Table of Content

Architecture 2

1 Overview 2

2 Register 3

3 ISA Summary 3

1.1 3 Operand operations. 3

1.2 Immediate mode 4

4 Flags & Branch 5

5 ISA detail 6

6 Pseudo Instructions 21

7 Memory Map 21

Micro-architecture 22

1 Overview 22

2 Memory Controller 23

3 Command Patcher 25

4 Patch Dispatcher 27

5 RT core 30

6 IC buffer 32

7 IC core 34

8 IP cores 36

9 Memory 37

9.1 Memory overview 37

9.2 Instruction memory 37

9.3 Constant memory 38

9.4 Main memory 39

9.5 Triangle memory 40

10 Coding standard 42

Architecture

1 Overview

Our processing unit is a parallel-processing unit that mimics the behavior of current GPU. From a software perspective, the same instructions will be executed for each pixel. Each thread will have a pixel ID showing its location on the final image. It will also be assigned a thread ID, which is between 31-0. This is the ID within each patch. The hardware will keep track of the pixel ID of each thread and place it in the register file before the thread starts. Each thread will use this ID to compute which pixel this thread is working on and find the correct camera ray. The PC and the stack-base pointer will also be prepared by the hardware and placed in register file.

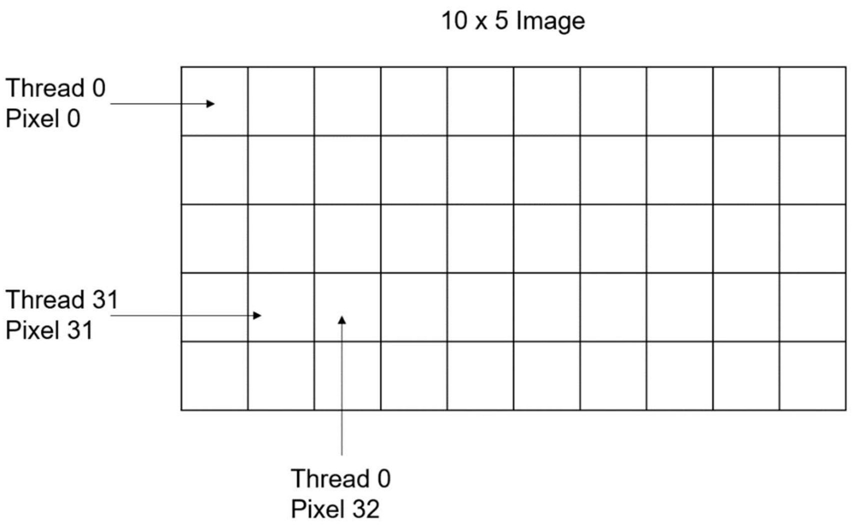

Figure 1 Relationship between Thread ID and Pixel ID

Each thread can fully utilize any resources on its RT core. It has its own PC (we are no longer doing lock step), register file, integer compute unit, floating point compute unit, as well as memory access. We have two types of memory: constant memory, and local memory. Constant memory defines the global state, like camera location, direction, material information. In Computer Graphics terminology, this is the “Uniform”. All constant memory address is the same for each thread. The other type of memory is local memory, and it is used for stack and output. Each thread has its own memory space. Software can access these memories freely as there is only one thread, (since we support multiple reads performed at the same time). Our hardware will make sure this thread is accessing its own memory space. Constant memory is four-byte addressable, and it only supports scalar read. Main Memory is four-byte addressable and support scalar vector read write. Both memories use little endian formatting.

To access IC core accelerator, a thread evokes a special instruction called “trace”. Before calling trace, it needs to put the function call parameters in special locations in vector registers. When this instruction is executed, it will halt the thread and perform a hardware context switch. Hardware will store the current PC and the stack pointer of the thread, and then will read from those vector registers. Then hardware will give this information to the IC core and schedule a different thread to this core. When the IC core is finished and hardware is ready to schedule this thread back, it will put the trace result, PC and stack pointer back to the register file before it executes. If software needs to preserve other registers, these registers must be pushed to the stack. Finally, each thread can return one vector back to the host as the pixel color. They need to write the return value to the last vector of their main memory space.

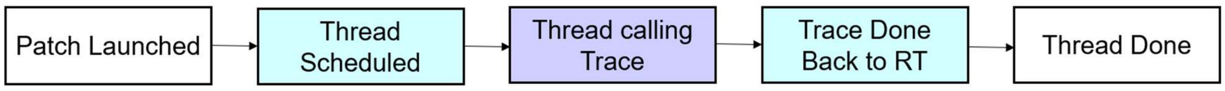

Figure 2 Typical Thread execution cycle

After careful deliberations, we decided to remove command processor and replace it with a state machine. Therefore, its behavior is fixed and only the RT core needs instructions.

2 Register

Each thread has 32 32-bit scalar registers from R0-R31. The R31 register is reserved for PC register. R0 is reserved for the value zero and it is read-only. R30 is reserved for link register. R29 is reserved for stack pointer and R28 is reserved for stack base pointer. We also provide 16 4x32-bit vector registers, RV0-RV15. Similarly, R0 is reserved for the value zero and it is read-only.

3 ISA Summary

We have specially designed the ISA to reduce the number of multiplexors in our decode process.

1.1 3 Operand operations.

All three operand operations do not have an immediate value and have a 0 for their fist opcode bit.

Next 3 bits: Operand type

000: Vector * Vector -> Vector

010: Vector * Scalar -> Vector

011: Scalar * Scalar -> Scalar (And Or Xor Not)

110: Scalar * Scalar -> Scalar (Int)

111: Scalar * Scalar -> Scalar (Int) Update Flag

100: Scalar * Scalar -> Scalar (Float)

101: Scalar * Scalar -> Scalar (Float) Update Flag

Next 2 bits: operation

00: Add (And)

01: Sub (Or)

10: Mul (Xor)

11: Div (Not)

Opcode [31: 26]	Instruction	Explanation
000000	vv_add	Add two vectors (vec4_float)
000001	vv_sub	Subtract two vectors (vec4_float)
000010	vv_mul_ele	Elementwise multiply two vectors (vec4_float)

000011	vv_div	Divide two vectors (vec4_float)
001000	vf_add	Add a scaler/float to a vector (vec4_float)
001001	vf_sub	Subtract a scaler/float from a vector (vec4_float)
001010	vf_mul	Multiply a scaler/float to a vector (vec4_float)
001011	vf_div	Divide a scaler/float from a vector (vec4_float)
001100	not	Bitwise not
001101	and	Bitwise and
001110	or	Bitwise or
001111	xor	Bitwise xor
011000	ii_add	Add two integers
011001	ii_sub	Subtract an integer by another integer
011010	ii_mul	Multiply two integers
011011	ii_div	Divide an integer by another integer
011101	cmp_i	Compare two integer scalars and output to flag register
010000	ff_add	Add two floats
010001	ff_sub	Subtract a float by another float
010010	ff_mul	Multiply two floats
010011	ff_div	Divide a float by another float
010101	cmp_f	Compare two float scalars and output to flag register

1.2 Immediate mode

The last 16 bits of these instructions are either immediate values or "don't care" values. These opcodes start with 1.

- Next bits 0: This instruction has writeback.
  - Next bits 00: Integer core special function
  - Next bits 01: floating (vector) core special function
  - Next bits 10: Integer core operation with immediate
  - Next bits 11: memory load
- Next bits 1: This instruction does not have writeback.
  - Next bits 00: Jump and branch.
  - Next bits 10: Program control
  - Next bits 11: memory store

Opcode[31: 26]	**Instruction**	**Explanation**
100000	s_write_high	Write immediate into higher 16 bits of the register
100001	s_write_low	Write immediate into lower 16 bits of the register
100010	s_itof	Convert integer to float
100011	s_ftoi	Convert float to integer
100100	v_reduce	Add all elements together

100101	s_get_from_v	Read one scalar from a vector into a register
100110	s_sqrt	Calculate the square root of the scalar (float)
100111	v_get_from_s	Put one scalar into a location of the vector register
101000	ii_addi	Add an integer immediate to an integer
101001	ii_subi	Subtract an integer by an integer immediate
101010	ii_muli	Multiply an integer immediate and an integer
101011	ii_divi	Divide an integer by an integer immediate
101100	s_load_4byte	Load four bytes of scalar data from memory
101111	v_load_16byte	Load vector (four four-byte dimension) from memory
111100	s_store_4byte	Store four bytes of scalar data into memory
111111	v_store_16byte	Store vector (four four-byte dimension) into memory
111010	Trace	Start a trace command
111011	Fin	Program ends
110000	be	Branch if equal
110001	bne	Branch if not equal
110010	bl	Branch If less than
110011	bge	Branch if greater or equal
110100	bg	Branch if greater than
110101	ble	Branch if less or equal
110110	jmp	Jump to a label
110111	ret	Return to a stored address

4 Flags & Branch

Flag status is stored in a 2-bit register during the execution stage by only the “compare” instructions (cmp_i, cmp_f). Therefore, software needs to compare first and then evoke the branch instruction. Software developers are advised to avoid checking the zero flag after the cmp_f command. This is because there might be some rounding error due to floating points. We advise software developers to use the following equation to check if two floats are the same.

$$|Op_1 - OP_2| < \varepsilon, \varepsilon = 0.001$$

Value	Abbreviation	Description
00	Z	Zero
11	N	Negative
10	P	Positive

5 ISA detail

5.1 vv_add

Syntax:

vv_add <RVdst>, <RVa>, <RVb>

Pseudo Code: (Can be in C/Verilog)

RVdst = RVa + RVb

Encoding:

000000 xdddd xaaaa xbbbb xxx xxxx xxxx

Usage and Examples:

Add two 4x4byte vectors

vv_add RV3, RV1, RV2

5.2 vv_sub

Syntax:

vv_sub <RVdst>, <RVa>, <RVb>

Pseudo Code: (Can be in C/Verilog)

RVdst = RVa - RVb

Encoding:

000001 xdddd xaaaa xbbbb xxx xxxx xxxx

Usage and Examples:

Subtract two 4x4byte vectors

vv_sub RV3, RV1, RV2

5.3 vv_mul_ele

Syntax:

vv_mul_ele <RVdst>, <RVa>, <RVb>

Pseudo Code: (Can be in C/Verilog)

```
RVdst = RVa .* RVb
RVdst[0] = RVa[0] * RVb[0]
RVdst[1] = RVa[1] * RVb[1]
RVdst[2] = RVa[2] * RVb[2]
RVdst[3] = RVa[3] * RVb[3]
```

Encoding:

000010 xdddd xaaaa xbbbb xxx xxxx xxxx

Usage and Examples:q

Elementwise multiply two 4x4byte vectors

vv_mul_ele RV3, RV1, RV2

5.4 vv_div

Syntax:

vv_div <RVdst>, <RVa>, <RVb>

Pseudo Code: (Can be in C/Verilog)

RVdst = RVa./RVb

RVdst[0] = RVa[0] / RVb[0]

RVdst[1] = RVa[1] / RVb[1]

RVdst[2] = RVa[2] / RVb[2]

RVdst[3] = RVa[3] / RVb[3]

Encoding:

000011 xdddd xaaaa xbbbb xxx xxxx xxxx

Usage and Examples:

Elementwise divide two 4x4byte vectors

vv_div RV3, RV1, RV2

5.5 vf_add

Syntax:

vf_add <RVdst>, <RVa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RVdst = RVa + Rsb

RVdst[0] = RVa[0] + Rsb

RVdst[1] = RVa[1] + Rsb

RVdst[2] = RVa[2] + Rsb

RVdst[3] = RVa[3] + Rsb

Encoding:

001000 xdddd xaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Add 4x4byte vector with a 4byte scaler float

vf_add RV1, RV2, R1

5.6 vf_sub

Syntax:

vf_sub <RVdst>, <RVa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RVdst = RVa - Rsb

RVdst[0] = RVa[0] - Rsb

RVdst[1] = RVa[1] - Rsb

RVdst[2] = RVa[2] - Rsb

RVdst[3] = RVa[3] - Rsb

Encoding:

001001 xdddd xaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Subtract 4byte scaler float from 4x4byte vector

vf_sub RV1, RV2, R1

5.7 vf_mul

Syntax:

vf_mul <RVdst>, <RVa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

```
RVdst = RVa * Rsb
RVdst[0] = RVa[0] * Rsb
RVdst[1] = RVa[1] * Rsb
RVdst[2] = RVa[2] * Rsb
RVdst[3] = RVa[3] * Rsb
```

Encoding:

001010 xdddd xaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Multiply 4x4byte vector with a 4byte scaler float

vf_mul RV1, RV2, R1

5.8 vf_div

Syntax:

vf_div <RVdst>, <RVa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

```
RVdst = RVa / Rsb
RVdst[0] = RVa[0] / Rsb
RVdst[1] = RVa[1] / Rsb
RVdst[2] = RVa[2] / Rsb
RVdst[3] = RVa[3] / Rsb
```

Encoding:

001011 xdddd xaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Divide 4byte scaler float from 4x4byte vector

vf_div RV1, RV2, R1

5.9 not

Syntax:

not <RSd>, <RSs>

Pseudo Code: (Can be in C/Verilog)

```
RSd = ~RSs;
```

Encoding:

001100 ddddd sssss xxxx xxxx xxxx xxxx

Usage and Examples:

Inverse the data in the source register in scalar register file and store the result into the destination register.

not R2, R1

Inverse the data in R1 and store the result into R2.

5.10 and

Syntax:

and <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa and RSb;

Encoding:

001101 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

The conjunction of two scalar registers and store the result into the destination register.

and R3, R1, R2

R1 and R2. Store the result into R3.

5.11 or

Syntax:

or <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa or RSb;

Encoding:

001110 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

The disjunction of two scalar registers and store the result into the destination register.

or R3, R1, R2

R1 or R2. Store the result into R3.

5.12 xor

Syntax:

xor <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa xor RSb;

Encoding:

001111 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

The exclusive disjunction of two scalar registers and store the result into the destination register.

xor R3, R1, R2

R1 xor R2. Store the result into R3.

5.13 ii_add

Syntax:

ii_add <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa + RSb;

Encoding:

011000 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Add two integers from the scalar register file and store the result into the destination register.

ii_add RS3, RS1, RS2

Add RS1 and RS2. Then, store the result to RS3.

5.14 ii_sub

Syntax:

ii_sub <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa - RSb;

Encoding:

011001 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Subtract one integer from another integer, which both data are from register file. Then, store the result into the destination register.

ii_sub RS3, RS1, RS2

Subtract RS2 from RS1. Then, store the result to RS3.

5.15 ii_mul

Syntax:

ii_mul <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa * RSb;

Encoding:

011010 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Multiply two integers from the scalar register file and store the result into the destination register.

ii_mul RS3, RS1, RS2

Multiply RS1 and RS2. Then, store the result to RS3.

5.16 ii_div

Syntax:

ii_div <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa / RSb;

Encoding:

011011 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Divide one integer by another integer, which both data are from register file. Then, store the result into the destination register.

ii_div RS3, RS1, RS2

Divide RS1 by RS2. Then, store the result to RS3.

5.17 cmp_i

Syntax:

Cmp_i <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

Flag_reg = flag(RSa - RSb);

Encoding:

011101 xxxxx aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Compare two integer scalars by computing (RSa – RSb) and output flag to flag register of corresponding thread.

Cmp_I R1, R2

Compare the integer values in R1 and R2 by computing (R1 - R2) and output flag to flag register of corresponding thread.

5.18 ff_add

Syntax:

ff_add <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa + RSb;

Encoding:

010000 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Add two floats from the scalar register file and store the result into the destination register.

ff_add RS3, RS1, RS2

Add RS1 and RS2. Then, store the result to RS3.

5.19 ff_sub

Syntax:

ff_sub <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa - RSb;

Encoding:

010001 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Subtract one float from another float, which both data are from register file. Then, store the result into the destination register.

ff_sub RS3, RS1, RS2

Subtract RS2 from RS1. Then, store the result to RS3.

5.20 ff_mul

Syntax:

ff_mul <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa * RSb;

Encoding:

010010 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Multiply two floats from the scalar register file and store the result into the destination register.

ff_mul RS3, RS1, RS2

Multiply RS1 and RS2. Then, store the result to RS3.

5.21 ff_div

Syntax:

ff_div <RSd>, <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

RSd = RSa / RSb;

Encoding:

010011 ddddd aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Divide one float by another float, which both data are from register file. Then, store the result into the destination register.

ff_div RS3, RS1, RS2

Divide RS1 by RS2. Then, store the result to RS3.

5.22 cmp_f

Syntax:

Cmp_f <RSa>, <RSb>

Pseudo Code: (Can be in C/Verilog)

Flag_reg = flag(RSa - RSb);

Encoding:

010101 xxxxxx aaaaa bbbbb xxx xxxx xxxx

Usage and Examples:

Compare two float scalars by computing (RS1 – RSb) and output flag to flag register of corresponding thread.

Cmp_f R1, R2

Compare the float values in R1 and R2 by computing (R1 - R2) and output flag to flag register of corresponding thread.

5.23 s_write_high

Syntax:

s_write_high <RSdst>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

RSdst[31:16] = imm[15:0]

Encoding:

100000 ddddd ddddd iiii iiii iiii iiii

Usage and Examples:

Fill the highest 2 byte of the scaler register

s_write_high R2, #-24

5.24 s_write_low

Syntax:

s_write_low <RSdst>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

RSdst[15:0] = imm[15:0]

Encoding:

100001 ddddd ddddd iiii iiii iiii iiii

Usage and Examples:

Fill the lowest 2 byte of the scaler register

s_write_low R2, #24

5.25 itof

Syntax:

itof <RSd>, <RSs>

Pseudo Code: (Can be in C/Verilog)

RSd = float(RSs);

Encoding:

100010 ddddd sssss xxxx xxxx xxxx xxxx

Usage and Examples:

Convert an integer from the source scalar register to a float and store the result into destination scalar register.

itof R2, R1

Convert the integer in R2 to float and store it into R2.

5.26 ftoi

Syntax:

itof <RSd>, <RSs>

Pseudo Code: (Can be in C/Verilog)

RSd = int(RSs);

Encoding:

100011 ddddd sssss xxxx xxxx xxxx xxxx

Usage and Examples:

Convert a float from the source scalar register to an integer and store the result into destination scalar register.

itof R2, R1

Convert the float in R2 to integer and store it into R2.

5.27 V_reduce

Syntax:

vv_mul_dot <RSdst>, <RVsrc>

Pseudo Code: (Can be in C/Verilog)

RSdst = RVsrc [0] + RVsrc [1] + RVsrc [2] + RVsrc [3]

Encoding:

100100 ddddd xaaaa xxxx xxxx xxxx xxxx

Usage and Examples:

Add four element in a vector together

V_reduce R3, RV1

5.28 s_get_from_v

Syntax:

s_get_from_v <RSdst>, <RVsrc>, #<index> {0,1,2,3}

Pseudo Code: (Can be in C/Verilog)

R2 = RV1[index]

Encoding:

100101 ddddd xssss xx xxxx xxxx xxxx ii

Usage and Examples:

Get a 4byte scaler from a specified index of a 4x4byte vector

s_get_from_v R2, RV1, #2

5.29 S_sqrt

Syntax:

s_sqrt <RSdst>, <RSsrc>

Pseudo Code: (Can be in C/Verilog)

RSdst = sqrt(RSsrc)

Encoding:

100110 ddddd sssss xxxx xxxx xxxx xxxx

Usage and Examples:

Find square root of a floating point number.

s_sqrt R2, R1

5.30 v_get_from_s

Syntax:

v_get_from_s <RVdst>, <RSsrc>, #<index> {0,1,2,3}

Pseudo Code: (Can be in C/Verilog)

RV1[index] = R2

Encoding:

100111 xdddd sssss xx xxxx xxxx xxxx ii

Usage and Examples:

Put a 4byte scaler to a specified index of a 4x4byte vector

v_get_from_s RV1, RV2, #2

5.31 ii_addi

Syntax:

ii_addi <RSd>, <RSs>, imm

Pseudo Code: (Can be in C/Verilog)

RSd = RSs + imm

Encoding:

101000 ddddd sssss iiii iiii iiii iiii

Usage and Examples:

Add an integer immediate to an integer from source register in scalar register file. Then, store the result into the destination register.

ii_addi R2, R1, #7

Add 7 to R1. Then, store the result into R2.

5.32 ii_subi

Syntax:

ii_subi <RSd>, <RSs>, imm

Pseudo Code: (Can be in C/Verilog)

RSd = RSs - imm

Encoding:

101001 ddddd sssss iiii iiii iiii iiii

Usage and Examples:

Subtract an integer immediate from an integer from source register in scalar register file. Then, store the result into the destination register.

ii_subi R2, R1, #7

Subtract 7 from R1. Then, store the result into R2.

5.33 ii_muli

Syntax:

ii_muli <RSd>, <RSs>, imm

Pseudo Code: (Can be in C/Verilog)

RSd = RSs * imm

Encoding:

101010 ddddd sssss iiii iiii iiii iiii

Usage and Examples:

Multiply an integer immediate to an integer from source register in scalar register file. Then, store the result into the destination register.

ii_muli R2, R1, #7

Multiply 7 to R1. Then, store the result into R2.

5.34 ii_divi

Syntax:

ii_divi <RSd>, <RSs>, imm

Pseudo Code: (Can be in C/Verilog)

RSd = RSs / imm

Encoding:

101011 ddddd sssss iiii iiii iiii iiii

Usage and Examples:

Divide an integer from the source register in scalar register file by an integer immediate. Then, store the result into the destination register.

ii_divi R2, R1, #7

Divide R1 by 7. Then, store the result into R2.

5.35 s_load_4byte

Syntax:

s_load_4byte < RSdst >, <RSaddress>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

RSdst = *mem[RSaddress+imm]*

Encoding:

101100 ddddd aaaaa iiii iiii iiii iiii

Usage and Examples:

Load a 32bit scaler from the address+imm

s_load_4byte R2, R1, #7

5.36 v_load_16byte

Syntax:

v_load_16byte < RVdst >, <RVaddress>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

RVdst = *mem[RVaddress+imm]*

Encoding:

101111 xdddd aaaaa iiii iiii iiii iiii

Usage and Examples:

Load a 4x32bit vector from the address+imm

v_load_16byte RV2, R1, #7

5.37 s_store_4byte

Syntax:

s_store_4byte <RSsrc>, <RSaddress>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

mem[RSaddress+imm] = RSsrc

Encoding:

111100 sssss aaaaa iiii iiii iiii iiii

Usage and Examples:

Store a 32bits scaler to the memory address+imm

s_store_4byte R2, R1, #7

5.38 v_store_16byte

Syntax:

v_store_16byte <RVsrc>, <RVaddress>, #<immediate>

Pseudo Code: (Can be in C/Verilog)

mem[RVaddress+imm] = RVsrc

Encoding:

111111 xssss aaaaa iiii iiii iiii iiii

Usage and Examples:

Store a 4x32bits vector to the memory address+imm

v_store_16byte RV2, R1, #7

5.39 trace

Syntax:

trace

Pseudo Code: (Can be in C/Verilog)

Trace: a hardware function call that the argument should be in special registers and returned value will also be in the special registers. Patch Dispatcher will do context switch after calling this function.

Calling: Origin + Direction (retrieve value automatically from RV14 and RV15)

Return: Hit point (X, Y, Z, shader-ID int) + Normal (return automatically to RV14 and RV15)

Encoding:

111010 xxxxx xxxxx xxxx xxxx xxxx xxxx

Usage and Examples:

Trace a ray from its origin along its direction, which are from two different vector registers, to find intersection with triangles.

Trace

5.40 Fin

Syntax:

FIN

Encoding:

111011 xxxxx xxxxx xxxx xxxx xxxx xxxx

Usage and Examples:

Signal the current program has finished.

5.41 be

Syntax:

Be LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

If (flag_reg == Z)

Program counter <= PC+LABEL;

Encoding:

110000 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is Z

Bne FUNCTION_A

5.42 bne

Syntax:

Bne LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

```
If (flag_reg != Z)
Program counter <= PC+LABEL;
```

Encoding:

110001 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is not Z.

Bne FUNCTION_A

5.43 bl

Syntax:

Bl LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

```
If (flag_reg == N)
Program counter <= PC+LABEL;
```

Encoding:

110010 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is N, indicating the previous comparison result RSa < RSb.

Bl FUNCTION_A

5.44 bge

Syntax:

Bge LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

```
If (flag_reg == (P || Z))
Program counter <= PC+LABEL
```

Encoding:

110011 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is P or Z, indicating the previous comparison result RSa >= RSb.

Bge FUNCTION_A

5.45 bg

Syntax:

Bg LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

If (flag_reg == P)

Program counter <= PC+LABEL;

Encoding:

110100 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is P, indicating the previous comparison result RSa > RSb.

Bg FUNCTION_A

5.46 ble

Syntax:

Ble LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog)

If (flag_reg == (N || Z))

Program counter <= PC + LABEL;

Encoding:

110101 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate if the flag register of corresponding thread is N or Z, indicating the previous comparison result RSa <= RSb.

Ble FUNCTION_A

5.47 jmp

Syntax:

jmp LABEL (#<immediate_16>)

Pseudo Code: (Can be in C/Verilog):

Program counter <= PC+LABEL;

Encoding:

110110 xxxxx xxxxx iiii iiii iiii iiii

Usage and Examples:

Jump to the label address represented by the immediate by setting the program counter to it.

Jmp FUNCTION_A

5.48 return

Syntax:

Ret

Pseudo Code: (Can be in C/Verilog)

Program counter <= R30;

Encoding:

110111 xxxxx 11110 xxxx xxxx xxxx xxxx

Usage and Examples:

Return subroutine call by setting the program counter to the address saved in R30 by jmp_link

6 Pseudo Instructions

These instructions can be understood by Assembler, but they are not real instructions that can be execute by the RT core.

Instructions	Real instructions
S_pop <RSdst>	s_load_4byte <RSdst> <R29>, ii_addi <R29>, <R29>, -4
S_push <RSsrc>	s_store_4byte < RSsrc> <R29>, ii_addi <R29>, <R29>, 4
V_pop <RVdst>	v_load_16byte <RVdst> <R29>, ii_addi <R29>, <R29>, -16
V_push <RVsrc>:	v_store_4byte < RVsrc> <R29>, ii_addi <R29>, <R29>, 16
jmp_link <immediate_16>	ii_addi R30 R31 4, real_jmp $l
s_setzero <RSdst>	xor <RSdst> <RSdst> <RSdst>
s_mov <RSdst> <RSsrc>	ii_add <RSdst> <RSsrc> R0
v_mov <RVdst> <RVsrc>	vv_add <RVdst> <RVsrc> R0

7 Memory Map

From a software point of view, there are two memories: constant and main memory. Other memory addresses are invalid, and they will not return anything.

Constant memory space: 0x2000000 - 0x20007FF. The constant memory will stay the same no matter the thread ID. The content is generated by software developer and it is the software developers' responsibility to make sure their software and constants agree with each other.

Main memory space: 0x80(Thread ID)0000 - 0x80(Thread ID)FFFF. Each thread will have its own main memory space. We advise software developer to use this memory space as a stack to store local variables and make function calls. At the start of each thread. 0x80(Thread ID)0000 will be placed into R29 automatically so software developer can start push and pop immediately.

Return memory space: 0x80(Thread ID)FFF0 - 0x80(Thread ID)FFFF: Before the thread calling Fin, it needs to write the returned value (typically a vector) to this address. Then the hardware will automatically write the value back to the host.

Micro-architecture

1 Overview

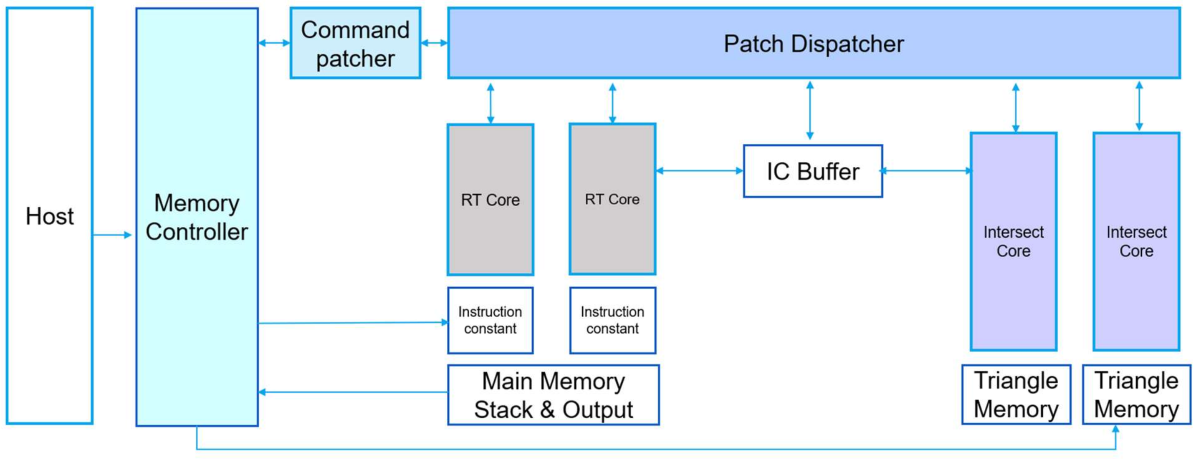

Figure 3 Block diagram of top-level

Our hardware aims to supports all the functionality we defined in the Architecture Chapter. Host and self-made memory controller work together through DMA and MMIO interface to move instructions and triangles from host to chip and move the chip to the host.

Command Patcher is a simplified version of the Command Processor we planned. Host will tell the Command Patcher the total number of Pixels and it will break them into a 32-thread chunk. Then it will assign this chunk to Patch Dispatcher. It will also communicate with host when a patch has completed and move the output back to the host.

Patch dispatcher is responsible for scheduling 32 thread to 4 RT Core and 8 IC Core. When a thread evoke trace, it will push this thread to a queue and assign a new thread to this RT Core. It interacts with IC Core in a similar fashion. When all 32 threads have completed, it will signal the Command Patcher that it is ready for the next 32 threads.

RT Core is a generic CPU core with vector and floating-point support and the intersection core is a hardware accelerator. It utilizes the Möller-Trumbore algorithm to achieve a faster execution. The RT Core passes the ray information to IC Buffer. The intersection cores fetch triangle information from triangle memory. A pipeline is used to reduce the down time of the hardware, utilizing the advantages of the algorithm.

Since the Instruction memory, triangle memory and the constant memory is read-only to all the thread, we can simply duplicate them to avoid bank conflict. Each RT Core has its own Instruction and Constant memory, and each IC Core has its own Triangle memory. However, RT Core can read-write main memory therefore we have a unified main memory for all the RT Core.

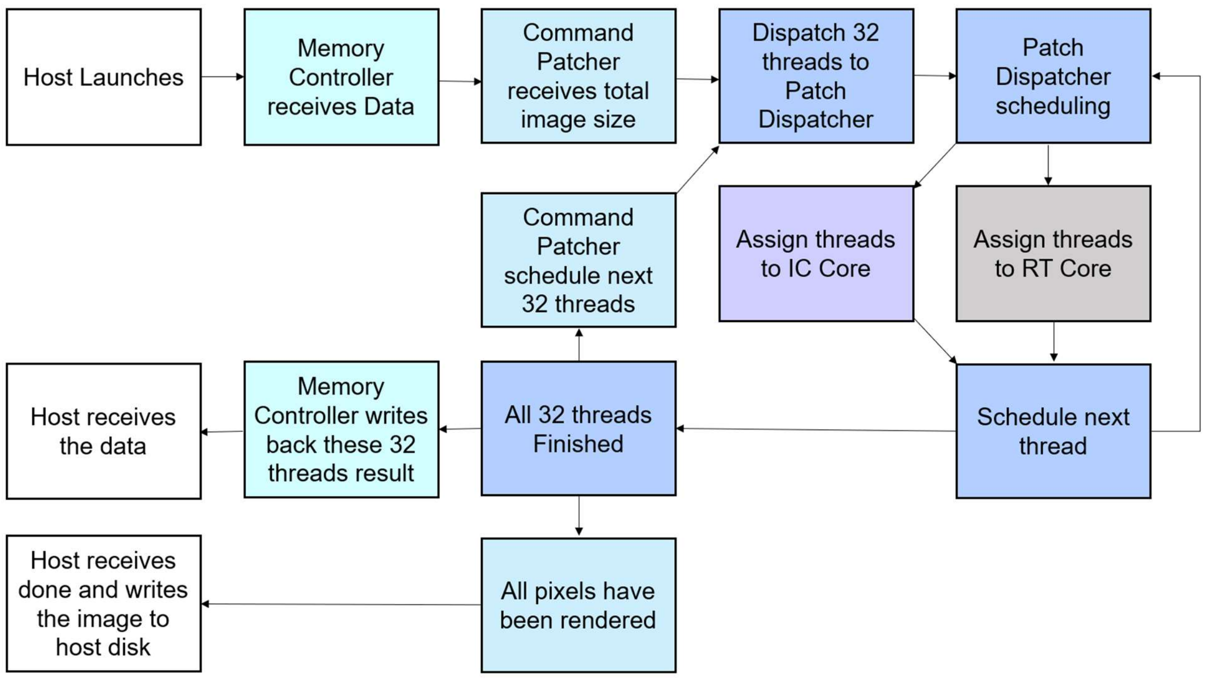
Host Launches
Memory Controller receives Data
Command Patcher receives total image size
Dispatch 32 threads to Patch Dispatcher
Patch Dispatcher scheduling
Command Patcher schedule next 32 threads
Assign threads to IC Core
Assign threads to RT Core
Host receives the data
Memory Controller writes back these 32 threads result
All 32 threads Finished
Schedule next thread
Host receives done and writes the image to host disk
All pixels have been rendered

Figure 4 Hardware control flow chart

2 Memory Controller

This is the hardware interface for software host. We wrote our own controller to load memory to from the host to our on-chip memory and we did not use the loader provided by the course martials. MMIO passes the starting address, length, and load information of each memory one by one. All information from MMIO is stored into registers. The read controller communicates with the DMA base on the information from MMIO to get content. Then, the data is parsed and formatted for the memory block, and the memory block write-enable signal is raised high. Another purpose of the memory controller is to write output to DMA for software to receive. The write controller requests information from main memory after the ray tracing is done, and the result is stored in registers for an easier write to DMA.

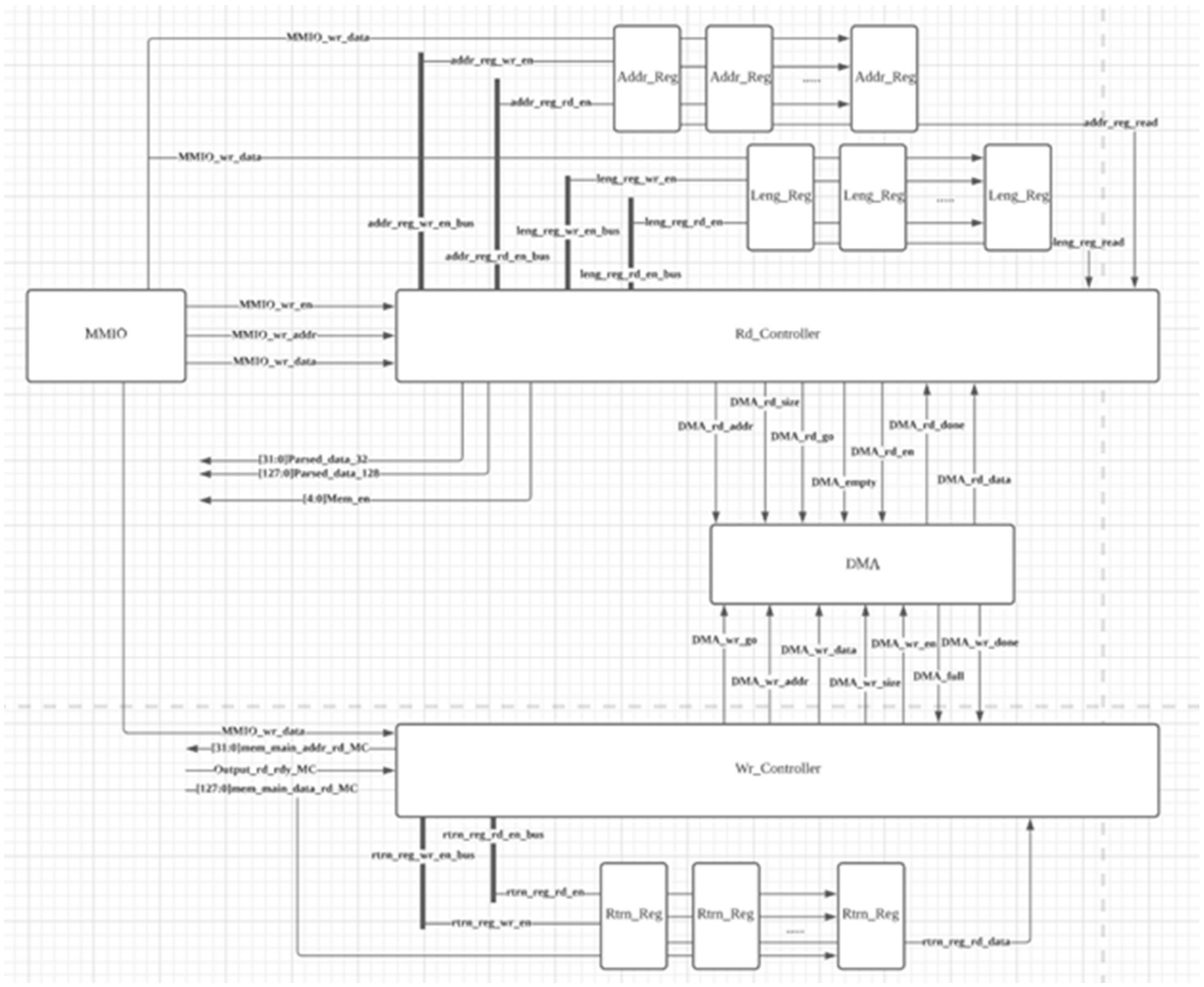

Figure 5 Block Diagram of the internal structure of Memory Controller

Global signals

Signal Name	I/O	Width	Description
Clk	I	1	Global Clock
Rst_n	I	1	Global Reset

MC interface

Signal Name	I/O	Width	Description
MMIO_wr_en	I	1	Write enable from MMIO indicating data is writing to MC. 1 = writing 0 = not writing
MMIO_wr_addr	I	63:0	The address from software indication what data is passing to MC
MMIO_wr_data	I	63:0	The data from MMIO
DMA_rd_done	I	1	Indicating the read from DMA is done. 1 = done

			0 = reading
DMA_rd_data	I	511:0	The data from DMA
DMA_full	I	1	Indicating the DMA is full or receiving data not yet finished. 1 = full 0 = not busy
DMA_wr_done	I	1	Indicating the writing from DMA is done. 1 = done 0 = writing
Output_rd_rdy_MC	I	1	Indicating that the result of ray tracing is ready to be read and write to DMA
Mem_main_data_rd_MC	I	127:0	The data from main memory
DMA_empty	I	1	The availability of data from DMA
DMA_rd_addr	O	63:0	The read address of DMA
DMA_rd_size	O	9:0	The size of transfer
DMA_rd_go	O	1	The start signals of reading from DMA
DMA_rd_en	O	1	The receive signal of data from DMA
DMA_wr_go	O	1	The start signals of writing to DMA
DMA_wr_addr	O	63:0	The writing address of DMA
DMA_wr_data	O	511:0	The data write to DMA
DMA_wr_size	O	1	The size of writing data to DMA
DMA_wr_en	O	1	Write enable for DMA

Registers.

Addr_Reg

Depth(bits)	RW	Width (bits)	Function
2:0	RW	63:0	The starting address for content of each memory block

Leng_Reg

Depth(bits)	RW	Width (bits)	Function
2:0	RW	9:0	The length of cache line of reading from DMA for each memory block

Rtrn_Reg

Depth(bits)	RW	Width (bits)	Function
1:0	RW	127:0	The return data from main memory for writing back to DMA

3 Command Patcher

The Command Patcher initiates a patch of threads for the patch dispatcher. Once the memory controller assigns CP a maximum pixel number and signals the CP to start operating, it calculates the correct pixel id and writes out the pixel id to patch dispatcher one thread per cycle for 32 cycles. Then it goes back to idle and waits for the done signal from the memory controller again. It also signals the memory controller when it finishes computing all the pixels.

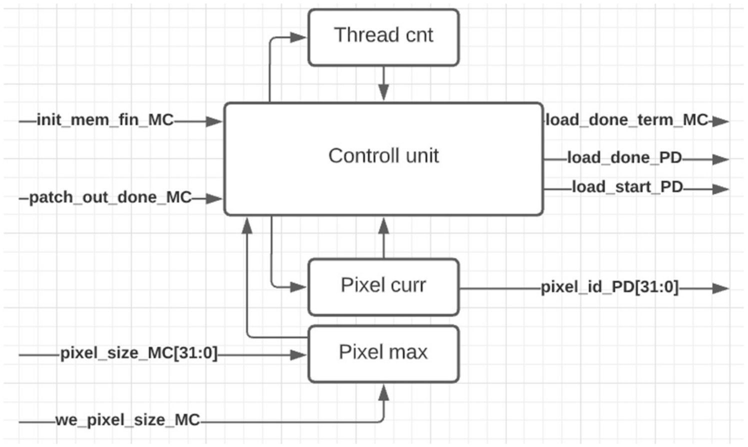

Figure 6 Block Diagram of the internal Structure of the Command Patcher

Signal Name (name_SRC/TGT)	I/O	Source/Target (blocks)	Description
Clk	I		Universal clock signal
Rst_n	I		Universal reset low signal
We_pixel_size_MC	I	Memory Controller	Enable high signal for one cycle that loads the maximum pixel id from MC.
[31:0] pixel_size_MC	I	Memory Controller	Maximum pixel size to patch, the processor ends after all pixels have been patched and calculated.
Init_mem_fin_MC	I	Memory Controller	Done signal from MC indicate that all memories have finished loading, act as the starting signal for CP and the whole processor.
Patch_out_done_MC	I	Memory Controller	Done signal from MC indicate that a patch of output has finished transmitting, act as the starting signal for CP.
Load_done_term_MC	O	Memory Controller	Done signal for one cycle to MC indicate that all the pixels have been patched.
Load_start_PD	O	Patch dispatcher	Starting signal for one cycle that indicate to the PD to start loading new threads, which takes 32 cycles.

Load_done_PD	O	Patch dispatcher	Done signal for one cycle after loading of a patch, which indicate to the PD that the job assignment is finished.
[31:0] Pixel_id_PD	O	Patch dispatcher	Pixel id assigned for the next thread to PD.

4 Patch Dispatcher

Patch Dispatcher oversees scheduling 32 threads of operations in between the available number of RT and IC cores. Whenever a RT core reaches the point where the next instruction involves tracing the intersection between the ray and triangles, it would notify the Patch Dispatcher to transfer the process to a dedicated hardware accelerator (IC). RT would send a context switch signal to Patch Dispatcher indicating that it is ready to be transferred. RT would also send its program counter and stack pointer so that this thread could restore its working status when being reassigned.

To re-schedule the thread context switched from RT, the thread ID would be enqueued into a FIFO structure named rt2ic_queue. FIFO structure enables a round-robin fashion scheduling so that no thread is being starved. Every clock cycle Patch Dispatcher would assign the FIFO output thread id to an available IC core (if exists, otherwise no operation performed) and dequeue the FIFO. An internal structure is utilized to keep trace of the availability of each RT and IC core and updated when assigning jobs or when the jobs is done. Corresponding program counter and stack pointer are also stored in array type structure indexed by the thread ID.

When IC core finishes tracing and ready to be returned to the RT, it would perform similar procedure by sending a context switch signal to Patch Dispatcher which will enqueue the thread ID into another FIFO structure named ic2rt_queue. When any RT core becomes available, Patch Dispatcher will assign the FIFO output thread and dequeue the FIFO. Corresponding program counter, stack pointer and pixel ID are also sent to the RT core to restore the working status of this thread.

Patch Dispatch can deal with multiple number of RT and IC cores. The number of each core are parameterized into the design. For multiple context switch signals from multiple cores at the same clock cycle, Patch Dispatcher would cache the context switch signal and perform the enqueue operation in a fixed descending order once per clock cycle. The number of jobs assigned for each direction (RT to IC or IC to RT) is also one since only one element dequeue is supported by the FIFO structure.

The number of RT core, however, cannot exceed four due to the limitation of the main memory. Main memory supports a maximum of four vector (512bit) memory access bandwidth due to the timing limitation. The number of IC core, on the other hand, could be set to as large as the number of threads.

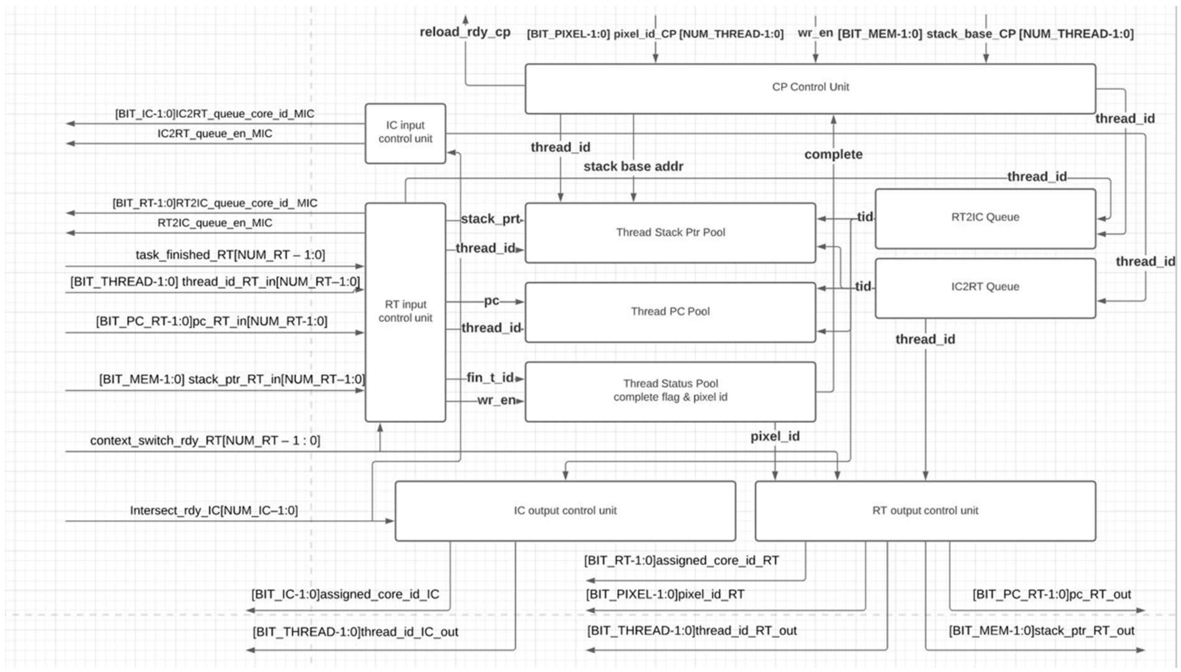
reload_rdy_cp
[BIT_PIXEL-1:0] pixel_id_CP [NUM_THREAD-1:0]
wr_en
[BIT_MEM-1:0] stack_base_CP [NUM_THREAD-1:0]
CP Control Unit
thread_id
[BIT_IC-1:0]IC2RT_queue_core_id_MIC
IC2RT_queue_en_MIC
IC input control unit
thread_id
stack base addr
complete
thread_id
[BIT_RT-1:0]RT2IC_queue_core_id_MIC
RT2IC_queue_en_MIC
task_finished_RT[NUM_RT – 1:0]
[BIT_THREAD-1:0] thread_id_RT_in[NUM_RT–1:0]
[BIT_PC_RT-1:0]pc_RT_in[NUM_RT-1:0]
[BIT_MEM-1:0] stack_ptr_RT_in[NUM_RT–1:0]
RT input control unit
stack_prt
thread_id
pc
thread_id
fin_t_id
wr_en
Thread Stack Ptr Pool
Thread PC Pool
Thread Status Pool complete flag & pixel id
tid
tid
RT2IC Queue
IC2RT Queue
thread_id
thread_id
context_switch_rdy_RT[NUM_RT – 1 : 0]
pixel_id
Intersect_rdy_IC[NUM_IC–1:0]
IC output control unit
RT output control unit
[BIT_RT-1:0]assigned_core_id_RT
[BIT_IC-1:0]assigned_core_id_IC
[BIT_PIXEL-1:0]pixel_id_RT
[BIT_PC_RT-1:0]pc_RT_out
[BIT_THREAD-1:0]thread_id_IC_out
[BIT_THREAD-1:0]thread_id_RT_out
[BIT_MEM-1:0]stack_ptr_RT_out

Figure 7 Block diagram of the internal microarchitecture of Patch Dispatcher

Signal Name (name_SRC/TGT)	I/O	Source/Target (blocks)	Description
Clk	I		Universal clock signal
Rst_n	I		Universal reset low signal
Wr_en_CP	I	Command Patcher	Enable high signal for one cycle that starts the load of new thread, which takes 64 cycles.
[31:0] pixel_id_CP	I	Command Processor	Unique id for the current pixel that is been calculated, id = y*num_column + x, used to compute the initial direction of the ray
[31:0] stack_base_CP	I	Command Processor	Initial allocated stack base pointer for each thread
Task_finished_RT [3:0]	I	Ray Tracing Core	Finish signal from RT core to indicate the job is done, and output data is ready.
Context_switch_rdy_RT[3:0]	I	Ray Tracing Core	Context switch signal ready from RT, indicate that all registers are saved, and RT is ready to compute a new thread.
[5:0] thread_id_RT_in[3:0]	I	Ray Tracing Core	Thread Id indicate which job is running on each RT core.
[31:0] stack_ptr_RT_in[3:0]	I	Ray Tracing Core	Stack pointer of each RT core.

[31:0] pc_RT_in [3:0]	I	Ray Tracing Core	Program counter of each RT core.
Intersect_rdy_IC[3:0]	I	Intersection Core	Context switch signal ready from IC, indicate that IC is ready to compute a new thread.
[5:0] thread_id_IC_in[3:0]	I	Intersection Core	Thread Id indicate which job is running on each IC core.
Reload_rdy_CP	O	Command Patcher	Indicate all 64 threads has finished computing and ready to load another 64 new threads.
[1:0]Assigned_core_id_RT	O	Ray Tracing Core	Core Id indicate which RT core should receive this new job.
[31:0] pixel_id_RT	O	Ray Tracing Core	Pixel id of the new job
[5:0] thread_id_RT_out	O	Ray Tracing Core	Thread Id of this new job
[31:0] stack_ptr_RT_out	O	Ray Tracing Core	Stack pointer of this new job
[31:0] pc_RT_out	O	Ray Tracing Core	Program counter of this new job
[1:0] assigned_core_id_IC	O	Intersection Core	Core Id indicate which IC should receive this new job.
[5:0] thread_id_IC_out	O	Intersection Core	Thread id of this new job.
[3:0] RT2IC_queue_core_id_MIC	O	Intersection Core Memory	RT core id for MIC indicate which RT's output should MIC enqueue.
[3:0] IC2RT_queue_core_id_MIC	O	Intersection Core Memory	IC id for MIC indicate which IC's output should MIC enqueue.
RT2IC_queue_en_MIC	O	Intersection Core Memory	Enable enqueue for RT2IC queue.
IC2RT_queue_en_MIC	O	Intersection Core Memory	Enable enqueue for IC2RT queue.

Stack pointer/ PC/ Status pool registers

Depth	RW	Width	Function
5:0	RW	31:0	32-bits Data (stack ptr/PC/status) for each thread is stored in this pool, it can be accessed by its associated thread id.

5 RT core

Our RT CPU adopts a 5-stage pipelined standard design. IF stage will find the next PC counter and send this address location to instruction memory before the next clock edge. DE stage will decode the instructions into multiple control signals. Since our R0 is read only, when an address is a don't care, we always set it to 0. If the current instruction at the DE stage is a branch, it will branch after the Flag writing from Integer ALU.

In EX stage and MEM stage, as most of our operations cannot be done in one cycle. we implemented a state machine to wait for the done signal from Floating point unit or Main memory. Our vectors are always floating so we implement 4 float ALU to compute each element at the same time. One special thing we added was the Vector Reduce Adder. Vectors reduce needs to two back-to-back floating points addition and it is too hard to fit in one pipeline stage. We separated this instruction and put it in both EX and MEM stage.

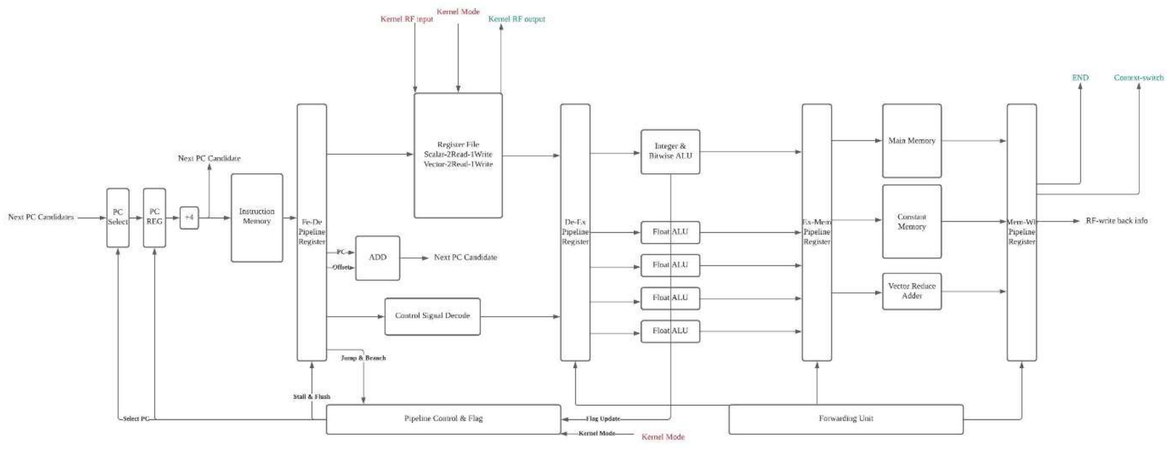

Figure 8 Block Diagram of the internal Structure of the Ray Tracing Core

Global signals

Signal Name	I/O	Width	Description
Clk	I	1	Global Clock
Rst_n	I	1	Global Reset

PD_RT Interface related signals

Signal Name	I/O	Width	Description
Kernel_mode	I	1	This signal will determine if the core is running on kernel mode. In Kernel mode, the pipeline will stall and the core itself cannot write to anything. The input to the Register file will be switch to signals coming from interface. 0 = Normal mode 1 = Kernel mode

PD_scalar_wen	I	1	**This signal is useful only in KERNEL mode!** When enable is active, the register file will write the PD_scalar_wb_data to PD_scalar_wb_address 0 = does not write back 1 = enable write back
PD_scalar_wb_address	I	4:0	**This signal is useful only in KERNEL mode!** Determine which register the data should be write to
PD_scalar_wb_data	I	31:0	**This signal is useful only in KERNEL mode!** Data being write to the Register File
PD_vector_wen	I	1	**This signal is useful only in KERNEL mode!** When enable is active, the register file will write the PD_vector_wb_data to PD_vector_wb_address 0 = does not write back 1 = enable write back
PD_vector_wb_address	I	3:0	**This signal is useful only in KERNEL mode!** Determine which register the data should be write to
PD_vector_wb_data	I	31:0	**This signal is useful only in KERNEL mode!** Data being write to the Register File
PD_scalar_read_address1	I	4:0	**This signal is useful only in KERNEL mode!** The register file will output the information in the register specified by this address. Data will be ready at port 1
PD_scalar_read_address2	I	4:0	**This signal is useful only in KERNEL mode!** The register file will output the information in the register specified by this address. Data will be ready at port 2
PD_vector_read_address1	I	3:0	**This signal is useful only in KERNEL mode!** The register file will output the information in the register specified by this address. Data will be ready at port 1
PD_vector_read_address2	I	3:0	**This signal is useful only in KERNEL mode!** The register file will output the information in the register specified by this address. Data will be ready at port 2
PD_scalar_read1	O	31:0	**This signal is useful only in KERNEL mode!** Supply data to specified by a PD_scalar_read_address1
PD_scalar_read2	O	31:0	**This signal is useful only in KERNEL mode!** Read data specified by a PD_scalar_read_address2
PD_vector_read1	O	127:0	**This signal is useful only in KERNEL mode!** Read data specified by a PD_vector_read_address1
PD_vector_read2	O	127:0	**This signal is useful only in KERNEL mode!** Read data specified by a PD_vector_read_address2

END	O	1	This signal will notify the Patch dispatcher that the current thread running on the core has finished. Then patch dispatcher can assign other thread to this core.
Context_switch	O	1	This signal will notify the interface that this thread wants to do a ray trace and this thread handled all register other than Stack based Pointer and PC. Then Interface will read tracing data, stack pointer and PC from the core then send to Patch dispatcher and IC memory.

Memory Related Signals

Signal Name	I/O	Width	Description
MRTI_addr	O	31:0	The address of next PC instruction
MRTI_data	I	31:0	The current instruction read from instruction memory
MEM_addr	O	31:0	The address of memory address
MEM_data_write	O	127:0	Memory data write to the memory
MEM_data_read	I	127:0	Memory data read from the memory
MEM_read_en	O	1	Enable memory read
MEM_write_en	O	1	Enable memory write
MEM_done	I	1	Indicate the read data is ready
MEM_s_or_v	O	1	Indicate the current operation is scalar or vector

6 IC buffer

The IC buffer is the storage space between intersection cores and ray-tracing cores. Once any core has finished its calculations, the patch dispatcher will control the buffer to select and load the corresponding data into the buffer queue. After the same thread has been assigned again on the other type of core. IC cores and RT cores then read from this memory and dequeue the buffer when starting. This process utilizes two queues which support random-access writes to store and pass data. All of the input data will be written into the tail of the queue. All IC cores and RT cores connect to the data input and MUXs are used to choose input based on PD signals. For example, once an IC has finished and signals the PD to push data into the queue, it waits until it has been re-assigned to a RT core, and then dequeues the data from the front of the queue. Since enqueue and dequeue are carefully managed by PD, the data remains synchronized with the PD assignment.

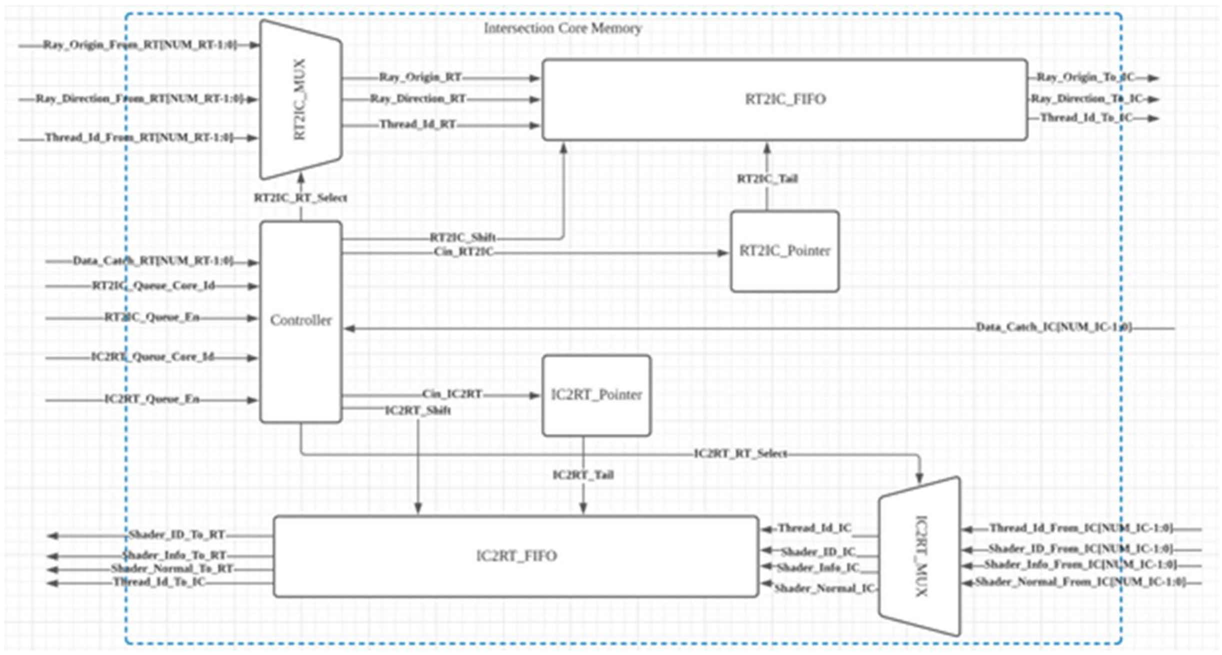

Figure 9 Block Diagram of the internal Structure of intersection Core buffer

Global signals

Signal Name	I/O	Width	Description
Clk	I	1	Global Clock
Rst_n	I	1	Global Reset

ICM interface

Signal Name	I/O	Width	Description
Ray_origian_RT[3:0]	I	95:0	The ray origin transferred from RT cores.
Ray_direction_RT[3:0]	I	95:0	The direction of the ray transferred from RT cores.
Dequeue_RT	I	1	Dequeue signal from RT indicate that RT has finished reading the data and it is safe to discard the data in queue.
Q_en_rt2ic_PD	I	1	Enqueue signal from PD indicates that RT has finished calculation and should enqueue data from the selected RT.
Core_id_rt2ic_PD [3:0]	I	1	One hot selects signal indicates which RT core to read data from.
Q_en_ic2rt_PD	I	1	Enqueue signal from PD indicates that IC has finished calculation and should enqueue data from the selected IC.
Core_id_ic2rt_PD [3:0]	I	1	One hot selects signal indicates which IC core to read data from.

Shader_Info_IC[3:0]	I	127:0	The coordinate and shader id of intersection points transferred from IC cores.
Normal_IC[3:0]	I	95:0	The normal vector of the triangle on the intersection points transferred from IC cores.
Dequeue_IC	I	1	Dequeue signal from IC indicate that IC has finished reading the data and it is safe to discard the data in queue.
Ray_origian_IC	O	95:0	The ray origin data output for IC cores.
Ray_direction_IC	O	95:0	The direction of the ray output for IC cores.
Shader_Info_RT	O	127:0	The coordinate of intersection points output for RT cores.
Shader_Normal_RT	O	95:0	The normal vector of the triangle on the intersection points output for RT cores

RT2IC Queue

Depth(bits)	RW	Width (bits)	Function
31:0	RW	191:0	A queue with the supporting of random access write to the tail of the queue for storing ray information from RT cores and pass it to IC cores. A pointer points to the tail for writing new data.

IC2RT Queue

Depth(bits)	RW	Width (bits)	Function
31:0	RW	223:0	A queue with the supporting of random access write to the tail of the queue for storing ray information from RT cores and pass it to IC cores. A pointer points to the tail for writing new data.

7 IC core

Our intersection core (IC) is implemented with a 4-stage pipelined design. The IC utilizes Möller-Trumbore algorithm for fast detection of intersections. Floating point addition, subtraction, multilocation, reciprocal, and comparison are used in this module.

Based on the nature of the algorithm, the process can be divided in to 4 serial stages, allowing us to pipeline our detections. Furthermore, stage pipelines can be switched to using multiple FIFOs with different lengths to meet the latency between input signals. By doing this, the throughput can be further increased to one cycle per output. However, this "FIFO design" has two downsides. First, it requires a fixed clk frequency since the latency of the IPs are depended on the clk frequency. The FIFOs cannot be too long nor too short. It would lead to false results. Secondly, the bottle neck of the performance would then be fetching from Triangle Memory. It requires at least 6 cycles to fetch. Therefore, using a stage design can best balance the throughput and robustness of the design.

The detection goes through all of the triangles and finds the closest intersecting triangle. The IC receives the starting signal from PD and ray information from IC memory. The controller would fetch the triangle data with triangle memory. Based on the return data, it can be determined where the ray intersects the triangle, if it intersects the triangle at all. Finally, the IC core returns the necessary data back to IC memory. If there is no intersection with any triangle, then shader ID 0 is returned to indicate. We further notice that there is one more condition for intersecting with a triangle that is not present in the algorithm, which is the distance t between origin of the light and the triangle should be positive. The original algorithm only comparing the determent, u, and v to determine hitting or not. This is crucial for getting the right shadow of the final rendering.

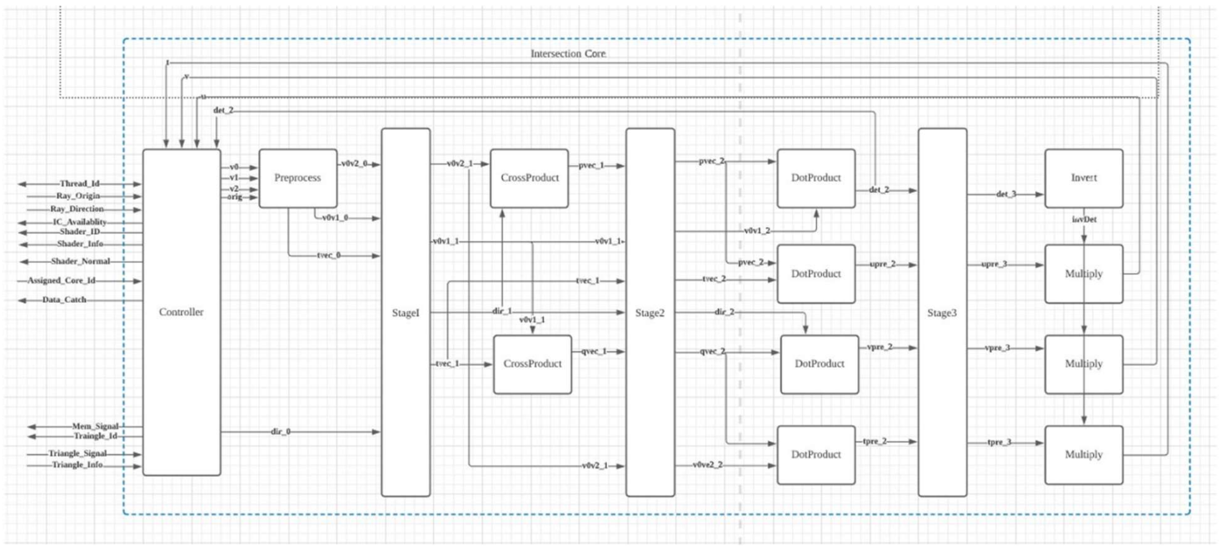

Figure 10 Block Diagram of the internal Structure of the Intersection Core

Global signals

Signal Name	I/O	Width	Description
Clk	I	1	Global Clock
Rst_n	I	1	Global Reset

IC interface

Signal Name	I/O	Width	Description
Thread_Id	I/O	5:0	Documenting the thread id of the intersection detection. This is passed from and sent back to IC memory
Ray_origin	I	127:0	The origin of the ray
Ray_direction	I	127:0	The direction of the ray
Assigned_core_Id	I	1:0	The start signals from PD. Only receiving ray data from IC memory when this signal matches with the core ID
Triangle_Signal	I	1:0	The triangle information ready signal. Only receiving triangle data when this signal matches with the core ID

Triangle_Info	I	127:0	The triangle information receiving from triangle memory
IC_availability	O	1	Indicating that the IC core finished its job and is available. 1 = available 0 = busy
Shader_Id	O	8:0	Return the ID of closest intersecting triangle. If no intersection, return 0
Shader_Info	O	127:0	The coordinate of intersection point. If no intersection, don't care
Shader_Normal	O	127:0	The normal vector of the triangle on the intersection point. If no intersection, don't care
Data_catch	O	1	Indicating that the information from IC memory is catched. Stay high for 1 cycle after catched. 1 = catched 0 = not yet catched
Mem_signal	O	1	Indicating that the core is requesting triangle information. 1 = requesting 0 = not requesting
Traingle_Id	O	8:0	The id of requesting triangle

8 IP cores

Ray tracing has many calculations in the algorithms, including both integers and floating point. In order to have a better performance per resource, we decided to use IP cores for the fundamental calculations for the integers and floating points.

The IP cores are Intel Arithmetic IPs for fixed point and floating point. By setting the fraction of the fixed point to 0, we can use it as integers. Even though Intel IP cores are very powerful, it has some inconveniences for us to integrate them into our design. First, almost all the IPs need a couple cycles to calculate. However, there is no 'done' signal for us to now whether the calculation has finished or not. Luckily, the IP provided a estimated latency for different clock frequency. We could use those information to write a counter to generate a 'done' signal for outer level. Secondly, as mentioned above, the latency for different IPs is different under different clock frequency. And, this information is not documented in anywhere since it is device specific. We have to record each latency manually while generating the IPs. Third, the simulation of the IPs is very difficult. All of them requires a library called 'fourteennm' which is not included for the standard version Modelsim. After many trails and errors, we found out that the Pro version of Modelsim is needed to simulate IPs. Even though there is a free starter license for the Pro version, it limits the line count of the simulated project to 10,000 lines. When we tested our ALUs, we found out that the IP themselves had already surpassed the line limitation.

Even though there is many hardships with using the Intel IPs, the performance is promising. The best benefit of them all is that it guarantees the timing for synthesis.

9 Memory

Memory space is separated into several groups on the top level to serve specific purposes. Specifications of each memory module are demonstrated in the following table. We have optimized the memory structure to best support our multicore design structure.

9.1 Memory overview

Name	Type	Internal Data Width (bit)	IO Data Width (bit)	Depth	Bank #	Size (bit)
Command Patcher Instruction	Simple Dual Port	32	32	512	1	16 K
RT Instruction x 4	Simple Dual Port	32	32	4096	1	128 x 4 = 512 K
RT Constant x 4	Simple Dual Port	32	32	512	1	16 x 4 = 64 K
Triangle x 4	Simple Dual Port	32	32	2048	4	16 x 4 x 4 = 256 K
Main Memory	Simple Dual Port	32	32 x 4 x 4	4096	256	32 M

Figure 11 Memory Specification table

9.2 Instruction memory

RT instruction memory module stores the software instructions that will be run on the RT core. The instructions will be initially loaded from the memory controller incrementally and terminate with a control signal assigned by MC. There is a single ram bank within the module to support one read from RT or one write from MC each cycle.

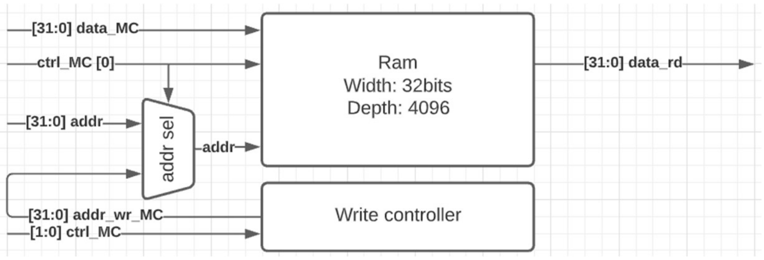

Figure 12 Block diagram of the internal structure of the Ray Tracing instruction memory

Signal Name	I/O	Source/Target (blocks)	Description
clk	I		Universal Clock signal
Rst_n	I		Universal reset signal
[1:0] ctrl_MC	I	Memory controller	Control signal for memory controller. Bit 0 is the write enable signal. Bit 1 is the write finished signal.
[31:0]data_MC	I	Memory controller	Data to write to the address by MC.
[31:0]addr	I	Ray Tracing	Address to read from by RT.
[31:0]data_rd	O	Ray Tracing	Data read from the address by RT.

Memory Bank

Bits	RW	Width	Function
11:0	RW	31:0	Memory bank used to store the instructions of RT.

9.3 Constant memory

The RT constant memory module stores the software constants that are needed for running the RT core. The constants will be initially loaded from the memory controller incrementally, terminating with a control signal assigned by MC. There is a single ram bank within the module to support one read from RT or one write from MC each cycle.

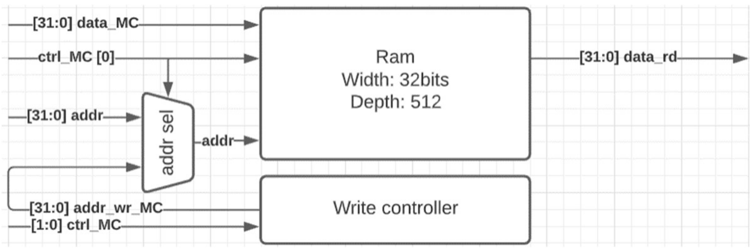

Figure 13 Block diagram of the internal structure of the Ray Tracing constant memory

Signal Name	I/O	Source/Target (blocks)	Description
clk	I		Universal Clock signal
Rst_n	I		Universal reset signal
[1:0] ctrl_MC	I	Memory controller	Control signal for memory controller. Bit 0 is the write enable signal. Bit 1 is the write finished signal.

[31:0]data_MC	I	Memory controller	Data to write to the address by MC.
[31:0]addr	I	Ray Tracing	Address to read from by RT.
[31:0]data_rd	O	Ray Tracing	Data read from the address by RT.

Memory Bank

Bits	RW	Width(bits)	Function
8:0	RW	32	Memory bank used to store the constant of RT.

9.4 Main memory

The RT Main memory is a unified memory space accessible to software, which is reserved for the stack space of each thread and ray tracing output. The RT main memory is word-addressable and supports four parallel vectors (128 bits width) reads/writes at the same time. The internal structure of this memory module consists of 128 banks of RAM with 4 banks per thread to minimize the memory conflict and maximize throughput. Since the logic essentially requires a mapping from 4x4x32 bits to 128x32 bits, in order to optimize for timing and throughput, we have developed a pipelined version of this memory. We first have made a mapping from 4x4x32 to 8x4x32 and then a second mapping from 8x4x32 to 128x32 in a stage of the pipeline to limit the size of the selecting logic. Thus, with the optimization, we are able to have the maximum throughput of 512 bits per cycle and thus vastly improved our timing as well as throughput.

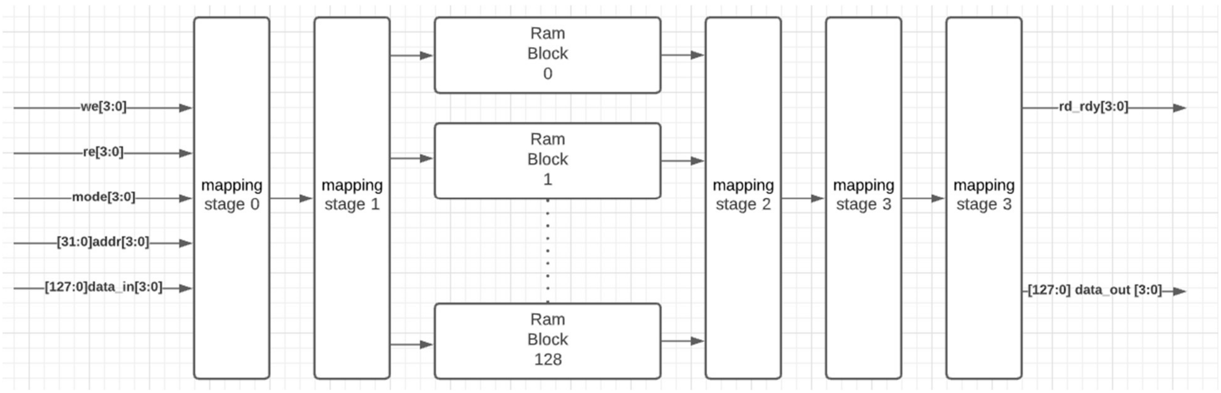

Figure 14 Block diagram of the Structure of the Main memory (Note: the mapping logics are hidden)

Signal Name	I/O	Source/Target (blocks)	Description
clk	I		Universal Clock signal
rst_n	I		Universal reset signal
we [3:0]	I	Ray tracing / Memory controller	Write enable signal for each port

re [3:0]	I	Ray tracing / Memory controller	Read enable signal for each port
mode [3:0]	I	Ray tracing	Mode signal for different mode of write for RT core 0: Scaler write 1: Vector write
[31:0] addr [3:0]	I	Ray tracing / Memory controller	Read/write address to read data from main memory
[127:0] data_in [3:0]	I	Ray tracing / Memory controller	Vector Data input written into the main memory
rd_rdy [3:0]	O	Ray tracing / Memory controller	Ready signal to indicate the read operation is complete and the data is available at data_out
[127:0] data_out [3:0]	O	Memory controller / Memory controller	Vector Data read from the main memory requested by the memory controller

Ram Bank[127:0]

Depth	RW	Width	Function
11:0	RW	31:0	Memory bank on which the unified main memory is mapped to enable parallel read / write without conflict. Each 4 linear Ram banks will be combined into a 2D structure on which the unified memory address maps in a row major fashion for a single thread, and total 32x4 banks for 32 threads.

9.5 Triangle memory

The triangle memory module stores the vertices of all triangles, their corresponding x, y, z coordinate data, and their shader ids which are used in the software for image calculation. Due to the overlapping of vertices among triangles, it would be wasteful to store all information on each vertex of each triangle in memory. Instead, we have designed a special data structure to optimize space utilization. Triangle memory module stores information on each individual vertex directly in the index memory bank, and stores a mapping table in the vertex memory banks. This is to find the corresponding vertex data of any given triangles using the mapping table. As a result, it reduces the memory required to store the triangles by a factor up to 3. On the other hand, we have also optimized the design to improve the throughput. To read the 320 bits data for one triangle (3x32 bits per-vertex and 32 bits shader id), the process takes up to 5 cycles, thus a prefetch mechanism is implemented to reduce latency. When the memory is idle from IC's quest, it starts prefetching the next triangle's data. Since IC is comparatively slower, prefetching lowers the latency to at minimum 1 cycle, thus increases relative throughput to 320 bits per cycle.

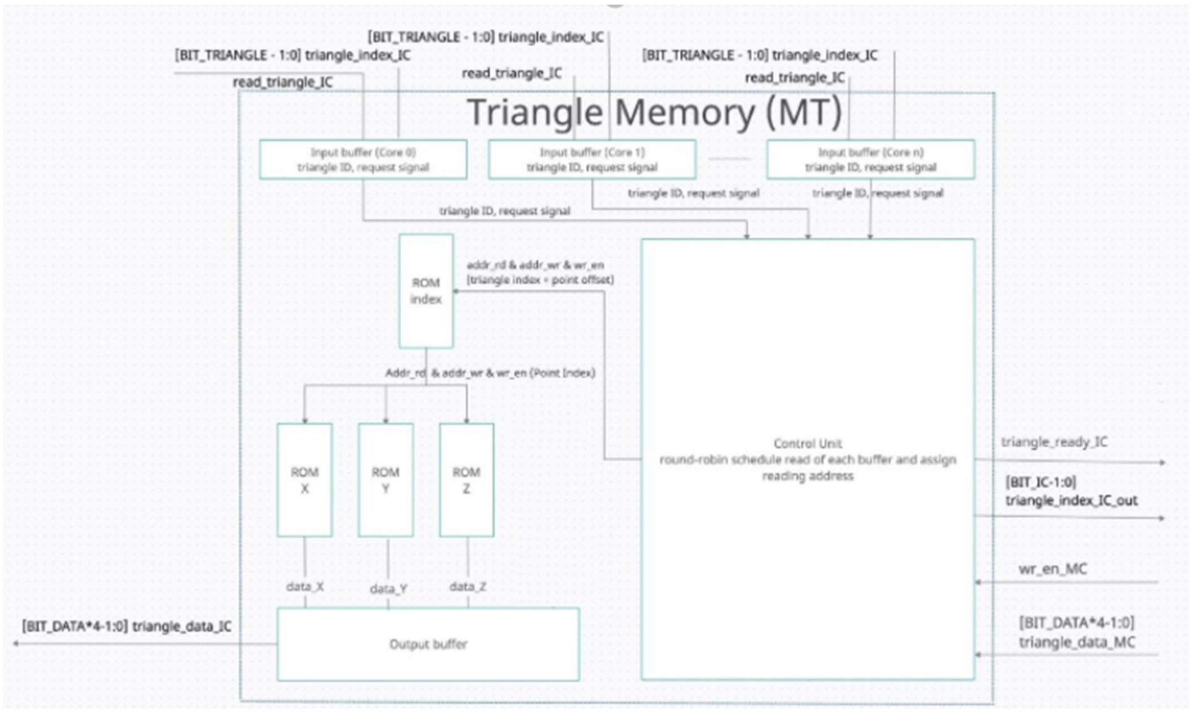
[BIT_TRIANGLE - 1:0] triangle_index_IC
[BIT_TRIANGLE - 1:0] triangle_index_IC
[BIT_TRIANGLE - 1:0] triangle_index_IC
read_triangle_IC
read_triangle_IC
read_triangle_IC
Triangle Memory (MT)
Input buffer (Core 0) triangle ID, request signal
Input buffer (Core 1) triangle ID, request signal
Input buffer (Core n) triangle ID, request signal
triangle ID, request signal
triangle ID, request signal
triangle ID, request signal
ROM index
addr_rd & addr_wr & wr_en (triangle index + point offset)
Addr_rd & addr_wr & wr_en (Point Index)
ROM X
ROM Y
ROM Z
data_X
data_Y
data_Z
Output buffer
[BIT_DATA*4-1:0] triangle_data_IC
Control Unit
round-robin schedule read of each buffer and assign reading address
triangle_ready_IC
[BIT_IC-1:0] triangle_index_IC_out
wr_en_MC
[BIT_DATA*4-1:0] triangle_data_MC

Figure 15 Block Diagram of the internal structure of Triangle Memory

Signal Name	I/O	Source/Target (blocks)	Description
clk	I		Universal Clock signal
rst_n	I		Universal reset signal
re_IC	I	Intersection core	Signal indicate IC want to read triangle data from the triangle memory.
[8:0] triangle_id_IC	I	Intersection core	Id for triangle indicate which triangle IC want to read.
[127:0] data_MC	I	Memory controller	Triangle data to write in during MC memory write.
We_MC	I	Memory controller	Write enable signal for MC data write.
Done_MC	I	Memory controller	Terminate signal from MC indicate that the loading stage of the memory is completed.
rdy_IC	O	Intersection core	Signals indicate the triangle read is ready.
Not_valid_IC	O	Intersection core	Terminate signal for IC indicate that the address id read is not a valid triangle id which acts as the termination signal for IC’s intersection calculation.

Rdy_MC	O	Memory controller	Singal indicate the triangle write from MC is done.
[95:0] vertex1_IC	O	Intersection core	X, Y, Z data information for the first vertex of the triangle read
[95:0] vertex2_IC	O	Intersection core	X, Y, Z data information for the second vertex of the triangle read
[95:0] vertex3_IC	O	Intersection core	X, Y, Z data information for the third vertex of the triangle read
[95:0] sid_IC	O	Intersection core	Shader id data for the triangle read, used in software calculation.

Memory Bank Index

Bits	RW	Function
10:0	RW	Memory bank used to store triangle data, this bank is used to store the vertex id and shader id of all triangles, each triangle takes 4 memory entries.

Memory Bank X/Y/Z

Depth	RW	Width	Function
10:0	RW	31:0	Memory bank used to store X / Y / Z coordinates and alpha value of each vertex.

Output Buffer

Bits	RW	Function
320:0	RW	Output buffer holds all the data output from triangle memory loaded

10 Coding standard

System Verilog:
All wire and interface name are in lowercase. Use underscore for separating words.
Name wire in the format: wirename_SS_DD.
Example: thread_id_cp_pd
Name interface in the format of: interfacename_XX
Example: thread_id_pd
If input and output have the same name, add _in or _out
Example: thread_id_pd_in or thread_id_pd_out

Module name		Wire name	
Command Patcher	CP	Pointer	ptr
Patch Dispatcher	PD	Ready	rdy
Ray Trace Core	RT	Address	addr
Intersection Core	IC	Read	rd
Triangle Memory	MT	Write	wr

IC Memory	MIC	Enable	en
CP Memory	MCP	Instruction	inst
RT Main Memory	MRTM	Constant	const
RT instruction memory	MRTI	Branch	br
RT constant memory	MRTC		
Memory Controller	MC		