ECE 428 Programmable ASIC Design

Design Example 2:
Halftone Pixel Image Converter

Haibo Wang
ECE Department
Southern Illinois University
Carbondale, IL 62901

Background

- Image to be processed consists of 6×8 pixels.
- Each pixel is represented by an 8-bit data (black and white image).
- In the conversion, there are five values associated with a pixel.

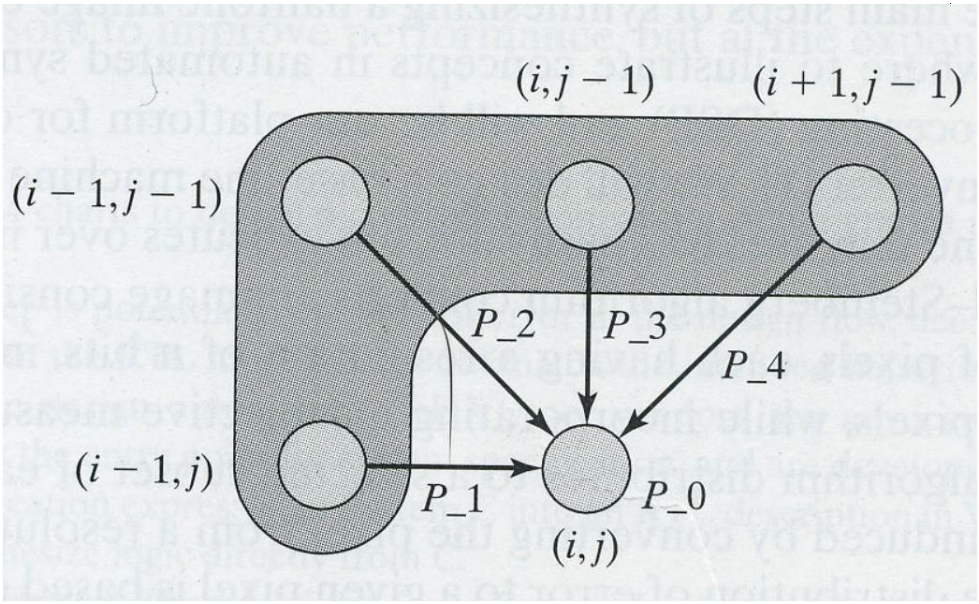
— Assume the pixel is located at the i^{th} column and j^{th} row

• PV(i,j) :	pixel value	8-bit	input
• CPV(i,j):	Corrected pixel value	8-bit	intermediate value
• e(i,j):	error of the pixel	8-bit	intermediate value
• E_av(i,j):	Average error	8-bit	intermediate value
• HTPV(i,j):	Halftone pixel value	1-bit	output

Calculation of Average Error

- Equation for calculating E_av

$$E\_av = \frac{w_1 \bullet e(i-1, j) + w_2 \bullet e(i-1, j-1) + w_3 \bullet e(i, j-1) + w_4 \bullet e(i+1, j-1)}{w_1 + w_2 + w_3 + w_4}$$



Floyd-Steinberg Algorithm

```
For each pixel
   CPV = PV(i,j) + E_av;
   if CPV < CPV_thresh
         HTPV(i,j) = 0;
         e(i,j) = CPV;
    else
         HTPV(i,j) = 1;
         e(i,j) = CPV-CPV_max
```

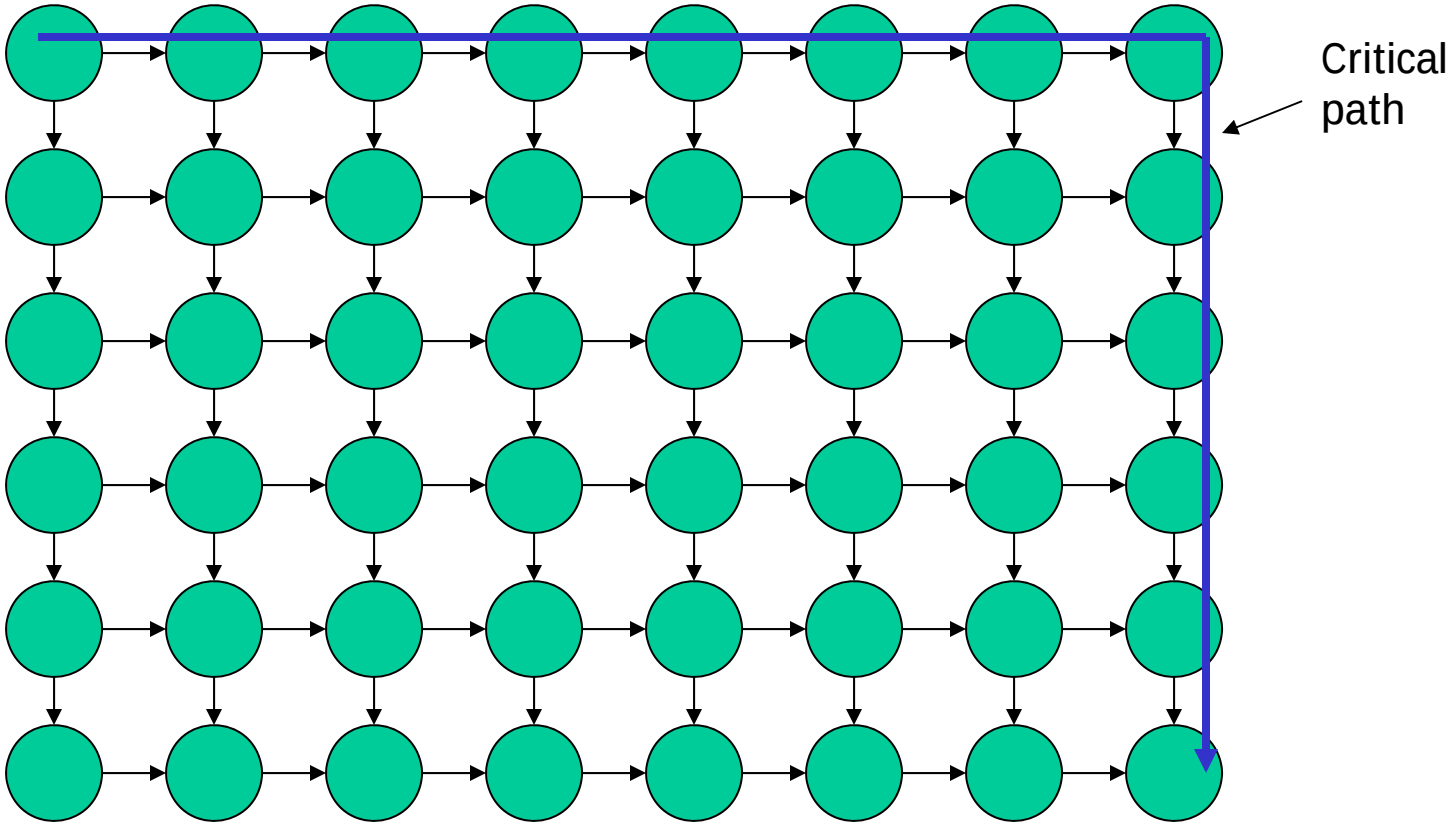
(CPV_thresh=128 & CPV_max=255)

Pseudo-code

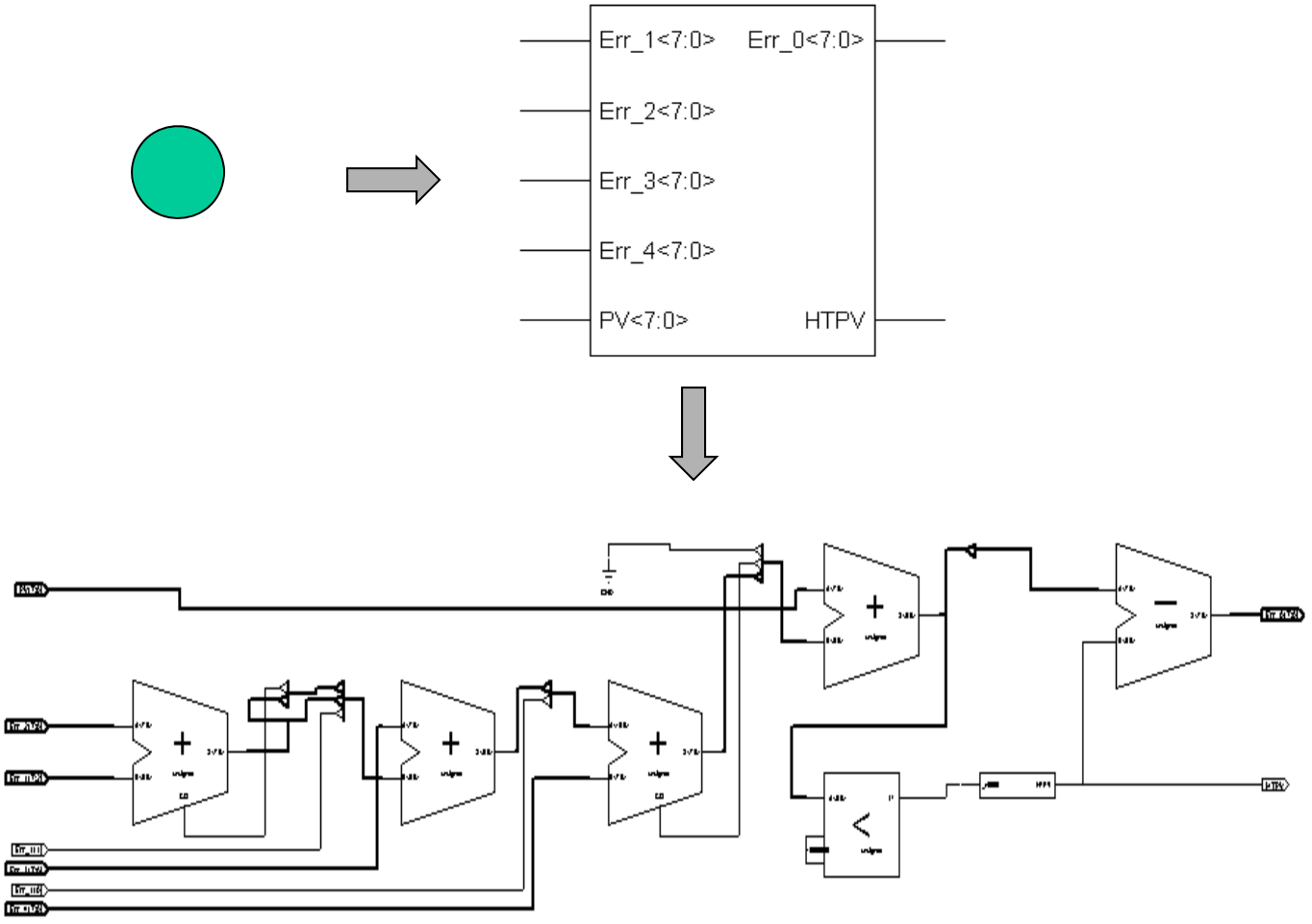
```
for(k=1;k<N;k++)  {Err[k][0]=0;}
for(k=0;k<=M;k++) {Err[0][k]=0;Err[N+1][0]=0;}  //boundary initialization
for(j=1;j<M;j++)
{
    for(i=1;i<N;i++)
    {
        E_av=(7*Err[i-1][j]+1*Err[i-1][j-1]+5*Err[i][j-1]+3*Err[i+1][j-
    1])/16;
        CPV=PV[i][j]+E_av;
        CPV_round=(if CPV<T then 0 else 255);      //T is thresh=128;
        HTPV[i][j]=if(CPV_round==0 then 0 else 1);
        Err[i][j]=CPV-CPV_round;
    }
}
```

Implementation 1

- Using 48 identical processing unit



Design of Pixel Processing Unit

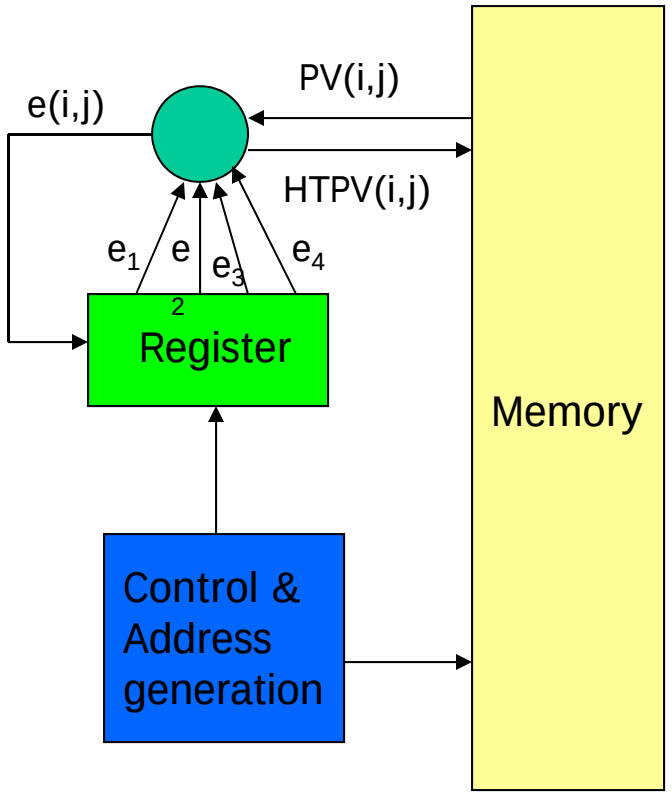


Design summary of Implementation 1

- High hardware cost
- Demand high memory throughput (or multiple-port memory)
- Slow performance (critical path delay is $14*t_{ppu}$)
- Straightforward design
- No control logic

Implementation 2

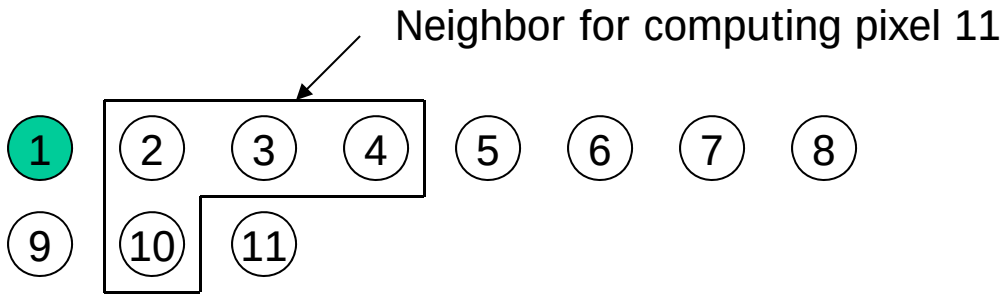
- Using a single processing unit to sequentially process each pixel
- Using register file to store intermediate data during processing
- Control and address generation circuits determine where to load data and which pixel to be processed
- Total processing time $48*t_{ppu}$



e_1: e(i-1,j); e_2: e(i-1,j-1);
e_3: e(i,j-1); e_4: e(i+1,j-1);

Register File Design

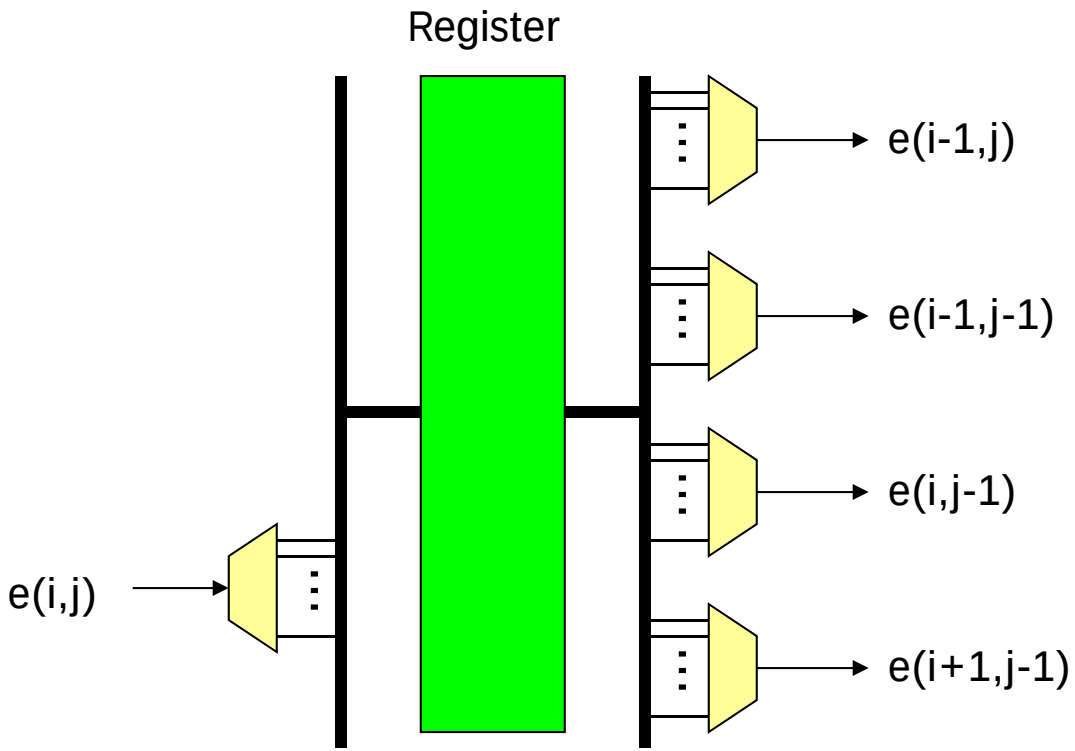
- Register file does not have to contain 48 registers since not all intermediate date need to be stored during the same time period.
  - Assuming that the processing order is 1, 2, 3, ... to 48
  - After pixel 10 is processed, error of pixel is not needed. Hence, register for e_1 can be released for e_{10}



  - Only 9 registers are needed

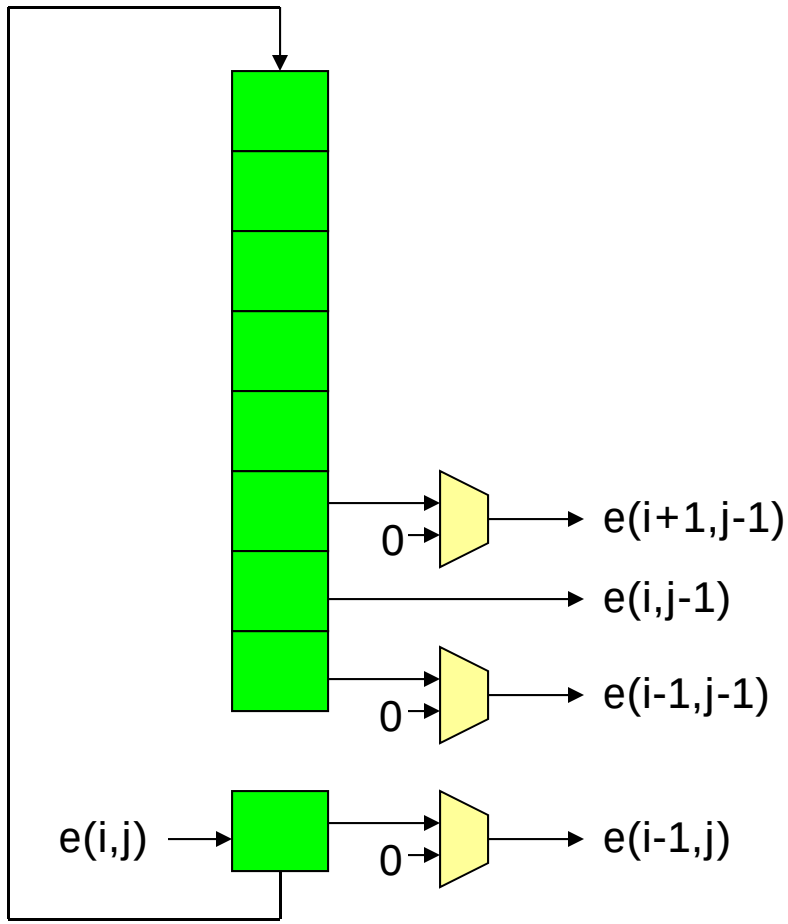
Register File Design

- Using 9-to-1 multiplexers and 1-to-9 demultiplexer is straightforward design approach. However, it results in large hardware and slow performance



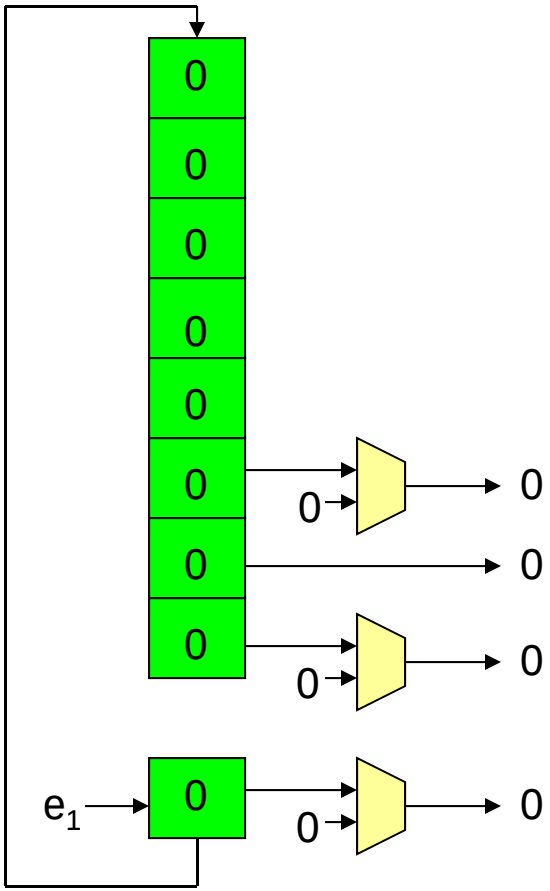
Register File Design

- A better design approach is to use shift-register based approach

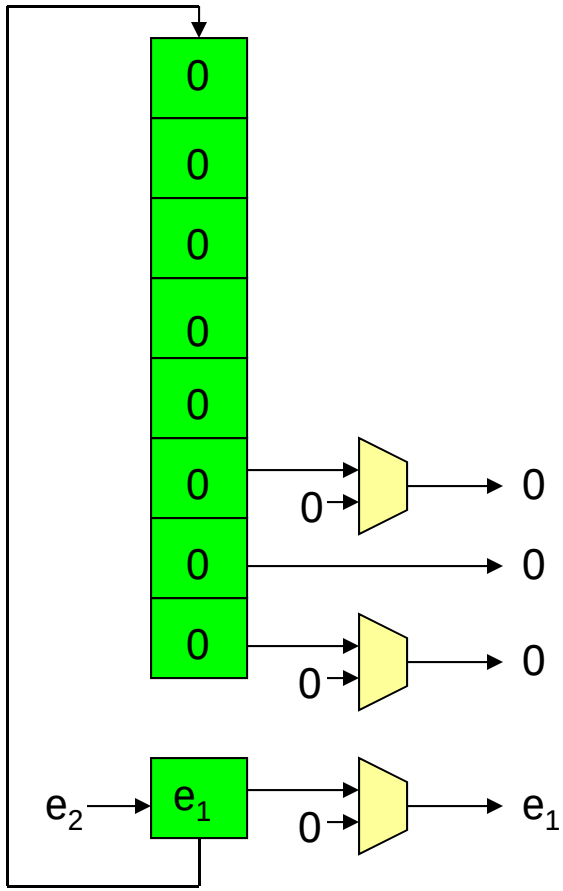


Register File Design

- Illustration of the operation



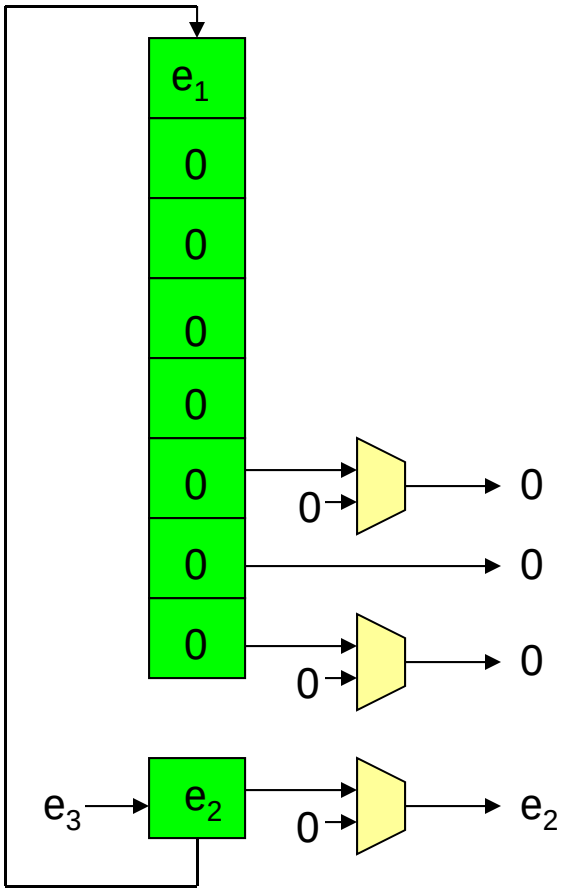
First clock cycle Pixel 1



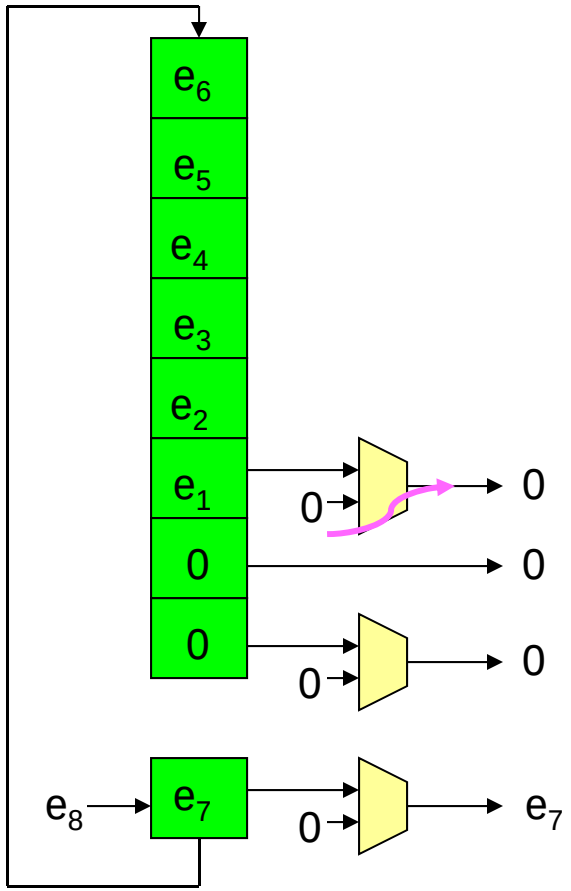
Second clock cycle Pixel 2

Register File Design

- Illustration of the operation



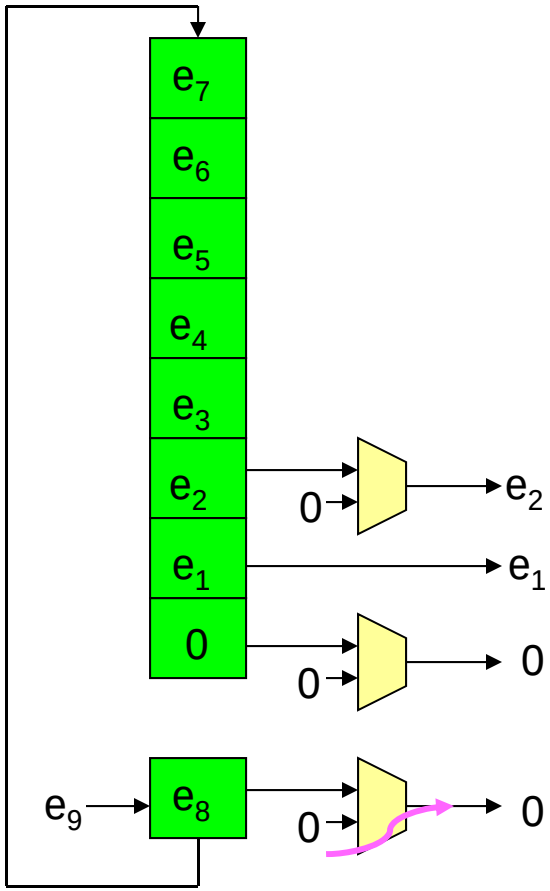
Third clock cycle Pixel 3



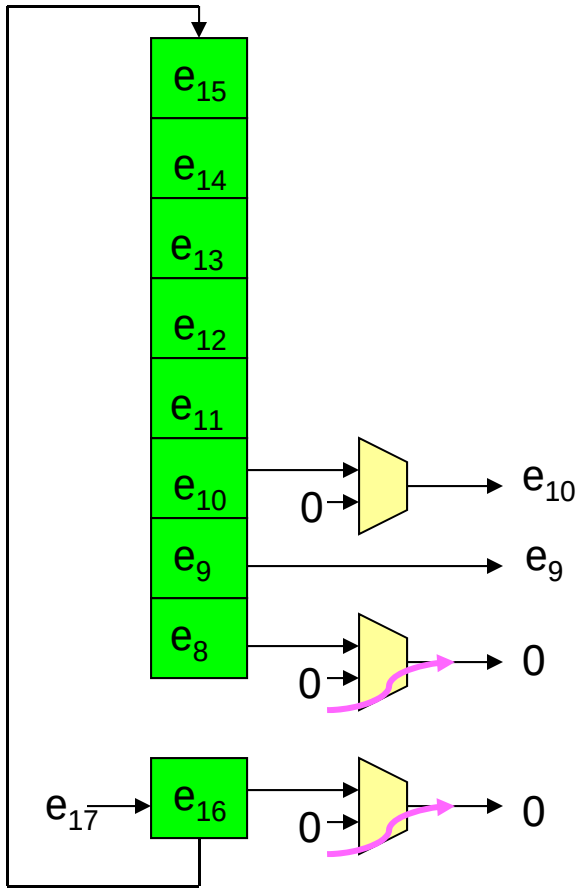
8th clock cycle Pixel 8

Register File Design

- Illustration of the operation



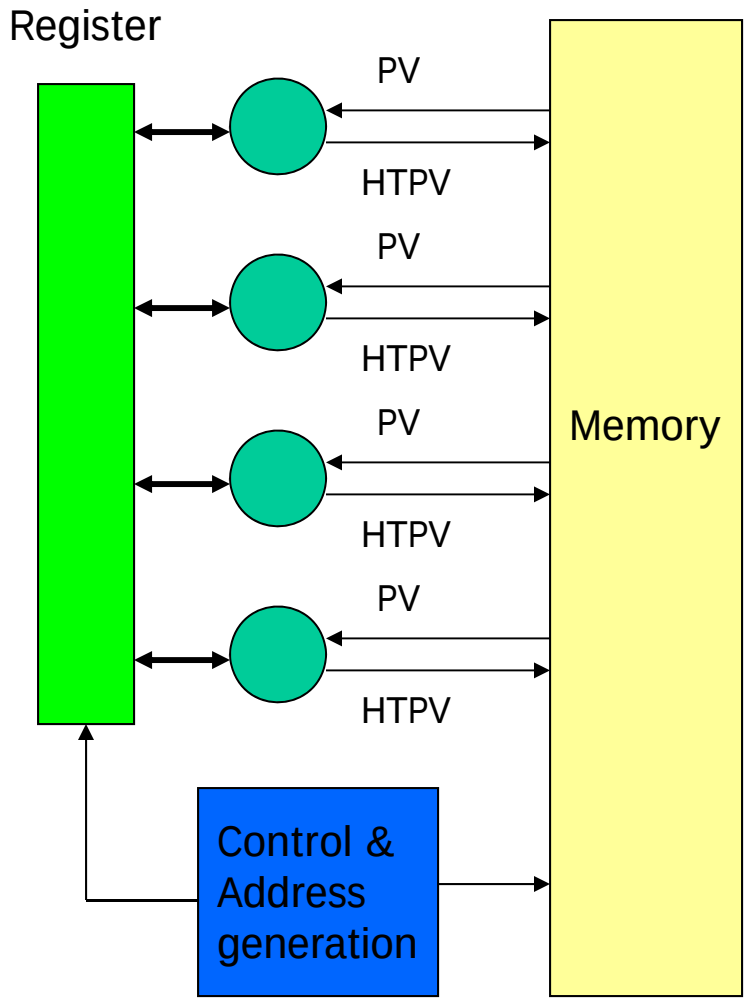
9th clock cycle Pixel 9



17th clock cycle Pixel 17

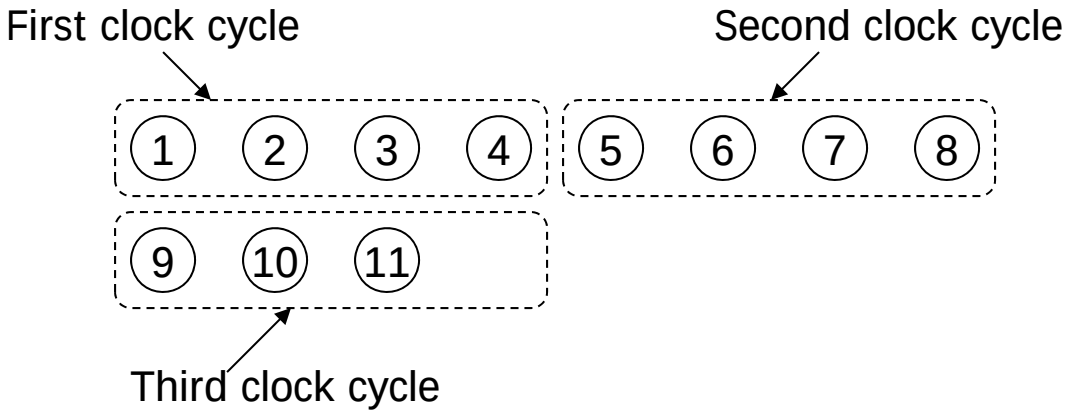
Implementation 3

- Using four processing units
- Ideally, we want the computation of the four processing units are carried out simultaneously. Hence, the clock cycle equal the delay of one processing unit.
- Which is the best order of computation?



Computation order in Implementation 3

- The straightforward solution does not leads to optimal performance.

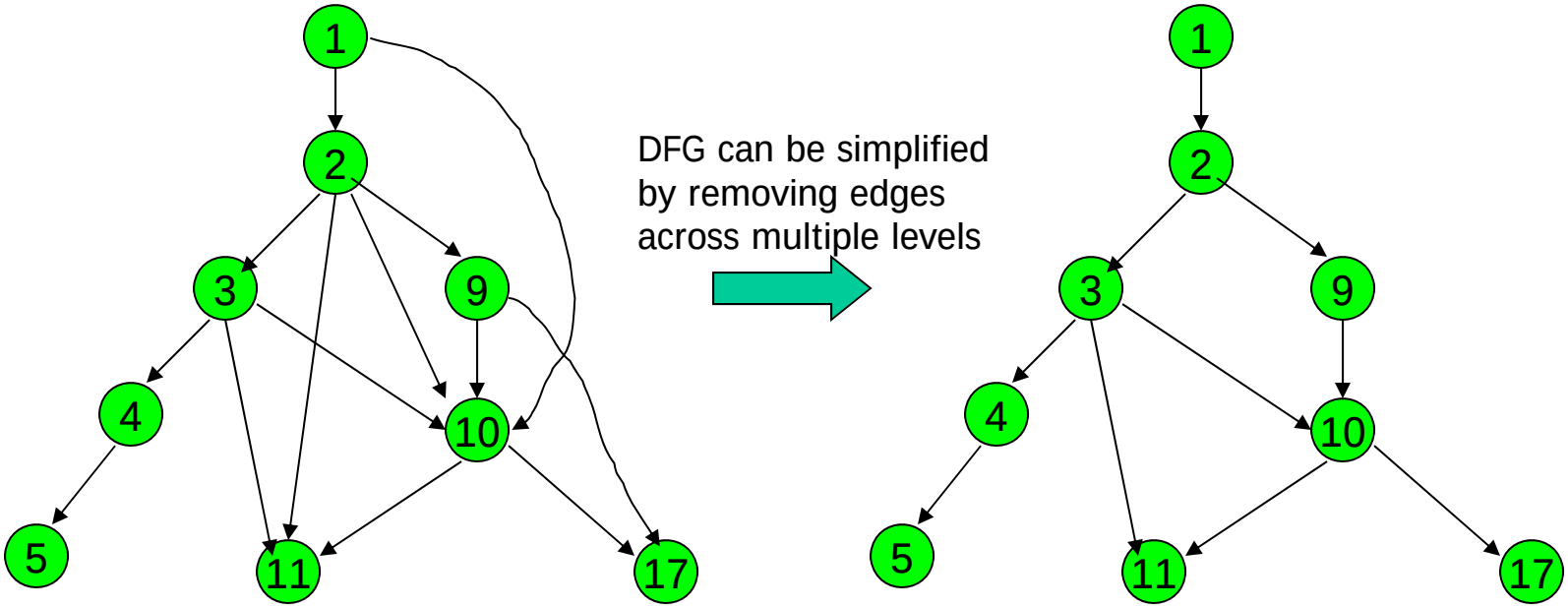


- In the above approach, the computation for pixel 4 can be started only if the computation for pixels 1, 2, and 3 are complete.

- The clock cycle is $4*t_{ppu}$

Computation order in Implementation 3

- The best computation order is determined by data dependence in the the given application.
- An easy method to find data dependence is to use data flow graph
- Data flow graph (DFG) of the halftone pixel converter:

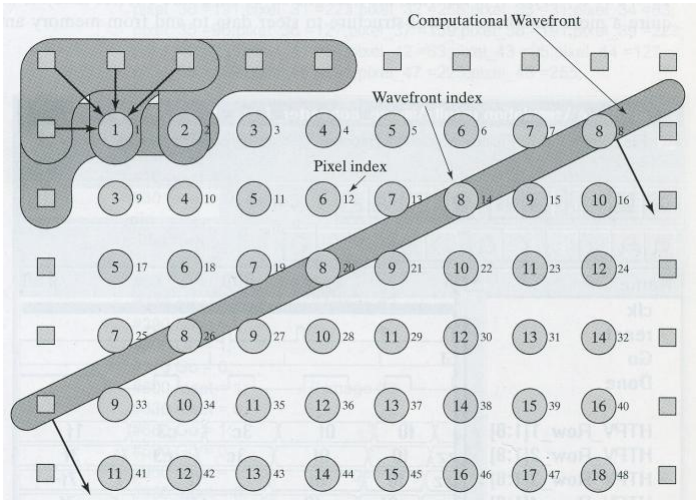


Computation order in Implementation 3

- Optimal computation schedule according to data dependence

		Time slots																	
		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}
Processors	P_1	1	2	3	4	5	6	7	8	15	16	23	24						
	P_2			9	10	11	12	13	14	21	22	29	30	31	32				
	P_3					17	18	19	20	27	28	35	36	37	38	39	40		
	P_4							25	26	33	34	41	42	43	44	45	46	47	48

- It take 18 clock cycles to convert a frame
- If multiple frames to be processes, some parts of the computation can be overlapped. Thus, it takes 12 cycles to convert a frame



Register File Design

- In order to find the number of registers needed, we can find the the lifetime of each register. Registers whose lifetimes do not overlap can be collapsed.