EE271 - Assignment 3 Write-up
Hardware Optimization for the Rasterizer

Khushal Sethi
Gongqi Li

1 Introduction

This assignment aims for optimizing our rasterizer to get a throughput of 2 ns/triangle, i.e., 500 million triangles/second, while minimizing the figure of merit, defined as 'Area * Power * Num_Rasterizer_Units^2'. The default approach we have to guarantee the required throughput is by simply duplicating the rasterizers.

For instance, if a single rasterizer in our design has a throughput of 0.1 triangle/ns, i.e., 100 million triangles/second, we need at least 5 rasterizers working in parallel, which indicates a multiplication of 25 in our figure of merit.

Given the quadratic relationship between FOM and Num_Rasterizer_Units, we tried to minimize the Num_Rasterizer_Units by increasing throughput without sacrificing much on area and power.

Also, as our FOM is not directly related to the frequency of the design, we can adjust the clock period to have the throughput of each rasterizer close to an integer fraction of 500 million triangles/second.

As a result, our design process can be summarized as follows:

1. Several attempts to get a lower area or power like clock gating and lower precision
2. Several attempts to get a higher throughput for each single rasterizer
3. Retiming to have our throughput close to an integer fraction of 500 million triangles/second

2 Approach

1. **Retiming:** For the retiming, on the overall optimization, we get a clock period of 1.33ns after retiming.

2. **Clock Gating:**
   For the flip-flops, clock gating can be used when the bounding box execution is halted. Clock gating is specified directly in the synthesis script with the flag '-gated_clock'.

   Further, we gate the clock with a halt signal for input to the 3-dimensional flip flops of bounding box and triangles to save idle energy consumption. The leakage power for the cycles with the halt signa**l is reduced by 5%, without changing the dynamic power consumption, as expected**. Since, the total power consumption is dominated by the dynamic power, we do not see a high benefit in the total power consumption.

3. **Lower Precision:**
    - We find that multiplication can run at a lower precision of 12 bits. This reduces our power consumption and area of the multiplier implementations. The addition and subtraction operations run at 24 bits.

    - Since, checking each sample is a hit or not, requires on the least significant 12 bits of the operation, this gives significant reduction in power consumption and area consumption.

    - Over the 24-bit implementation of multipliers the dynamic power, static power and area consumption are **reduced by a factor of 2, as expected**.

4. **Backface Culling**
    - **Backface Culling moves the step of judging the directionality of the triangle earlier, and at an earlier stage, deactivates counter-clockwise triangles as any further calculations related to them are useless anyway.**

    - The previous design deactivated these triangles during the 'sample test' module, resulting in the unnecessary processing of many sample points.

    - In our design, we judge the directionality of the triangle at the bbox stage, where the validTri signal is set to false if the triangle is back-facing and the signal is then transmitted to the test_iterators stage, preventing any sampling with this invalid triangle, saving the amount of cycles proportional to the number of pixels within backface triangles.

    - For vectors with **high back - to - forward triangle ratio (especially vect_02), the improvement over benchmark can be significant (over 100%)**.

    The results of Backface Culling is as followed (vect_02_short):

    Cycles: 134065
    Triangle / Cycle: 0.074598
    Cycle / Triangle: 13.405159

5. **Bubble Smashing**
    - In the test_iterator stage, when the state is TEST_STATE, all the pipeline stages in bbox are halted. But as Backface Culling produced many invalid triangles, these triangles are stacked in bbox without being processed. **With bubble smashing, we want to unhalt the bbox pipeline to flush all the invalid triangles out of the pipeline.**

- Qualitatively the marginal improvement from Bubble Smashing is approximately equal to the number of cycles wasted by those invalid triangles and thus proportional to the number of invalid triangles.
- This improvement is not expected to be large since each invalid triangle only takes one additional clock period if not being flushed by Bubble Smashing.

- **It should be noted that for vect_00, as we have exceptionally large triangles, an error arises in the testbench and we have to bump the pipe clean cycle duration in verif/rast_driver.sv from 10 to a very large value, in our case 10000, to ensure not missing any hitting from vect_00, and this treatment is not applied to other test vectors.**

The improvements of Bubble Smashing is shown as followed (vect_02_short):

Cycles: 133310
Triangle / Cycle: 0.075013
Cycle / Triangle: 13.331000

6. Multitest

The logic with Multitest is straightforward. We can test multiple samples in parallel in **Sampletest** stage instead of only one sample per cycle. This method is sure to increase out throughput, so the **key point here is to analyze whether it is a more efficient (in terms of power and area) method compared with simple duplication.**

- Let us assume here that we test 2 samples in **Sampletest** stage in parallel, which means that the **Sampletest** stage takes as input a triangle to be tested and 2 contiguous sample locations.

- If the t**riangle is very large,** containing hundreds of sample locations, it is easy to see that the cycles needed for sample testing is halved. And the overall throughput is increased by a factor smaller than 2 since the previous pipeline stages do not benefit from Multitest.

- If instead the **triangle is rather small**, take an extreme case that the triangle only has three sample locations, then the cycles needed for sample testing only reduces from 3 to 2, i.e., 33%, and the overall throughput enhancement is much smaller than 2.

- The tradeoff for Multitest is that our area and power for **Sampletest** stage and **Hash** stage nearly get doubled. In contrast, for simple duplication, the throughput, area and power all get doubled.

We thus observe that in our case**, Multitest it is not an effective design: the triangles are rather small with, for example, 2.7 sampleHits per valid triangle for**

vect_02_short. As a result, the **benefit from Multitest is not worthy of its cost on additional area and power, and thus not a good choice compared with simple duplication**.

7. **New FSM**
    - Our optimization of the FSM is based on the fact that in the original design, after the processing of the previous valid triangle and we got an input of a new validTri signal, the next_state signal is then set to TEST_STATE, which means that the new valid triangle starts to be processed one clock period afterwards.

    - This, as a result, brings us an additional required cycle for each valid triangle coming after our TEST_STATE. In our design, we make sure that the FSM process the valid triangle immediately after it receives the signal without an intermediate WAIT_STATE.

    This new improvement enhanced our throughput by about 5% as followed:

    Cycles: 128315
    Triangle / Cycle: 0.077933
    Cycle / Triangle: 12.831500

8. **Better Bounding Box**
    - Our original design defined the bounding box to be a rectangle with its lower left corner and upper right corner specified by the minimum and maximum coordinates of the vertices of the triangle, and in test_iterator stage, sweep the samples within the bounding box by incrementing along X axis (Y axis if at the right edge of the bounding box).

    - This produces inefficiency as we know only one-third of the area within the bounding box actually lies in the triangle, indicating a potential waste of cycles when iterating over the rest two-third of the pixels.

    - Alternatively, we can just sweep through all of the pixels within the region constrained by the three edge functions of the triangle. Ideally, this method can save the number of iterations by two-third for very large triangles containing many samples.

    - However, the effectiveness of this algorithm decreases for smaller triangles. To illustrate this, consider a triangle which generates a 2x2 bounding box (1x MSAA means 2 samples). In this case, no matter how we sweep, we have to iterate over all of the four samples.

- Taking into account that our test vectors typically only have several hits per valid triangle, for example for vect_02, we have 0.46mn triangles and 1.35mn sampleHits, we are not expecting this algorithm to make any significant improvements and thus have not implemented it.

This concludes our design process, and we summarize the progress so far in throughput as follows:

Triangle / Cycle		
Vector	Baseline	Backface Culling + Bubble Smashing + FSM
vect_01_short	0.042490	0.045323
vect_01	0.033	0.074744
vect_02_short	0.033697	0.077933
vect_02	0.037123	0.074140
vect_03_short	0.040861	0.062165

Note: vect_00 is not included, as **verif/rast_driver.sv** need to be modified as suggested previously, and this setting is not applied to the other test vectors.

3 Results

Overall Optimization:
The figure of merit is calculated in the following units =
(No. of Rasterizer Units)^2* Power(in mW) * Area(in mm^2)

Approach	Throughput (Tri / cycle)	Clk (ns)	Dynamic Power (mW)	Leakage Power (uW)	Area (um^2)	Num of Rasterizers	Total Power (mW)	FOM
Baseline	0.042490	1.2	22.1050	776.577	41029	15	22.882	207.7
Clock Gating	0.042490	1.2	22.232	742.973	39936	15	22.976	206.2
Lower Precision	0.043236	1.2	11.718	355.750	19713	14	12.073	46.6
Backface Culling	0.074598	1.2	18.369	422.729	22727	9	18.760	34.4
Bubble Smashing	0.075013	1.2	22.201	421.218	22694	8	22.623	32.7
Retimed Bubble Smashing	0.075013	1.5	16.264	407.570	21888	10	16.672	36.4

New FSM	**0.077933**	**1.3**	**19.567**	**407.130**	**21683**	**9**	**19.974**	**34.9**
Overall Optimization (with retiming)	**0.077933**	**1.33**	**17.930**	**393.090**	**21248**	**9**	**18.324**	**31.5**