

# LAB3实验报告

PB1811757 陈金宝

## 实验目标

阅读并理解助教提供的简单cache的代码，将它修改为N路组相连的（要求组相连度使用宏定义可调）、写回并带写分配的cache。要求实现FIFO、LRU两种替换策略。并将实现的Cache添加到Lab1的CPU中（替换先前的data cache），并添加额外的数据通路，统计Cache缺失率，在Cache缺失时，bubble当期指令及之后的指令。要求能成功运行这个算法（所谓成功运行，是指运行后的结果符合预期）

## 实验环境

操作系统:windows10 20H2  
仿真工具:Vivado 2019.2

## cache实现

为实现组相联，需要将助教原先的cache.sv中部分数据结构增加一个维度：

```
reg [31:0] cache_mem [SET_SIZE][WAY_CNT][LINE_SIZE];  
reg [TAG_ADDR_LEN-1:0] cache_tags [SET_SIZE][WAY_CNT];  
reg valid [SET_SIZE][WAY_CNT];  
reg dirty [SET_SIZE][WAY_CNT];
```

判断命中时，使用for语句并行判断。若命中则break。

```
always @ (*) begin  
    for(integer i = 0; i < WAY_CNT; i++) begin  
        if(valid[set_addr][i] && cache_tags[set_addr][i] == tag_addr) begin  
            cache_hit = 1'b1;  
            hit_pos = i;  
            break;  
        end  
    end  
    else begin  
        cache_hit = 1'b0;  
    end  
end  
end
```

## FIFO

为实现FIFO，为每个SET维持两个数：队首指针和长度。队首指针指向的set内的块号就是当前队列中最早入队的，也即是要被写入的块。队首指针和长度的初值均为0。当set内未满时（队列长度 < WAY\_CNT），则不进行换出，直接将新块换入到当前队列内队列长度对应的位置，并将队列长度+1和指针+1。当set满时指针正好回到0（如：当队列长度为0，也即set内是空时，新的line直接放入set内的第0个位置，队列长变为1，指针变为1）

若set内已经满，则此时需要将队首指针对应的块换出并将新块换入。同时使队首指针+1模WAY\_CNT。队首指针和队列长度的更新在状态IDLE中进行。

数据结构定义如下，其中cache\_fifo为存储每个set内队首指针。由于最大不超过WAY\_CNT。所以至多需要WAY\_CNT+1位即可表示。其中way\_length存储每个set内队列长度。由于最大不超过WAY\_CNT。所以至多需要WAY\_CNT+1位即可表示。swap\_out是要被写入的块号。用于向SWAP\_IN\_OK状态传递，指示要写入哪个块。

```
reg [      WAY_CNT:0] cache_fifo   [SET_SIZE];
reg [      WAY_CNT:0] way_length   [SET_SIZE];
reg [      WAY_CNT:0] swap_out;
```

fifo\_pos即是当前set的队首指针。queue\_len即是当前set的队列长度。

```
assign fifo_pos = cache_fifo[set_addr]; //queue header
assign queue_len = way_length[set_addr]; //length of the fifo queue
```

对队首指针和队列长度的维护在IDLE段。同时swap\_out用于传向SWAP\_IN\_OK，指示要写入的块号(即fifo\_pos)。

```
IDLE:  begin
        if(cache_hit) begin
            if(rd_req) begin
                rd_data <= cache_mem[set_addr][hit_pos][line_addr];
            end else if(wr_req) begin
                cache_mem[set_addr][hit_pos][line_addr] <= wr_data;
                dirty[set_addr][hit_pos] <= 1'b1;
            end
        end else begin
            if(wr_req | rd_req) begin
                swap_out <= fifo_pos;
                cache_fifo[set_addr] <= (fifo_pos + 1) % WAY_CNT;
                if(queue_len < WAY_CNT) begin
                    cache_stat <= SWAP_IN;
                    way_length[set_addr] <= queue_len + 1;
                end else begin
                    if(valid[set_addr][fifo_pos] && dirty[set_addr]
[fifo_pos]) begin
                        cache_stat <= SWAP_OUT;
                        mem_wr_addr <= {cache_tags[set_addr][fifo_pos],
set_addr};

                        mem_wr_line <= cache_mem[set_addr][fifo_pos];
                    end else begin
                        cache_stat <= SWAP_IN;
                    end
                end
                {mem_rd_tag_addr, mem_rd_set_addr} <= {tag_addr, set_addr};
            end
        end
    end
```

SWAP\_IN\_OK状态的操作:

```

SWAP_IN_OK: begin
    for(integer i=0; i<LINE_SIZE; i++)
        cache_mem[mem_rd_set_addr][swap_out][i] <= mem_rd_line[i];
    cache_tags[mem_rd_set_addr][swap_out] <= mem_rd_tag_addr;
    valid      [mem_rd_set_addr][swap_out] <= 1'b1;
    dirty      [mem_rd_set_addr][swap_out] <= 1'b0;
    cache_stat <= IDLE;
end

```

运行 $16 \times 16$ 矩阵乘法后的部分ram\_cell的仿真截图

> [1][31:0]	7fe9b631	7fe9b631
> [2][31:0]	41851251	41851251
> [3][31:0]	70e17a3a	70e17a3a
> [4][31:0]	fd8ae97b	fd8ae97b
> [5][31:0]	63de6762	63de6762
> [6][31:0]	02afd8a9	02afd8a9
> [7][31:0]	53505046	53505046
> [8][31:0]	75ed7940	75ed7940
> [9][31:0]	bf9d853a	bf9d853a
> [10][31:0]	85f43dc1	85f43dc1
> [11][31:0]	218f5fa3	218f5fa3
> [12][31:0]	ee4a481f	ee4a481f
> [13][31:0]	58f7590b	58f7590b
> [14][31:0]	28cc008d	28cc008d
> [15][31:0]	31752c1f	31752c1f
> [16][31:0]	85cf308d	85cf308d
> [61][31:0]	8eb6a299	8eb6a299
> [62][31:0]	17422555	17422555
> [63][31:0]	d8b9954b	d8b9954b
> [64][31:0]	5ac79831	5ac79831
> [65][31:0]	ae70e607	ae70e607

ram\_cell中的内容符合预期。

运行256个数的快排后的部分ram\_cell的仿真截图:



```

IDLE:  begin
        if(cache_hit) begin
            if(rd_req||wr_req)begin
                for(integer i = 0; i < stack_pos; i++)begin
                    lru_stack[set_addr][i+1] <= lru_stack[set_addr][i];
                end
                lru_stack[set_addr][0] <= hit_pos;
            end
            if(rd_req) begin
                rd_data <= cache_mem[set_addr][hit_pos][line_addr];
            end else if(wr_req) begin
                cache_mem[set_addr][hit_pos][line_addr] <= wr_data;
                dirty[set_addr][hit_pos] <= 1'b1;
            end
        end else begin
            if(wr_req | rd_req) begin
                if(stack_len < WAY_CNT)begin
                    cache_stat <= SWAP_IN;
                    way_length[set_addr] <= stack_len + 1;
                    lru_stack[set_addr][stack_len] <= stack_len;
                    swap_out <= stack_len;
                end else begin
                    swap_out <= lru_pos;
                    if(valid[set_addr][lru_pos] && dirty[set_addr][lru_pos])
begin
                        cache_stat <= SWAP_OUT;
                        mem_wr_addr <= {cache_tags[set_addr][lru_pos],
set_addr};

                        mem_wr_line <= cache_mem[set_addr][lru_pos];
                    end else begin
                        cache_stat <= SWAP_IN;
                    end
                end
                {mem_rd_tag_addr, mem_rd_set_addr} <= {tag_addr, set_addr};
            end
        end
    end
end

```

运行 $16 \times 16$ 矩阵乘法后的部分ram\_cell的仿真截图

> [1][31:0]	7fe9b631	7fe9b631
> [2][31:0]	41851251	41851251
> [3][31:0]	70e17a3a	70e17a3a
> [4][31:0]	fd8ae97b	fd8ae97b
> [5][31:0]	63de6762	63de6762
> [6][31:0]	02afd8a9	02afd8a9
> [7][31:0]	53505046	53505046
> [8][31:0]	75ed7940	75ed7940
> [9][31:0]	bf9d853a	bf9d853a
> [10][31:0]	85f43dc1	85f43dc1
> [11][31:0]	218f5fa3	218f5fa3
> [12][31:0]	ee4a481f	ee4a481f
> [13][31:0]	58f7590b	58f7590b
> [14][31:0]	28cc008d	28cc008d
> [15][31:0]	31752c1f	31752c1f
> [16][31:0]	85cf308d	85cf308d
> [61][31:0]	8eb6a299	8eb6a299
> [62][31:0]	17422555	17422555
> [63][31:0]	d8b9954b	d8b9954b
> [64][31:0]	5ac79831	5ac79831
> [65][31:0]	ae70e607	ae70e607
> [66][31:0]	0d13638e	0d13638e
> [67][31:0]	aad24331	aad24331
> [68][31:0]	3530c942	3530c942

ram\_cell中的内容符合预期

运行256个数的快排后的部分ram\_cell的仿真截图

> [1][31:0]	00000001	00000001
> [2][31:0]	00000002	00000002
> [3][31:0]	00000003	00000003
> [4][31:0]	00000004	00000004
> [5][31:0]	00000005	00000005
> [6][31:0]	00000006	00000006
> [7][31:0]	00000007	00000007
> [8][31:0]	00000008	00000008
> [9][31:0]	00000009	00000009
> [10][31:0]	0000000a	0000000a
> [11][31:0]	0000000b	0000000b
> [12][31:0]	0000000c	0000000c
> [13][31:0]	0000000d	0000000d
> [14][31:0]	0000000e	0000000e
> [15][31:0]	0000000f	0000000f
> [16][31:0]	00000010	00000010
> [61][31:0]	0000003d	0000003d
> [62][31:0]	0000003e	0000003e
> [63][31:0]	0000003f	0000003f
> [64][31:0]	00000040	00000040
> [65][31:0]	00000041	00000041

## 缺失率统计

对缺失率的统计在WBSegReg中进行。

miss时cache\_miss会持续50个周期。统计时只统计一次，用状态机来实现。cache\_miss时最终会转化为不miss的情况(目标块会调入)，所以统计总次数时只统计不cache\_miss的情况

相关实现如下:

```
wire we;
assign we = |WE;
reg [31:0] miss_cnt;
reg [31:0] total_cnt;
reg state;

always@(posedge clk or posedge rst) begin
    if(rst)begin
        state <= 0;
        miss_cnt <= 0;
    end else begin
        case(state)
            1'b0:begin
                if(DCacheMiss) begin
                    miss_cnt <= miss_cnt + 1;
                    state <= 1'b1;
                end
            end
            1'b1:begin
                if(!DCacheMiss) begin
                    state <= 1'b0;
                end
            end
        endcase
    end
end

always@(posedge clk or posedge rst) begin
    if(rst)begin
        total_cnt <= 0;
    end else begin
        if((MemReadM || we)&&!DCacheMiss) begin
            total_cnt <= total_cnt + 1;
        end
    end
end
```

仿真后通过miss\_cnt和total\_cnt的值就可以计算缺失率，通过最后一次访存的时间估算运行时间

同时对cache分别综合，得到不同策略，不同参数所使用的硬件相应信息。

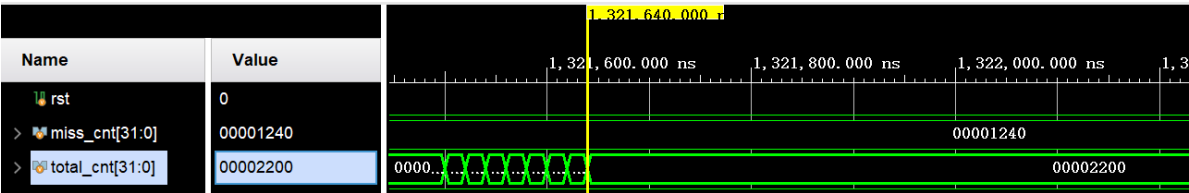
## 统计分析

对FIFO,LRU分别使用 $16 \times 16$ 矩阵乘法，256个数的快排进行仿真。仿真时取cache参数为3,3,6,3、3,3,6,4和3,3,6,5。

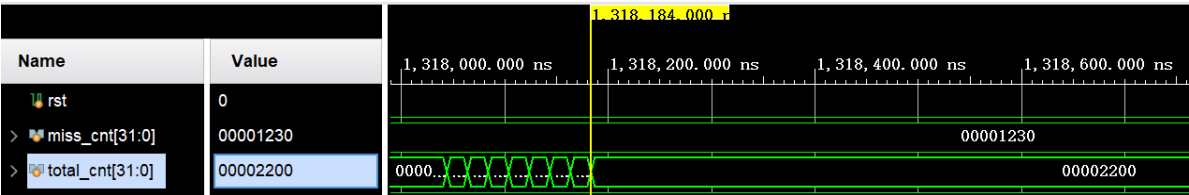
分析结果如下。

当cache的参数为3, 3, 6, 3时的结果

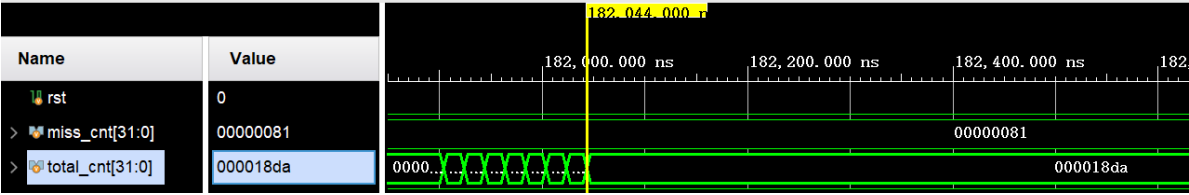
fifo, $16 \times 16$ 矩阵乘:



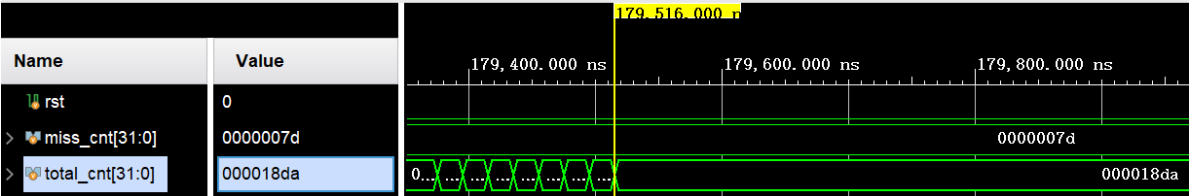
lru,16×16矩阵乘



fifo,256个数快排



lru,256个数快排



fifo所需硬件资源

Resource	Utilization	Available	Utilization %
LUT	3248	63400	5.12
FF	7337	126800	5.79
BRAM	4	135	2.96
IO	81	210	38.57

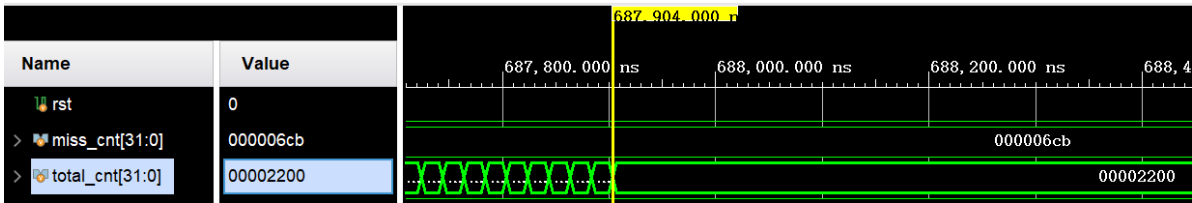
lru所需硬件资源

Resource	Utilization	Available	Utilization %
LUT	3295	63400	5.20
FF	7420	126800	5.85
BRAM	4	135	2.96
IO	81	210	38.57

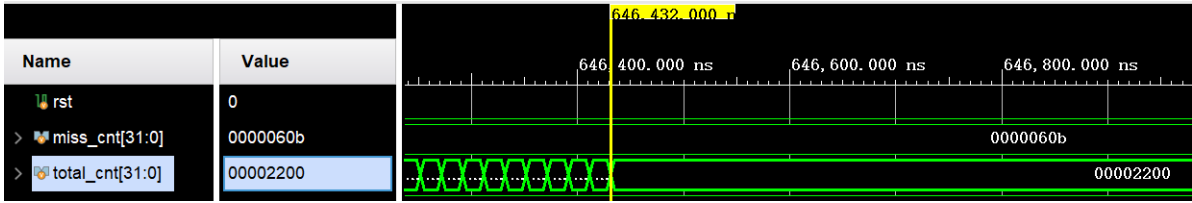
当cache参数为3364时的结果:

fifo,16×16矩阵乘:

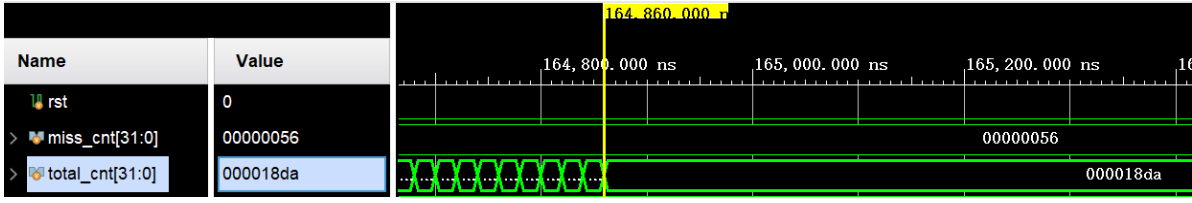




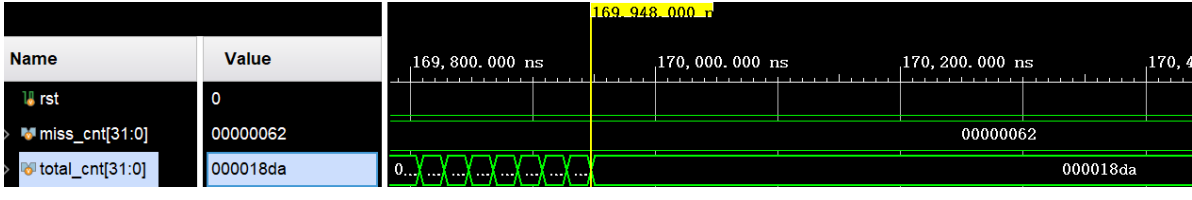
lru, 16×16矩阵乘



fifo, 256个数快排



lru, 256个数快排



fifo所需硬件资源

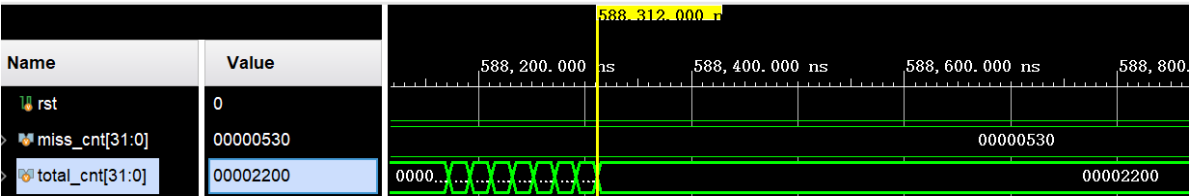
Resource	Utilization	Available	Utilization %
LUT	4146	63400	6.54
FF	9455	126800	7.46
BRAM	4	135	2.96
IO	81	210	38.57

lru所需硬件资源

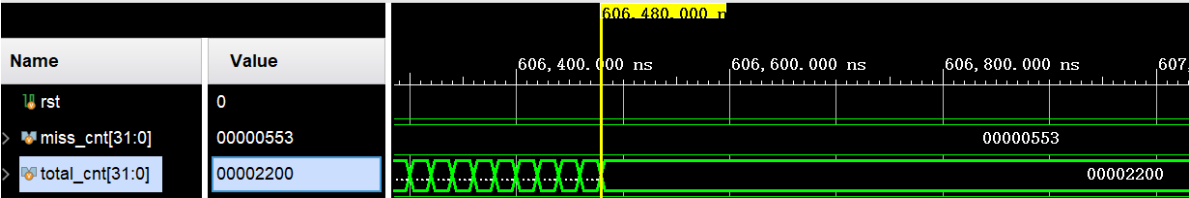
Resource	Utilization	Available	Utilization %
LUT	4350	63400	6.86
FF	9611	126800	7.58
BRAM	4	135	2.96
IO	81	210	38.57

当参数为3365时:

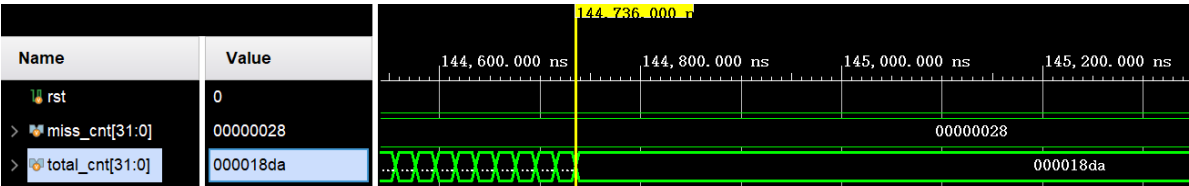
fifo,16×16矩阵乘



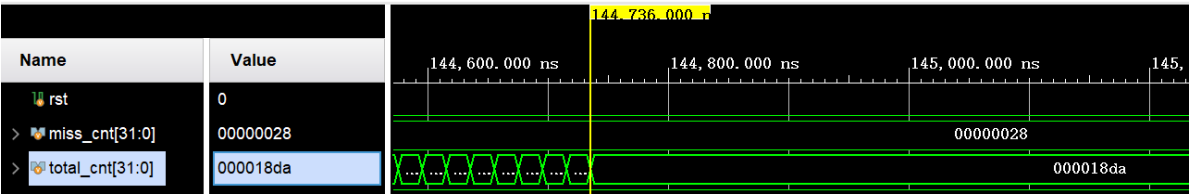
LRU,16×16矩阵乘



fifo,256个数快排



LRU,256个数快排



fifo所需硬件资源

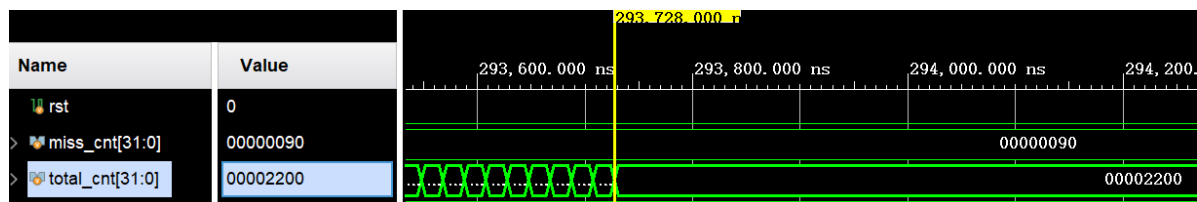
Resource	Utilization	Available	Utilization %
LUT	4986	63400	7.86
FF	11591	126800	9.14
BRAM	4	135	2.96
IO	81	210	38.57

LRU所需硬件资源

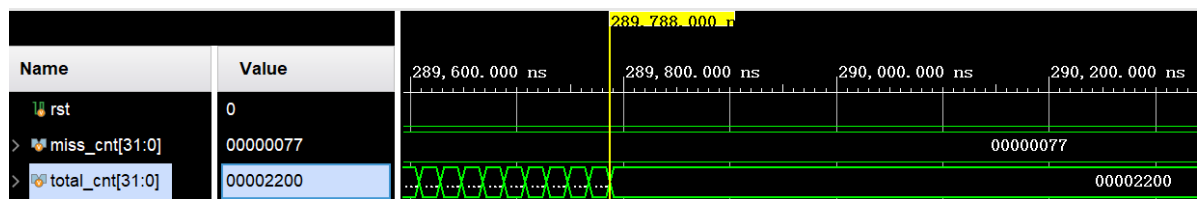
Resource	Utilization	Available	Utilization %
LUT	5114	63400	8.07
FF	11815	126800	9.32
BRAM	4	135	2.96
IO	81	210	38.57

当参数为3454时:

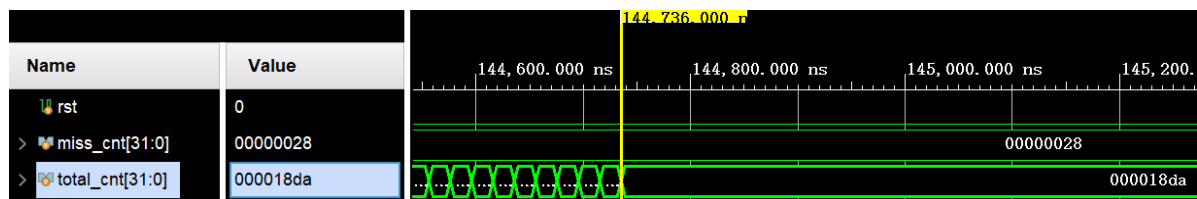
fifo,16×16矩阵乘



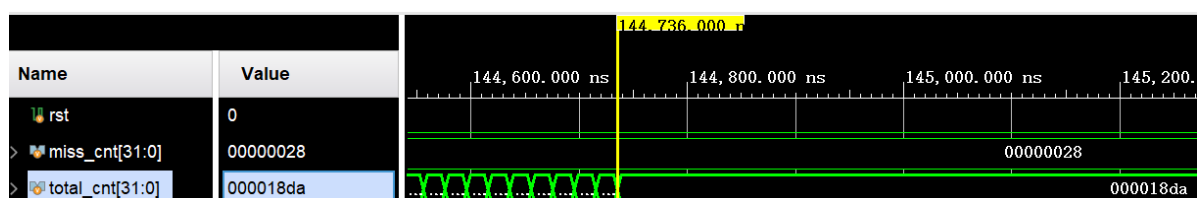
LRU,16×16矩阵乘



fifo,256个数快排



LRU,256个数快排



fifo所需硬件资源

Resource	Utilization	Available	Utilization %
LUT	7577	63400	11.95
FF	17904	126800	14.12
BRAM	4	135	2.96
IO	81	210	38.57

LRU所需硬件资源

Resource	Utilization	Available	Utilization %
LUT	7995	63400	12.61
FF	18195	126800	14.35
BRAM	4	135	2.96
IO	81	210	38.57

将上述结果制成表格，运行时间以最后一次访存为计算依据(total\_cnt不再变化)。

策略	参数	硬件资源(LUT,FF)	算法	运行时间(ns)	缺失率
FIFO	3363	3248,7337	MatMul 16*16	1,321,640	53.68%
LRU	3363	3295,7420	MatMul 16*16	1,318,184	53.49%
FIFO	3363	3248,7337	QuickSort 256	182,044	2.03%
LRU	3363	3295,7420	QuickSort 256	179,516	1.96%
FIFO	3364	4146,9455	MatMul 16*16	687,904	19.98%
LRU	3364	4350,9611	MatMul 16*16	646,432	17.77%
FIFO	3364	4146,9455	QuickSort 256	164,860	1.35%
LRU	3364	4350,9611	QuickSort 256	169,948	1.54%
FIFO	3365	4986,11591	MatMul 16*16	588,312	15.26%
LRU	3365	5114,11815	MatMul 16*16	606,480	15.66%
FIFO	3365	4986,11591	QuickSort 256	144,736	0.63%
LRU	3365	5114,11815	QuickSort 256	144,736	0.63%
FIFO	3454	7577,17904	MatMul 16*16	293,728	1.65%
LRU	3454	7995,18195	MatMul 16*16	289,788	1.37%
FIFO	3454	7577,17904	QuickSort 256	144,736	0.63%
LRU	3454	7995,18195	QuickSort 256	144,736	0.63%

通过以上分析可发现，当ram\_cell地址长度不变，cache内组相联度增加时，所需硬件资源会较大幅度增加。同时将set addr,tag addr从3，6变为4，5时，硬件资源也会大幅增加。

计算矩阵乘法时,当组相联度从3变到4时，所需硬件资源增加，但缺失率和运行时间有明显的下降(fifo:53.68% -> 19.98%;lru:53.49% -> 17.77%)。从4到5时，所需硬件资源进一步增加，miss率和运行时间也有所下降，但没有3-4下降得明显。当参数为3454时，缺失率降低非常明显(set翻倍)，但所需硬件资源也大幅增加。综合考虑到硬件成本和运行时间，运行矩阵乘法时cache最佳参数应为3364。此时可在控制一定硬件成本的情况下使得缺失率也较低。当运行矩阵乘法时，当相连度为3，4时,lru策略稍优于fifo。相连度变为5后，fifo略优于lru。

进行256个数的快排时，缺失率一直都较低，基本可保持在2%以下。保持cache前三个参数为336，当相连度为3时，lru略优于fifo，相连度为4时，fifo略优于lru。相连度为5时，两者相当。当缺失率降到0.63%后就不再降低。综合考虑到硬件成本和运行时间，若保持cache前三个参数为336，则运行256个数的快排时cache最佳参数应为3364。此时可在控制一定硬件成本的情况下使得缺失率也较低。

所以，参数3364可能是一种较优的cache参数。

## 实验总结

本实验实现了两种策略cache,并连上cpu，在校验cache正确性的同时也校验了前面cpu的正确性。并通过分析不同参数cache的硬件资源、运行时间等确定较优的策略参数。

## 实验建议

test\_bench可以更精细化，比如若测试通过可以有一个类似lab2，三号寄存器为1的标志。方便同学们确认是否通过测试。