CS2102 02
Convolution Engine for Image Processing

黃稚存

國立清華大學
資訊工程學系

Final Project

Objective

- Matrix convolution (2D convolution) is a popular computation in digital signal processing
  - Image processing
  - Signal processing
  - Deep learning (convolution neural network, CNN)
  - Etc.

- Design a simple image processing filter (also known as kernel) for edge detection
  - For gray-scale 256x256 images
  - Refer to Wikipedia for more description: https://en.wikipedia.org/wiki/Kernel_(image_processing)

What is Convolution?

- 2D convolution
- Inner product
- Also called *filter* (or kernel) in the image processing

filter

1	2	3
4	5	6
7	8	9

Image

a	b	c
d	e	f
g	h	i

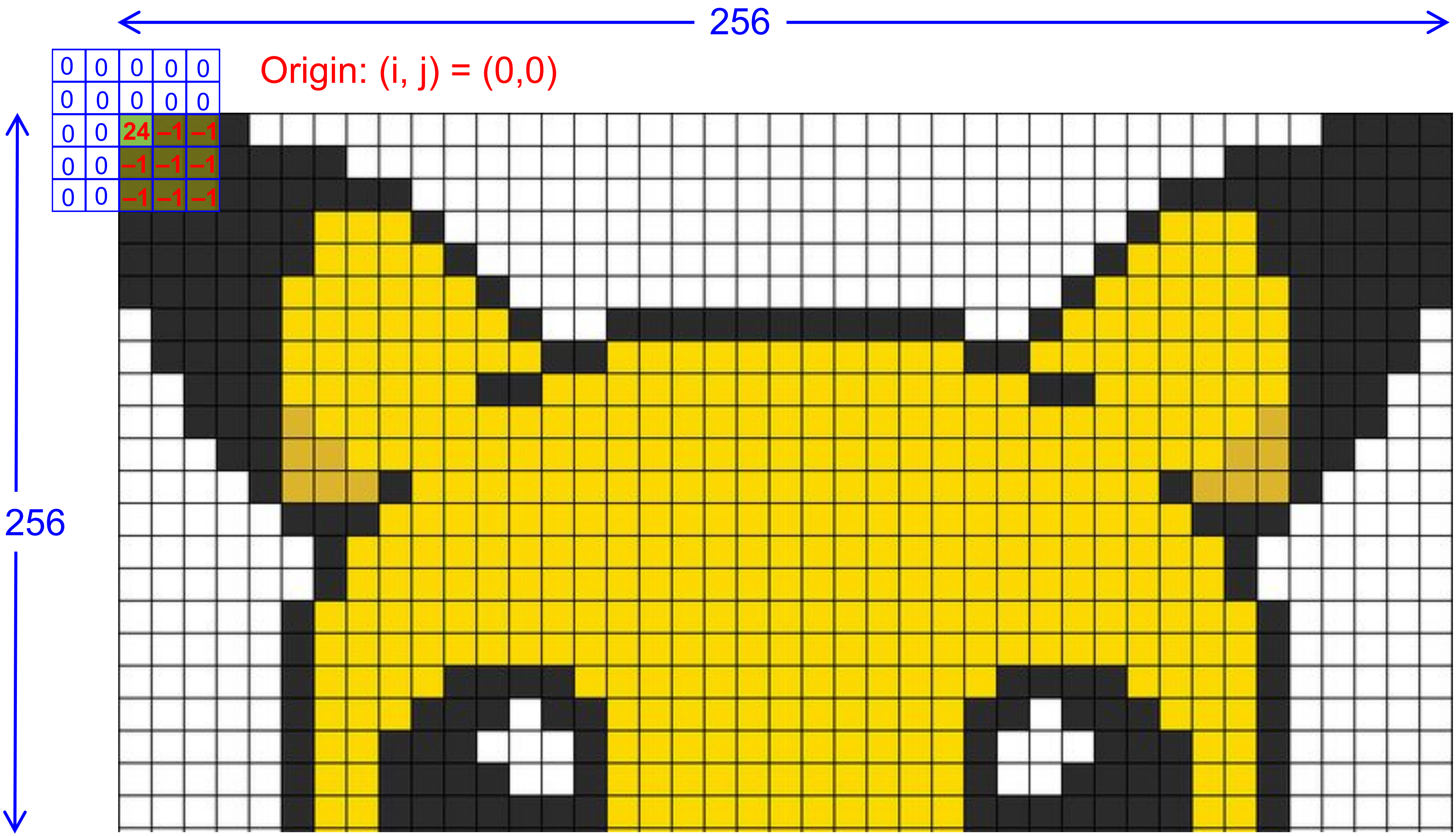
$$y = 1 * a + 2 * b + 3 * c + 4 * d + 5 * e + 6 * f + 7 * g + 8 * h + 9 * i$$

Convolution Filter (Kernel) of Edge Detection

- Emphasize the edges of the image
- 5x5 convolution filter (kernel)

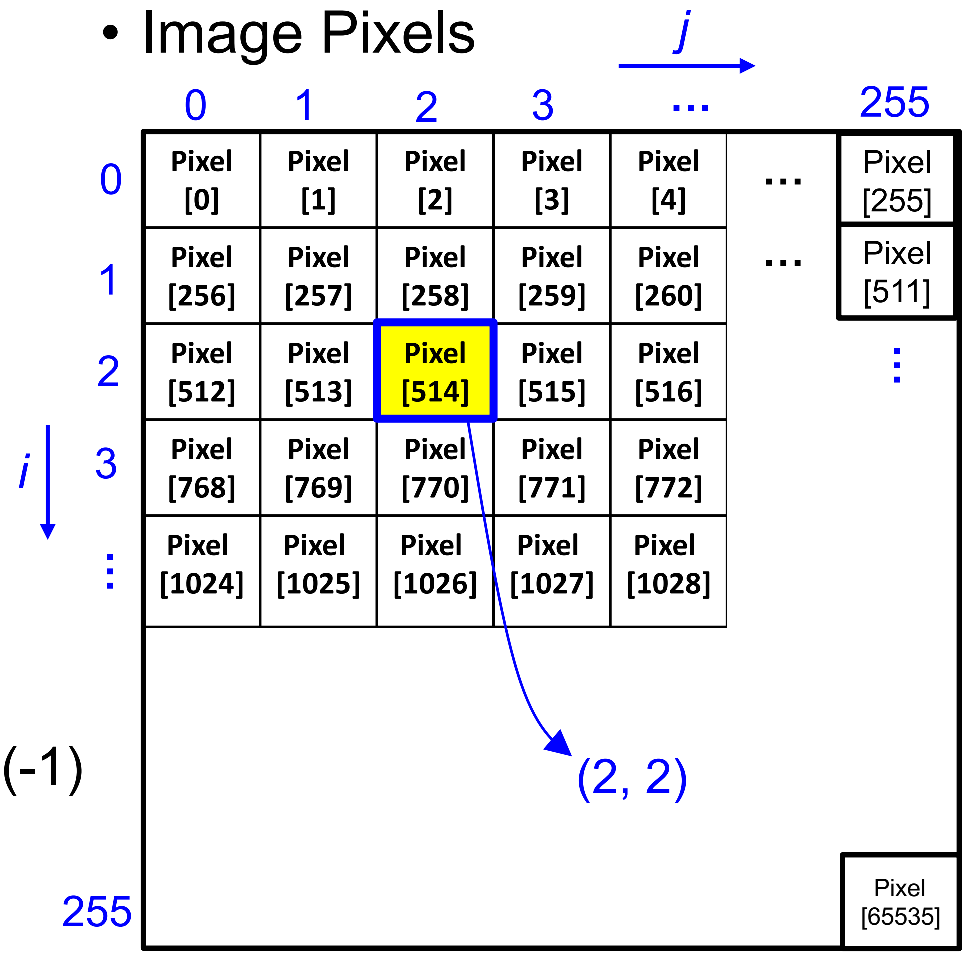
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	24	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

Apply to The Entire Image



Ex: To Compute The 514th Pixel

- Image Pixels



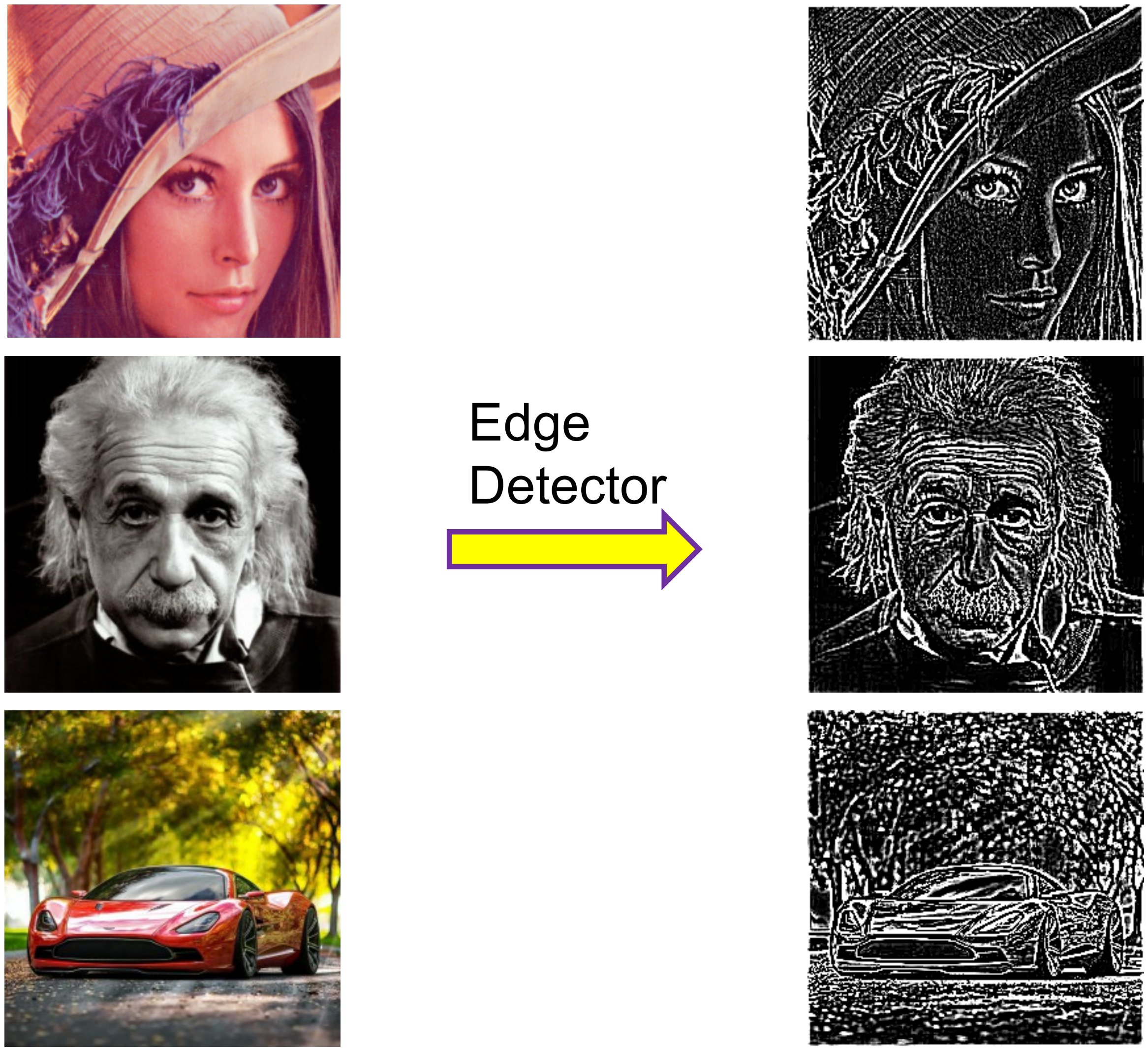
(addr = i * 256 + j)

- Filter Coefficients

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
-1	-1	24	-1	-1
-1	-1	-1	-1	-1
-1	-1	-1	-1	-1

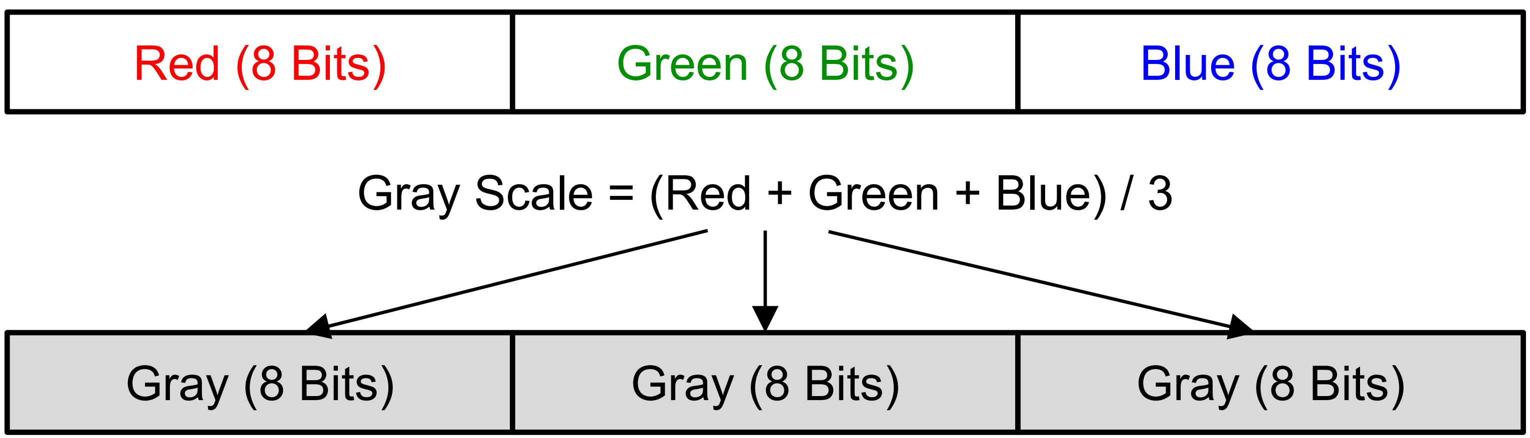
- result =
  Pixel[0]*(-1) + …… + Pixel[4]*(-1)
  + Pixel[256]*(-1) + …… + Pixel[260]*(-1)
  + Pixel[512]*(-1) + Pixel[513]*(-1)
  + Pixel[514]*(24)
  + Pixel[515]*(-1) + Pixel[516]*(-1)
  + Pixel[768]*(-1) + …… + Pixel[772]*(-1)
  + Pixel[1024]*(-1) + …… + Pixel[1028]*(-1)
- If result > 255, result = 255
- If result < 0, result = 0

Image Examples with Edge Detection



Input Image Format

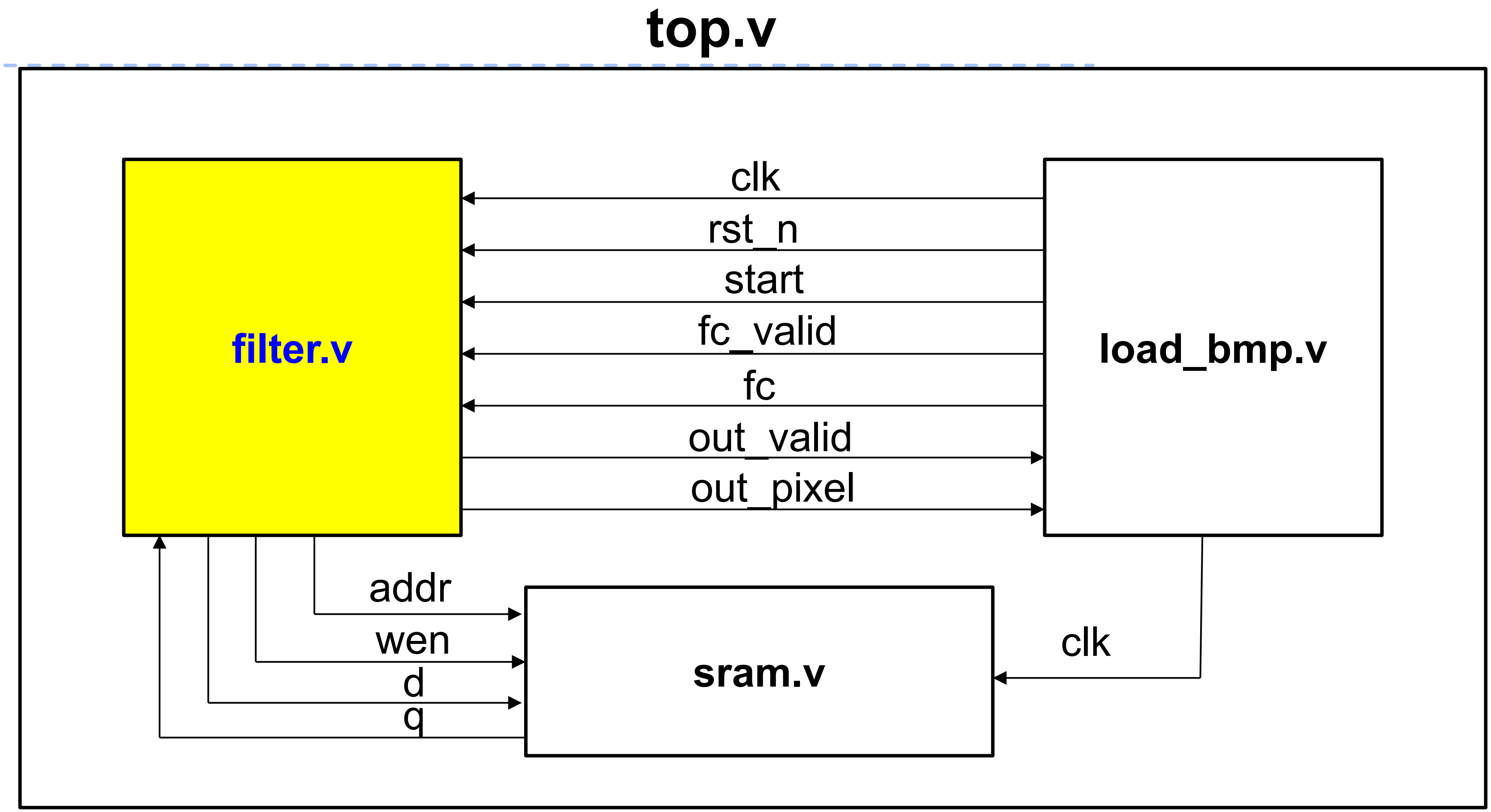
- In a color image, each pixel can be represented as a 24-bit data
  - Three 8-bit unsigned integers (ranging from 0 to 255)
  - RGB encoding



Testbench (load_bmp.v)

- Our testbench will load a color BMP image file, and convert to its gray-scale representation for you
- In addition, the testbench will load the gray-scale pixels into an SRAM block
  - The pixels will be stored in a linear order (row by row)
  - From address 0 to address 65535
  - Signal: start
- Then, it will output the filter coefficients to the filter
  - Signals: fc_valid and fc
- It will accept the processed (65,536) pixels one by one in order, and convert them into the output image
  - Signals: out_valid and out_pixel
- Source code is the best document

Overall Architecture

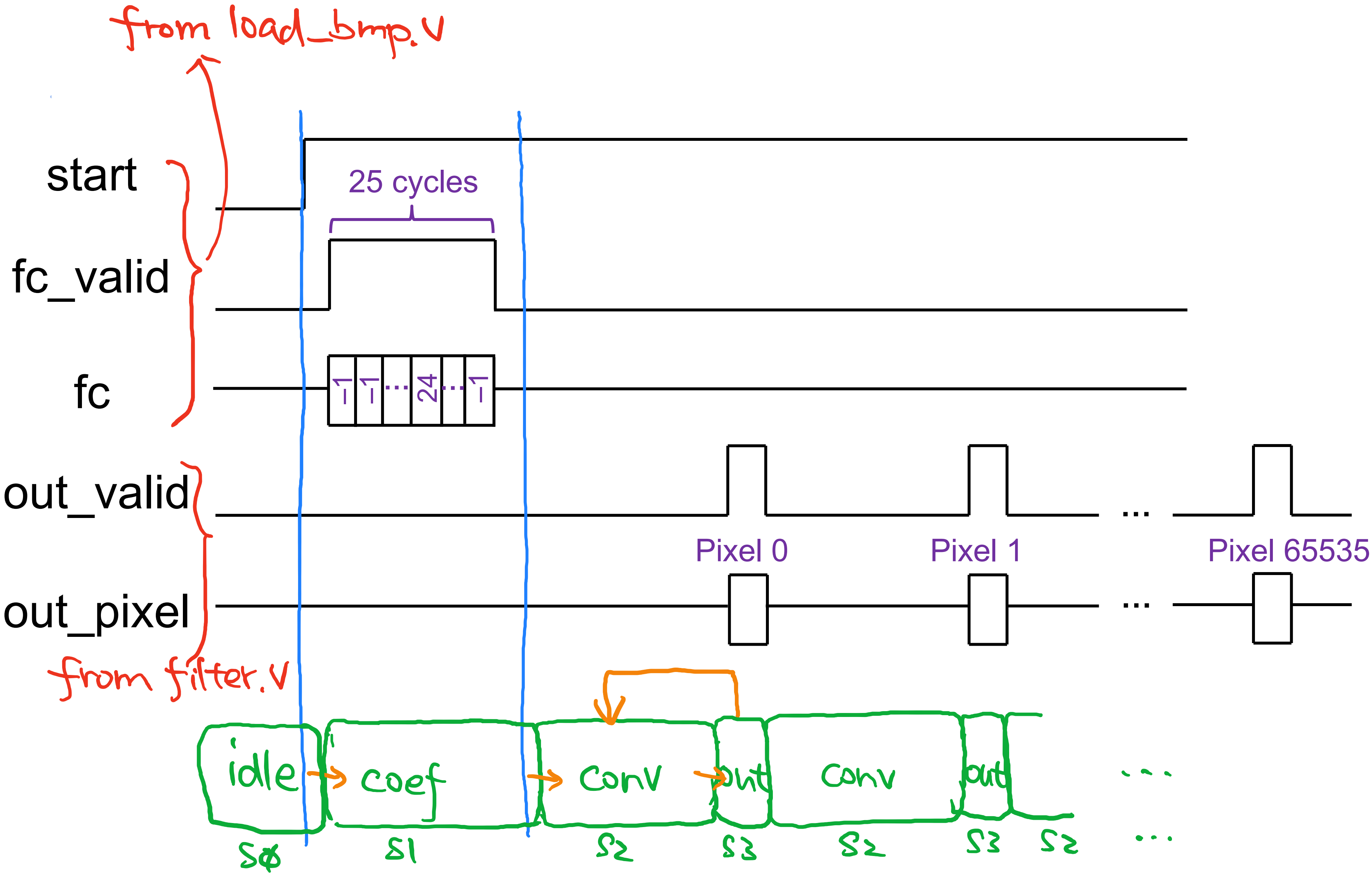


Note:
wen can be constant 1; d can be constant 0, if we don’t write into SRAM at all!

IO Signals

Signal	Explanation
start	When load_bmp.v finishes loading all the pixel value into SRAM, start = 1. Otherwise, start = 0
fc_valid	When load_bmp.v is feeding filter coefficients to filter.v, fc_valid = 1. Otherwise, fc_valid = 0
fc	Filter coefficients (passed from load_bmp.v to filter.v)
out_valid	When the out_pixel is ready, out_valid = 1. Otherwise, out_valid = 0
out_pixel	The pixel value which will be written to output file.
addr	The memory address to read or write
wen	Write enable signal for SRAM
d	Data written to SRAM
q	Data read from SRAM

Timing


from load_bmp.v
start
fc_valid
fc
25 cycles
-1
-1
...
24
...
-1
out_valid
out_pixel
Pixel 0
Pixel 1
...
Pixel 65535
from filter.v
idle
coef
conv
out
conv
out
...
S0
S1
S2
S3
S2
S3
S2
...

Design Files

- **Makefile**
  - Running the simulation easier by typing `make`
  - Also `make clean` for your reference (use it carefully!)
- **top.v**
  - Top module which connects load_bmp.v, sram.v, and filter.v
- **load_bmp.v**
  - Testbench
  - Parsing the input BMP image
  - Feeding the filter coefficients to filter.v
  - Writing out the output BMP image
- **filter.v**
  - The design you are going to implement
- **sram.v**
  - 65536x8 SRAM model
  - Loading the gray-scale pixels into SRAM initially
    - The 2D image is stored in a linear (1D) order
    - Ex: to address the pixel (i, j), you can access the address (i * 256 + j)

Data Files

- **Input image**
  - lena_256x256.bmp
  - einstein_256x256.bmp
  - car_256x256.bmp
- **Golden files (for the output validation)**
  - lena_golden.txt
  - einstein_golden.txt
  - car_golden.txt
- **Gray scale log**
  - img_gray_dec.txt: gray-scale input pixels (decimal)
  - img_gray_hex.txt: gray-scale input pixels (hex)
- **Your output log**
  - out_log.txt (containing all the computed pixel values)
  - Can be compared with golden files for output validation

    ```
    $ diff out_log.txt lena_golden.txt
    ```

Makefile

```
DEBUG = 3
SRC = top.v load_bmp.v sram.v filter.v
BAK = *.bak
LOG = *.log *.key *.fsdb img_gray*.txt out_log.txt *_output.bmp
INCA_libs = INCA_libs
all:
	ncverilog +debug=${DEBUG} ${SRC} +access+r
clean:
	-rm -f ${BAK} ${LOG}
	-rm -rf ${INCA_libs}
```

- A different way from shell script (*.sh) to help the simulation
- Type make to run the simulation
  $ make
  $ make clean
- You may refer to online resources
  - E.g., 鳥哥的Linux私房菜
    http://linux.vbird.org/linux_basic/1010index.php

Discussion

- How many cycles do you need to process the entire image?
- Any chance to improve the performance further?
  - E.g., reducing the number of memory accesses…
- Is it possible to utilize pipeline technique to improve the performance?
- You may also create a second SRAM to store the gray-scale output image if it helps

Hint: Signed Numbers

- Using signed integers may help

9 bits

```
reg signed [8:0] value;
…
   value = $signed({1'b0, gray_scale});
```

8bit unsigned int!

- Make sure you have enough data width to store the inner product

Hint: Possible Concept of Counters

- Filter Coefficients

$(y,x)=(i-2,j-2)$

$(-2,-2)$

	-2	-1	0	1	2
-2	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
0	-1	-1	24	-1	-1
1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1

$(0,0)$

$(+2,-2)$

two counters for 2D filter (m,n)

m: $-2\sim2$, n: $-2\sim2$

- Image Pixels

two counters for 2D image

$(y,x)=(i+0,\ j+0)$

y: $0\sim255$, x: $0\sim255$

$\rightarrow j$

$i \downarrow$

	0	1	2	3	4	… 255
0	Pixel [0]	Pixel [1]	Pixel [2]	Pixel [3]	Pixel [4]	
1	Pixel [256]	Pixel [257]	Pixel [258]	Pixel [259]	Pixel [260]	
2	Pixel [512]	Pixel [513]	Pixel [514]	Pixel [515]	Pixel [516]	
3	Pixel [768]	Pixel [769]	Pixel [770]	Pixel [771]	Pixel [772]	
4	Pixel [1024]	Pixel [1025]	Pixel [1026]	Pixel [1027]	Pixel [1028]	
⋮ 255						

$(i+2,\ j-2)$

$(y,x)=(i+2,\ j+2)$

$(+2,+2)$

$(i+m,\ j+n)$

convert (y,x) to memory address

$addr = y*256+x$

Hint: Some More Suggestions

- You may compute the first few output pixels for the verification before applying for the entire image
- Once again, a detailed planning before Verilog coding is always a good design strategy

And most importantly, enjoy your final project!!