
-- input image block is a black & white sandwitch
-- compare transform with values computed by C code
--
-- Gaurav Aggarwal; December 20, 1997.
--------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

entity tb is
end tb;

architecture behav of tb is
    component dct
        port( clk   : in  std_logic;
              start : in  std_logic;
              Din   : in  integer;
              Done  : out std_logic;
              Dout  : out integer);
    end component;

    signal start, Done : std_logic;
    signal clk : std_logic := '1';
    signal Din, Dout : integer;
begin

    U1 : dct
      port map (clk, start, Din, Done, Dout);

    clk <= not clk after 20 ns;

    start <= '0' after 0 ns,
             '1' after 50 ns,
             '0' after 80 ns;

    process
      type memory is array (0 to 7, 0 to 7) of integer;

      variable Result : memory := (
        (77785200, -21343500, 16837650, -5928750, 9248850, -711450, 5217300, 2371500),
        (-11375040, 3121200, -2462280, 867000, -1352520, 104040, -762960, -346800),
        (58882560, -16156800, 12745920, -4488000, 7001280, -538560, 3949440, 1795200),
        (-11542320, 3167100, -2498490, 879750, -1372410, 105570, -774180, -351900),
        (-13215120, 3626100, -2860590, 1007250, -1571310, 120870, -886380, -402900),
        (-6691200, 1836000, -1448400, 510000, -795600, 61200, -448800, -204000),
        (18066240, -4957200, 3910680, -1377000, 2148120, -165240, 1211760, 550800),
        (5854800, -1606500, 1267350, -446250, 696150, -53550, 392700, 178500));

```

```
    begin
        wait until Start =  '1';

        --------------------------------------
        -- give the input block: B&W sandwitch
        --------------------------------------
        for i in 0 to 23 loop
            wait until clk = '0';
            Din <= 255;
        end loop;

        for i in 24 to 39 loop
            wait until clk = '0';
            Din <= 0;
        end loop;

        for i in 40 to 63 loop
            wait until clk = '0';
            Din <= 255;
        end loop;

        -----------------------------
        -- wait till DCT unit is done
        -----------------------------
        wait until Done = '1';
        wait until clk = '1';

        --------------------------------------
        -- check the result with expect values
        --------------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '0';
                assert (Dout = Result(i, j))
                  report "DCT_TB: computation error" severity warning;
            end loop;
        end loop;
    end process;
end behav;


configuration cfg_tb of tb is
    for behav
    end for;
end cfg_tb;
```

D Behavioral model of Memory Optimized Design

In this appendix we give the behavioral model of the non-pipelined design for the memory optimized algorithm discussed in Section 5.

```
---------------------------------------------------
-- behavior of DCT component
-- compute the transform for a 8x8 image block
-- sequential memory optimized algorithm
--
-- Gaurav Aggarwal; December 26, 1997.
---------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity dct is
    port ( clk   : in  std_logic;
           start : in  std_logic;
           din   : in  integer;
           done  : out std_logic;
           dout  : out integer);
end dct;


architecture opt_beh of dct is
begin
    process
        type memory is array (0 to 7, 0 to 7) of integer;

        variable InBlock, OutBlock : memory;
        variable CosBlock : memory :=
          ((88,  122,  115,  103,  88,   69,   47,   24),
           (88,  103,   47,  -24, -88, -122, -115,  -69),
           (88,   69,  -47, -122, -88,   24,  115,  103),
           (88,   24, -115,  -69,  88,  103,  -47, -122),
           (88,  -24, -115,   69,  88, -103,  -47,  122),
           (88,  -69,  -47,  122, -88,  -24,  115, -103),
           (88, -103,   47,   24, -88,  122, -115,   69),
           (88, -122,  115, -103,  88,  -69,   47,  -24));
        variable a, b, c, d, p, prod, temp, sum : integer;
    begin

        -----------------------------
        -- wait for the start signal
        -----------------------------
        wait until start = '1';
        done <= '0';
        wait until clk = '1';

        ----------------------------------
        -- read input 8x8 block of pixels
        ----------------------------------
        for i in 0 to 7 loop
```

```
            for j in 0 to 7 loop
                wait until clk = '1';
                InBlock (i, j) := din;
            end loop;
        end loop;

        ------------------------------
        -- the matrix multiplications
        ------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop

                -----------------------------------
                -- generate one entry of TempBlock
                -----------------------------------
                for k in 0 to 7 loop
                    A := InBlock (i, j);
                    B := CosBlock (j, k);

                    P := A * B;

                    if (k = 0) then
                        sum := P;
                    elsif (k = 7) then
                        temp := sum + P;
                    else
                        sum := sum + P;
                    end if;
                end loop;

                -------------------------------------
                -- now use this entry for generating
                -- partial sums in the OutBlock
                -------------------------------------
                for k in 0 to 7 loop
                    C := OutBlock (k, j);
                    D := CosBlock (k, i);

                    prod := d * temp;

                    if (i = 0) then
                        OutBlock (k, j) := prod;
                    else
                        OutBlock (k, j) := C + prod;
                    end if;
                end loop;
            end loop;
        end loop;

        ------------------------
        -- give the done signal
        ------------------------
        wait until clk = '1';
        done <= '1';

```

```
        ------------------------------
        -- output the computed matrix
        ------------------------------
        for i in 0 to 7 loop
            for j in 0 to 7 loop
                wait until clk = '1';
                done <= '0';
                dout <= OutBlock (i, j);
            end loop;
        end loop;
    end process;
end opt_beh;
```

E Behavioral model of Pipelined Memory Optimized Design

In this appendix, we give the behavioral model of the loop and functional unit pipelined design for the memory optimized algorithm. The multiplier has two stages and there are eight stages in the pipeline. The model consists of separate states for filling and flushing of the pipeline. This design is dicussed in Section 5.1.

```
-------------------------------------------------------
-- Behavioral model of pipelined memory optimized DCT
-- no temporary matrix. single word used
--
-- December 18, 1997.
--------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity dct is
    port (
          clk   : in  std_logic;
          start : in  std_logic;
          din   : in  integer;
          done  : out std_logic;
          dout  : out integer
         );
end dct;


architecture pipe_beh of dct is
    type STATES is (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9,
                    S10, S11, S12, S13, S14, S15, S16, S17, S18);
    signal oldtemp, temp, tempProd, Prod, P, tempP : integer;
    signal A, B, C, D, sum, oldC, olderC : integer;
    signal fCount0, fCount1 : unsigned (8 downto 0) := "000000000";
    signal fCount2, fCount3 : unsigned (8 downto 0) := "000000000";
    signal sCount0, sCount1 : unsigned (8 downto 0) := "000000000";
    signal sCount2, sCount3 : unsigned (8 downto 0) := "000000000";
    signal state : STATES := S0;
begin

    process (clk)
        type memory is array (0 to 7, 0 to 7) of integer;

        variable InBlock, OutBlock : memory;
        variable CosBlock : memory := (
            ( 125,  122,  115,  103,  88,   69,   47,   24 ),
            ( 125,  103,   47,  -24, -88, -122, -115,  -69 ),
            ( 125,   69,  -47, -122, -88,   24,  115,  103 ),
            ( 125,   24, -115,  -69,  88,  103,  -47, -122 ),
            ( 125,  -24, -115,   69,  88, -103,  -47,  122 ),
            ( 125,  -69,  -47,  122, -88,  -24,  115, -103 ),
            ( 125, -103,   47,   24, -88,  122, -115,   69 ),
            ( 125, -122,  115, -103,  88,  -69,   47,  -24 ));

        variable i0, j0, k0, l0, m0, n0 : integer;
```

```
        variable k3, l3, m3, n3: integer;
    begin
        if (clk='1') then
            -- separate the bits of the 9 bit counter
            i0 := conv_integer(fCount0(8 downto 6));
            j0 := conv_integer(fCount0(5 downto 3));
            k0 := conv_integer(fCount0(2 downto 0));
            k3 := conv_integer(fCount3(2 downto 0));

            l0 := conv_integer(sCount0(8 downto 6));
            m0 := conv_integer(sCount0(5 downto 3));
            n0 := conv_integer(sCount0(2 downto 0));
            l3 := conv_integer(sCount3(8 downto 6));
            m3 := conv_integer(sCount3(5 downto 3));
            n3 := conv_integer(sCount3(2 downto 0));

            case state is
              when S0 =>
                  done <= '0';
                  if (start = '1') then
                      state <= S1;
                      fCount0 <= (others=>'0');
                  else
                      state <= S0;
                  end if;

              -- read the incoming matrix into InBlock
              when S1 =>
                  InBlock (j0, k0) := din;
                  if (fCount0 = 63) then
                      fCount0 <= (others=>'0');
                      state <= S2;
                  else
                      fCount0 <= fCount0 + 1;
                      state <= S1;
                  end if;

              when S2 =>
                  -- stage 1
                  A <= InBlock (i0, k0);
                  B <= CosBlock (j0, k0);

                  fCount1 <= fCount0;
                  fCount0 <= fCount0 + 1;
                  state <= S3;

              -- start filing up pipeline for first multiplication
              when S3 =>
                  -- stage 1
                  A <= InBlock (i0, k0);
                  B <= CosBlock (j0, k0);
                  fCount1 <= fCount0;
                  fCount0 <= fCount0 + 1;

                  -- stage 2
```

```
                tempP <= A * B;
                fCount2 <= fCount1;

                state <= S4;

            when S4 =>
                -- stage 1
                A <= InBlock (i0, k0);
                B <= CosBlock (j0, k0);
                fCount1 <= fCount0;
                fCount0 <= fCount0 + 1;

                -- stage 2
                tempP <= A * B;
                fCount2 <= fCount1;

                -- stage 3
                P <= tempP;
                fCount3 <= fCount2;

                state <= S5;

            when S5 =>
                -- stage 1
                A <= InBlock (i0, k0);
                B <= CosBlock (j0, k0);
                fCount1 <= fCount0;
                fCount0 <= fCount0 + 1;

                -- stage 2
                tempP <= A * B;
                fCount2 <= fCount1;

                -- stage 3
                P <= tempP;
                fCount3 <= fCount2;

                -- stage 4
                if (k3 = 0) then
                    sum <= P;
                elsif (k3 = 7) then
                    oldtemp <= sum + P;
                else
                    sum <= sum + P;
                end if;

                if (fCount0 = 10) then
                    state <= S6;
                    sCount0 <= (others=>'0');
                else
                    state <= S5;
                end if;

            when S6 =>
                -- stage 1
```

```
                    A <= InBlock (i0, k0);
                    B <= CosBlock (j0, k0);
                    fCount1 <= fCount0;
                    fCount0 <= fCount0 + 1;

                    -- stage 2
                    tempP <= A * B;
                    fCount2 <= fCount1;

                    -- stage 3
                    P <= tempP;
                    fCount3 <= fCount2;

                    -- stage 4
                    if (k3 = 0) then
                        sum <= P;
                    elsif (k3 = 7) then
                        oldtemp <= sum + P;
                    else
                        sum <= sum + P;
                    end if;

                    -- stage 5
                    C <= OutBlock (n0, m0);
                    D <= CosBlock (n0, l0);
                    temp <= oldtemp;
                    sCount1 <= sCount0;
                    sCount0 <= sCount0 + 1;

                    state <= S7;

                when S7 =>
                    -- stage 1
                    A <= InBlock (i0, k0);
                    B <= CosBlock (j0, k0);
                    fCount1 <= fCount0;
                    fCount0 <= fCount0 + 1;

                    -- stage 2
                    tempP <= A * B;
                    fCount2 <= fCount1;

                    -- stage 3
                    P <= tempP;
                    fCount3 <= fCount2;

                    -- stage 4
                    if (k3 = 0) then
                        sum <= P;
                    elsif (k3 = 7) then
                        oldtemp <= sum + P;
                    else
                        sum <= sum + P;
                    end if;
```

```
                    -- stage 5
                    C <= OutBlock (n0, m0);
                    D <= CosBlock (n0, l0);
                    temp <= oldtemp;
                    sCount1 <= sCount0;
                    sCount0 <= sCount0 + 1;

                    -- stage 6
                    tempProd <= D * temp;
                    oldC <= C;
                    sCount2 <= sCount1;

                    state <= S8;

                when S8 =>
                    -- stage 1
                    A <= InBlock (i0, k0);
                    B <= CosBlock (j0, k0);
                    fCount1 <= fCount0;
                    fCount0 <= fCount0 + 1;

                    -- stage 2
                    tempP <= A * B;
                    fCount2 <= fCount1;

                    -- stage 3
                    P <= tempP;
                    fCount3 <= fCount2;

                    -- stage 4
                    if (k3 = 0) then
                        sum <= P;
                    elsif (k3 = 7) then
                        oldtemp <= sum + P;
                    else
                        sum <= sum + P;
                    end if;

                    -- stage 5
                    C <= OutBlock (n0, m0);
                    D <= CosBlock (n0, l0);
                    temp <= oldtemp;
                    sCount1 <= sCount0;
                    sCount0 <= sCount0 + 1;

                    -- stage 6
                    tempProd <= D * temp;
                    oldC <= C;
                    sCount2 <= sCount1;

                    -- stage 7
                    Prod <= tempProd;
                    olderC <= oldC;
                    sCount3 <= sCount2;
```

```
                    state <= S9;

                when S9 =>
                    -- stage 1
                    A <= InBlock (i0, k0);
                    B <= CosBlock (j0, k0);
                    fCount1 <= fCount0;
                    fCount0 <= fCount0 + 1;

                    -- stage 2
                    tempP <= A * B;
                    fCount2 <= fCount1;

                    -- stage 3
                    P <= tempP;
                    fCount3 <= fCount2;

                    -- stage 4
                    if (k3 = 0) then
                        sum <= P;
                    elsif (k3 = 7) then
                        oldtemp <= sum + P;
                    else
                        sum <= sum + P;
                    end if;

                    -- stage 5
                    C <= OutBlock (n0, m0);
                    D <= CosBlock (n0, l0);
                    temp <= oldtemp;
                    sCount1 <= sCount0;
                    sCount0 <= sCount0 + 1;

                    -- stage 6
                    tempProd <= D * temp;
                    oldC <= C;
                    sCount2 <= sCount1;

                    -- stage 7
                    Prod <= tempProd;
                    olderC <= oldC;
                    sCount3 <= sCount2;

                    -- stage 8
                    if (l3 = 0) then
                        OutBlock (n3, m3) := Prod;
                    else
                        OutBlock (n3, m3) := olderC + Prod;
                    end if;

                    if (fCount0 = 511) then
                        state <= S10;
                    else
                        state <= S9;
                    end if;
```

```
            -- begin flushing the first mult stages
            when S10 =>
                -- stage 2
                tempP <= A * B;
                fCount2 <= fCount1;

                -- stage 3
                P <= tempP;
                fCount3 <= fCount2;

                -- stage 4
                if (k3 = 0) then
                    sum <= P;
                elsif (k3 = 7) then
                    oldtemp <= sum + P;
                else
                    sum <= sum + P;
                end if;

                -- stage 5
                C <= OutBlock (n0, m0);
                D <= CosBlock (n0, l0);
                temp <= oldtemp;
                sCount1 <= sCount0;
                sCount0 <= sCount0 + 1;

                -- stage 6
                tempProd <= D * temp;
                oldC <= C;
                sCount2 <= sCount1;

                -- stage 7
                Prod <= tempProd;
                olderC <= oldC;
                sCount3 <= sCount2;

                -- stage 8
                if (l3 = 0) then
                    OutBlock (n3, m3) := Prod;
                else
                    OutBlock (n3, m3) := olderC + Prod;
                end if;

                state <= S11;

            -- flushing the first mult stages
            when S11 =>
                -- stage 3
                P <= tempP;
                fCount3 <= fCount2;

                -- stage 4
                if (k3 = 0) then
                    sum <= P;
```

```
                elsif (k3 = 7) then
                    oldtemp <= sum + P;
                else
                    sum <= sum + P;
                end if;

                -- stage 5
                C <= OutBlock (n0, m0);
                D <= CosBlock (n0, l0);
                temp <= oldtemp;
                sCount1 <= sCount0;
                sCount0 <= sCount0 + 1;

                -- stage 6
                tempProd <= D * temp;
                oldC <= C;
                sCount2 <= sCount1;

                -- stage 7
                Prod <= tempProd;
                olderC <= oldC;
                sCount3 <= sCount2;

                -- stage 8
                if (l3 = 0) then
                    OutBlock (n3, m3) := Prod;
                else
                    OutBlock (n3, m3) := olderC + Prod;
                end if;

                state <= S12;


            -- flushing the first mult stages
            when S12 =>
                -- stage 4
                if (k3 = 0) then
                    sum <= P;
                elsif (k3 = 7) then
                    oldtemp <= sum + P;
                else
                    sum <= sum + P;
                end if;

                -- stage 5
                C <= OutBlock (n0, m0);
                D <= CosBlock (n0, l0);
                temp <= oldtemp;
                sCount1 <= sCount0;
                sCount0 <= sCount0 + 1;

                -- stage 6
                tempProd <= D * temp;
                oldC <= C;
                sCount2 <= sCount1;
```

```

                -- stage 7
                Prod <= tempProd;
                olderC <= oldC;
                sCount3 <= sCount2;

                -- stage 8
                if (l3 = 0) then
                    OutBlock (n3, m3) := Prod;
                else
                    OutBlock (n3, m3) := olderC + Prod;
                end if;

                state <= S13;


            -- flushed the first mult stages; only second mult
            when S13 =>

                -- stage 5
                C <= OutBlock (n0, m0);
                D <= CosBlock (n0, l0);
                temp <= oldtemp;
                sCount1 <= sCount0;
                sCount0 <= sCount0 + 1;

                -- stage 6
                tempProd <= D * temp;
                oldC <= C;
                sCount2 <= sCount1;

                -- stage 7
                Prod <= tempProd;
                olderC <= oldC;
                sCount3 <= sCount2;

                -- stage 8
                if (l3 = 0) then
                    OutBlock (n3, m3) := Prod;
                else
                    OutBlock (n3, m3) := olderC + Prod;
                end if;

                if (sCount0 = 511) then
                    state <= S14;
                else
                    state <= S13;
                end if;

            -- begin flushing the second mult stages
            when S14 =>
                -- stage 6
                tempProd <= D * temp;
                oldC <= C;
                sCount2 <= sCount1;
```

```
                        -- stage 7
                        Prod <= tempProd;
                        olderC <= oldC;
                        sCount3 <= sCount2;

                        -- stage 8
                        if (l3 = 0) then
                            OutBlock (n3, m3) := Prod;
                        else
                            OutBlock (n3, m3) := olderC + Prod;
                        end if;

                        state <= S15;


                    -- flushing the second mult stages
                    when S15 =>
                        -- stage 7
                        Prod <= tempProd;
                        olderC <= oldC;
                        sCount3 <= sCount2;

                        -- stage 8
                        if (l3 = 0) then
                            OutBlock (n3, m3) := Prod;
                        else
                            OutBlock (n3, m3) := olderC + Prod;
                        end if;

                        state <= S16;

                    -- flushing the second mult stages
                    when S16 =>
                        -- stage 8
                        if (l3 = 0) then
                            OutBlock (n3, m3) := Prod;
                        else
                            OutBlock (n3, m3) := olderC + Prod;
                        end if;

                        state <= S17;

                    -- finished with computation. give done signal
                    when S17 =>
                        done <= '1';
                        fCount0 <= (others=>'0');
                        state <= S18;

                    -- output the OutBlock matrix
                    when S18 =>
                        dout <= OutBlock (j0, k0);
                        if (fcount0 = 63) then
                            fCount0 <= (others=>'0');
                            state <= S0;
```

```
                    else
                        fCount0 <= fCount0 + 1;
                        state <= S18;
                    end if;

                when others =>
            end case;
        end if;
    end process;
end pipe_beh;
```