LSU

interface

input clk,

input reset,

input enable, //线程是否启用

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

// Memory Control Sgiansl

input decoded\_mem\_read\_enable // decoder输出的控制信号，指令为LDR时为高

input decoded\_mem\_write\_enable// decoder输出的控制信号，指令为STR时为高

// Registers

input [7:0] rs, // rs寄存器数值

input [7:0] rt, // rt寄存器数值

// Data Memory, 用于内存访问

output mem\_read\_valid, //内存读valid, 与内存握手通信

output [7:0] mem\_read\_address, //内存读地址

input mem\_read\_ready, //内存读ready, 与内存握手通信

input [7:0] mem\_read\_data, //内存读数据

output mem\_write\_valid, //内存写valid, 与内存握手通信

output [7:0] mem\_write\_address, //内存写地址

output [7:0] mem\_write\_data, //内存写数据

input mem\_write\_ready, //内存写ready, 与内存握手通信

// LSU Outputs

output [1:0] lsu\_state, //LSU执行状态

IDLE = 2'b00, //空闲

REQUESTING = 2'b01, //请求

WAITING = 2'b10, //等待

DONE = 2'b11; //完成

output [7:0] lsu\_out //从mem中读取到的数据

logic

lsu\_state

IDLE:

enable为高, 且decoded\_mem\_read\_enable或decoded\_mem\_write\_enable为高,

当core\_state == REQUEST时, 进入REQUESTING

REQUESTING:

读操作

拉高mem\_read\_valid

读地址设为rs数值

进入WAITING

写操作

拉高mem\_write\_valid

读地址设为rs数值

读数据设为rt数值

进入WAITING

WAITING:

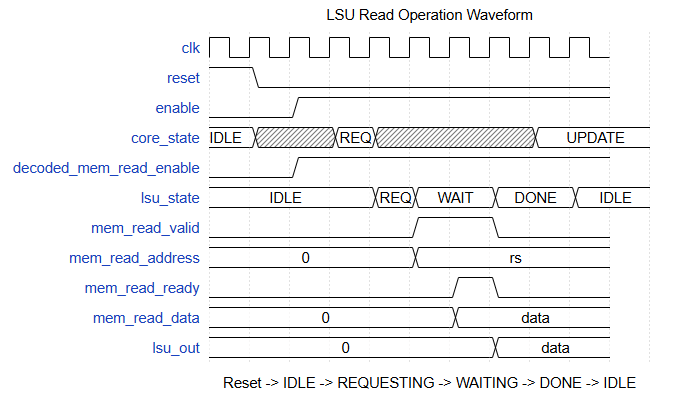
等待mem\_read\_ready, mem\_write\_ready

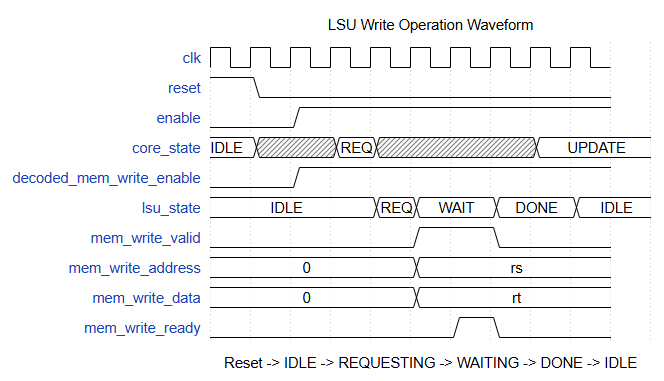
握手后,若为读操作,则将读数据赋值给lsu\_out,将对应valid信号拉低,进入DONE

DONE:

core\_state == UPDATE时, 返回IDLE空闲状态

wave





ALU

interface

input clk,

input reset,

input enable, //线程是否启用

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

input [1:0] decoded\_alu\_arithmetic\_mux, //decoder输出的控制信号, 选择计算功能

ADD = 2'b00,

SUB = 2'b01,

MUL = 2'b10,

DIV = 2'b11;

input decoded\_alu\_output\_mux, //decoder输出的控制信号, 为1时输出比较器结果, 为0时输出算术器结果

input [7:0] rs, //rs寄存器数值

input [7:0] rt, //rt寄存器数值

output [7:0] alu\_out //结果输出

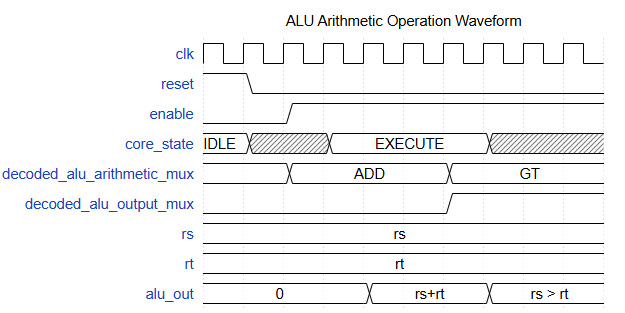
logic

enable为高, 当core\_state == EXECUTE时,

若decoded\_alu\_output\_mux == 1, 输出比较器结果,值为8’b: 0000\_0{rs>rt}{rs==rt}{rs<rt}

若decoded\_alu\_output\_mux == 0, 根据decoded\_alu\_arithmetic\_mux输出对应计算结果

wave



Register Files

interface

parameter THREADS\_PER\_BLOCK = 4,

parameter THREAD\_ID = 0,

parameter DATA\_BITS = 8

input clk,

input reset,

input enable, //线程是否启用

// Kernel Execution

input [7:0] block\_id, //blockidx, 保存在r[13]

// State

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

// Instruction Signals

input [3:0] decoded\_rd\_address, //decoder输出, rd编号

input [3:0] decoded\_rs\_address, //decoder输出, rs编号

input [3:0] decoded\_rt\_address, //decoder输出, rt编号

// Control Signals

input decoded\_reg\_write\_enable, //decoder输出, 允许写寄存器,CAL指令/LDR/CONST时为高

input [1:0] decoded\_reg\_input\_mux, //decoder输出, 选择写数据来源

ARITHMETIC = 2'b00,

MEMORY = 2'b01,

CONSTANT = 2'b10;

input [DATA\_BITS-1:0] decoded\_immediate, //decoder输出,指令为CONST时,对应立即数部分

// Thread Unit Outputs

input [DATA\_BITS-1:0] alu\_out, //alu输出,计算结果

input [DATA\_BITS-1:0] lsu\_out, //lsu输出,读取数据

// Registers

output [7:0] rs, //rs寄存器数值

output [7:0] rt //rt寄存器数值

logic

创建位宽为DATA\_BITS(8) bit, 大小为16(4bit地址)的寄存器阵列

reset阶段

r[14]存放blockdim

r[15]存放threadidx

其他寄存器值全为0

enable为高

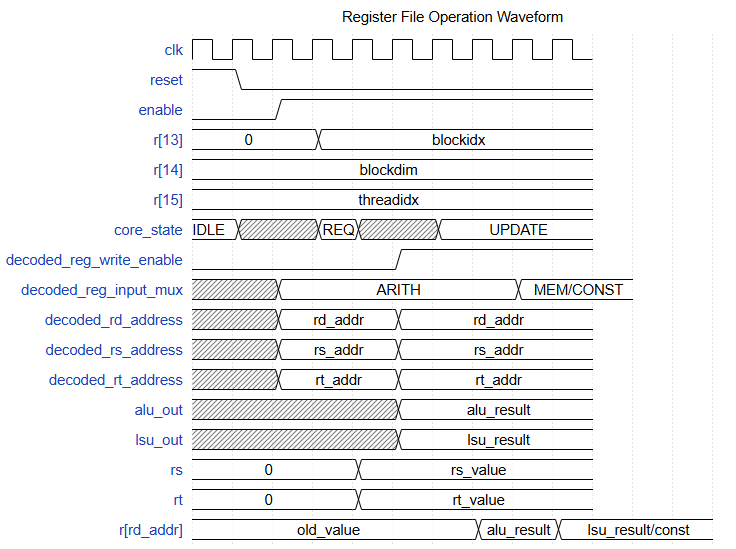
r[13]存放blockidx(应该由哪个core\_state执行??)

core\_state == REQUEST时, 从寄存器组中读取rs, rt寄存器数值

core\_state == UPDATE时, decoded\_reg\_write\_enable == 1且decoded\_rd\_address < 13时,

根据decoded\_reg\_input\_mux将对应数据写入寄存器

wave



PC

interface

parameter DATA\_MEM\_DATA\_BITS = 8, //数据位宽

parameter PROGRAM\_MEM\_ADDR\_BITS = 8 //地址位宽

input clk,

input reset,

input enable, //线程是否启用

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

// Control Signals

input [2:0] decoded\_nzp, //decoder输出, 指令为BRnzp时, 指令中nzp的值

input [DATA\_MEM\_DATA\_BITS-1:0] decoded\_immediate, //decoder输出, 指令为BRnzp时, 指令中立即数的值

input decoded\_nzp\_write\_enable, //decoder输出, 允许更改nzp寄存器, 指令为CMP时为高

input decoded\_pc\_mux, //decoder输出, 允许更改pc寄存器, 指令为BRnzp时为高

// ALU Output - used for alu\_out[2:0] to compare with NZP register

input [DATA\_MEM\_DATA\_BITS-1:0] alu\_out, //alu输出,指令为CMP使用

// Current & Next PCs

input [PROGRAM\_MEM\_ADDR\_BITS-1:0] current\_pc, //当前pc

output [PROGRAM\_MEM\_ADDR\_BITS-1:0] next\_pc //下一个pc(默认为pc+1)

logic

当enable为高时

core\_state == EXECUTE

若非BRnzp指令, next\_pc = pc + 1

若为BRnzp指令, 如果nzp寄存器与decoded\_nzp匹配(为1的位相同),

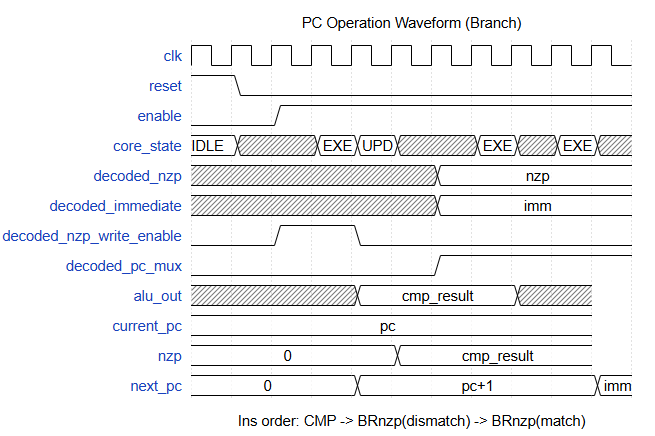
next\_pc = decoded\_immediate

否则next\_pc = pc + 1

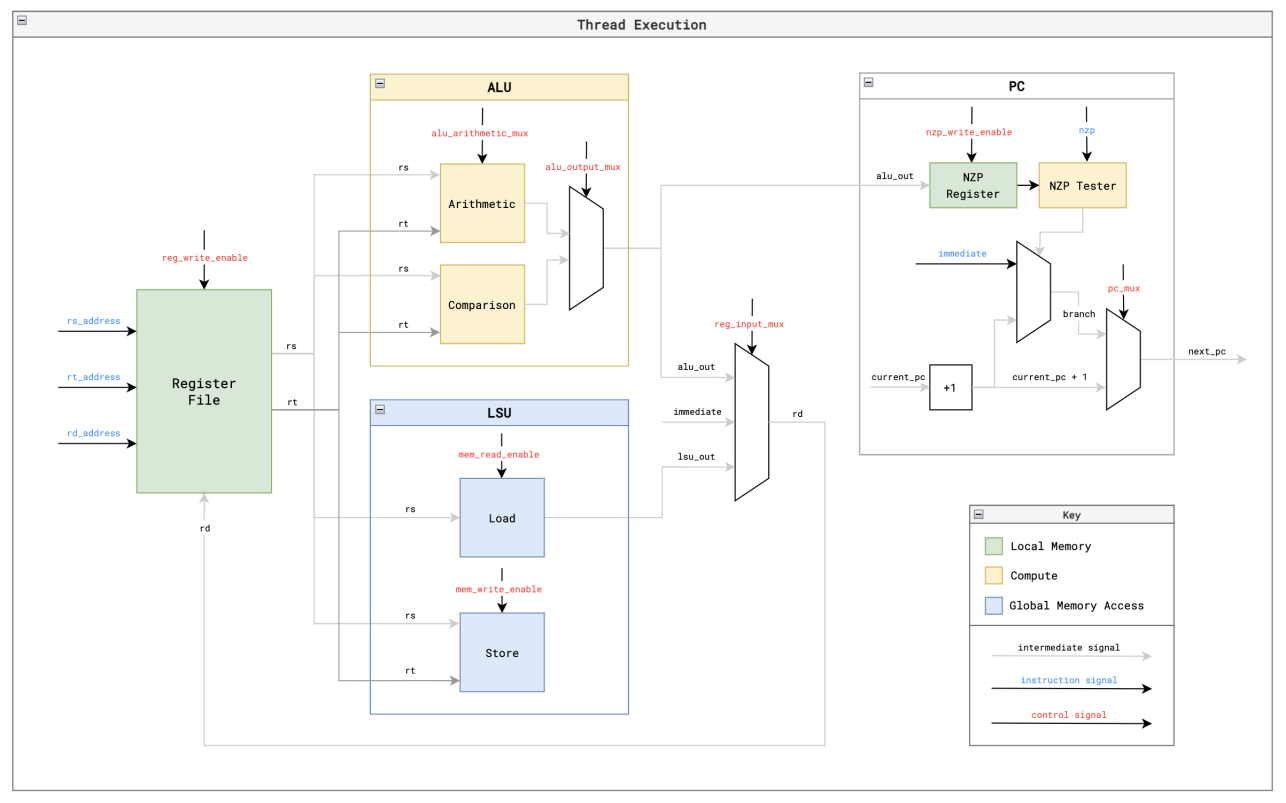
core\_state == UPDATE

若为CMP指令,则将alu\_out的低三位赋给nzp寄存器

wave



thread



decoder

interface

input clk,

input reset,

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

input [15:0] instruction, //需要译码的指令

instruction[15:12]:

NOP = 4'b0000,

BRnzp = 4'b0001,

CMP = 4'b0010,

ADD = 4'b0011,

SUB = 4'b0100,

MUL = 4'b0101,

DIV = 4'b0110,

LDR = 4'b0111,

STR = 4'b1000,

CONST = 4'b1001,

RET = 4'b1111;

// Instruction Signals

//registers

output [3:0] decoded\_rd\_address, //rd寄存器编号

output [3:0] decoded\_rs\_address, //rs寄存器编号

output [3:0] decoded\_rt\_address, //rt寄存器编号

//pc

output [2:0] decoded\_nzp, //BRnzp指令中指定的nzp的值

output [7:0] decoded\_immediate, //BRnzp指令指令的跳转地址

// Control Signals

//registers

output decoded\_reg\_write\_enable, //寄存器写使能

output [1:0] decoded\_reg\_input\_mux, //寄存器写数据来源

ARITHMETIC = 2'b00,

MEMORY = 2'b01,

CONSTANT = 2'b10;

//lsu

output decoded\_mem\_read\_enable, //LDR指令, 执行读内存

output decoded\_mem\_write\_enable, //STR指令, 执行写内存

//pc

output decoded\_nzp\_write\_enable, //CMP指令, 允许写nzp寄存器

output decoded\_pc\_mux, //BRnzp指令, 允许更改pc寄存器

//alu

output [1:0] decoded\_alu\_arithmetic\_mux, //选择计算功能

output decoded\_alu\_output\_mux, //选择输出比较器结果还是算术器结果

// Return (finished executing thread)

output decoded\_ret //RET指令, 线程执行完成

logic

reset阶段,所有输出置零

core\_state = DECODE时, 译码出所有的指令信号和控制信号

decoded\_rd\_address <= instruction[11:8];

decoded\_rs\_address <= instruction[7:4];

decoded\_rt\_address <= instruction[3:0];

decoded\_immediate <= instruction[7:0];

decoded\_nzp <= instruction[11:9];

// Control signals reset on every decode and set conditionally by instruction

decoded\_reg\_write\_enable <= 0;

decoded\_mem\_read\_enable <= 0;

decoded\_mem\_write\_enable <= 0;

decoded\_nzp\_write\_enable <= 0;

decoded\_reg\_input\_mux <= 0;

decoded\_alu\_arithmetic\_mux <= 0;

decoded\_alu\_output\_mux <= 0;

decoded\_pc\_mux <= 0;

decoded\_ret <= 0;

// Set the control signals for each instruction

case (instruction[15:12])

NOP: begin

// no-op

end

BRnzp: begin

decoded\_pc\_mux <= 1;

end

CMP: begin

decoded\_alu\_output\_mux <= 1;

decoded\_nzp\_write\_enable <= 1;

end

ADD: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b00;

decoded\_alu\_arithmetic\_mux <= 2'b00;

end

SUB: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b00;

decoded\_alu\_arithmetic\_mux <= 2'b01;

end

MUL: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b00;

decoded\_alu\_arithmetic\_mux <= 2'b10;

end

DIV: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b00;

decoded\_alu\_arithmetic\_mux <= 2'b11;

end

LDR: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b01;

decoded\_mem\_read\_enable <= 1;

end

STR: begin

decoded\_mem\_write\_enable <= 1;

end

CONST: begin

decoded\_reg\_write\_enable <= 1;

decoded\_reg\_input\_mux <= 2'b10;

end

RET: begin

decoded\_ret <= 1;

end

fetcher

interface

parameter PROGRAM\_MEM\_ADDR\_BITS = 8,

parameter PROGRAM\_MEM\_DATA\_BITS = 16

input clk,

input reset,

// Execution State

input [2:0] core\_state, //线程所属core的执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

input [7:0] current\_pc, //当前pc地址

// Program Memory

output mem\_read\_valid, //发起从mem中读取指令请求

output [PROGRAM\_MEM\_ADDR\_BITS-1:0] mem\_read\_address, //读取指令地址, =current\_pc

input mem\_read\_ready, //输入内存读ready,用于握手

input [PROGRAM\_MEM\_DATA\_BITS-1:0] mem\_read\_data, //输入内存读数据

// Fetcher Output

output [2:0] fetcher\_state, //当前fetcher状态

IDLE = 3'b000,

FETCHING = 3'b001,

FETCHED = 3'b010;

output [PROGRAM\_MEM\_DATA\_BITS-1:0] instruction //输出获取到的指令

logic

fetcher\_state == IDLE

当core\_state == FETCH时,拉高mem\_read\_valid,将current\_pc赋给mem\_read\_address,进入FETCHING状态

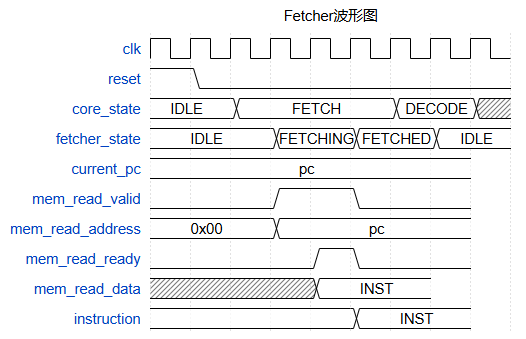
fetcher\_state == FETCHING

当接受到mem\_read\_ready为高时,握手成功,拉低mem\_read\_valid,将mem\_read\_data输出为instruction,进入FATCHED状态

fetcher\_state == FETCHED

当core\_state == DECODE时,回到IDLE状态

wave



scheduler

interface

parameter THREADS\_PER\_BLOCK = 4

input clk,

input reset,

input start, //输入启动信号

// Control Signals

input decoded\_mem\_read\_enable, //decoder输出的控制信号，指令为LDR时为高

input decoded\_mem\_write\_enable, //decoder输出的控制信号，指令为STR时为高

input decoded\_ret, //decoder输出的控制信号, 指令为RET时为高

// Memory Access State

input [2:0] fetcher\_state, //fetcher的执行状态

IDLE = 3'b000,

FETCHING = 3'b001,

FETCHED = 3'b010;

input [1:0] lsu\_state [THREADS\_PER\_BLOCK-1:0], //lsu的指令状态

IDLE = 2'b00,

REQUESTING = 2'b01,

WAITING = 2'b10,

DONE = 2'b11;

// Current & Next PC

output [7:0] current\_pc, //当前pc地址

input [7:0] next\_pc [THREADS\_PER\_BLOCK-1:0], //下一个pc地址

// Execution State

output [2:0] core\_state, //core执行状态

IDLE = 3'b000, // 空闲

FETCH = 3'b001, // 取值，从指令内存中获取指令

DECODE = 3'b010, // 从指令中译码出控制信号

REQUEST = 3'b011, // 从寄存器或内存中获取数据

WAIT = 3'b100, // 等待内存的响应

EXECUTE = 3'b101, // 执行ALU或PC计算

UPDATE = 3'b110, // 更新寄存器, NZP, and PC

DONE = 3'b111; // 完成该block中的执行

output done //完成信号

logic

reset阶段

current\_pc置为0, core\_state为IDLE, done为低

core\_state == IDLE

当start拉高时, 进入FETCH状态

core\_state == FETCH

当fetch\_state == FETCHED时,即完成取值,进入DECODE状态

core\_state == DECODE

decoder在该周期内完成译码,直接进入REQUEST状态

core\_state == REQUEST

直接进入WAIT状态

core\_state == WAIT

当所有lsu都不处于REQUESTING或WAITING状态时(enbale为高的lsu全部处于DONE状态)

进入EXECUTE状态

core\_state == EXECUTE

ALU在该周期完成计算,直接进入UPDATE阶段

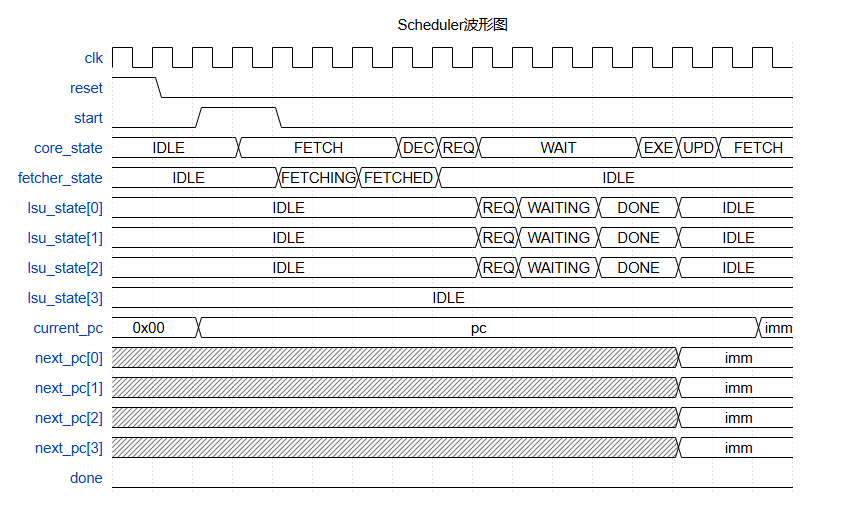
core\_state == UPDATE

该周期内完成寄存器, NZP, PC的更新

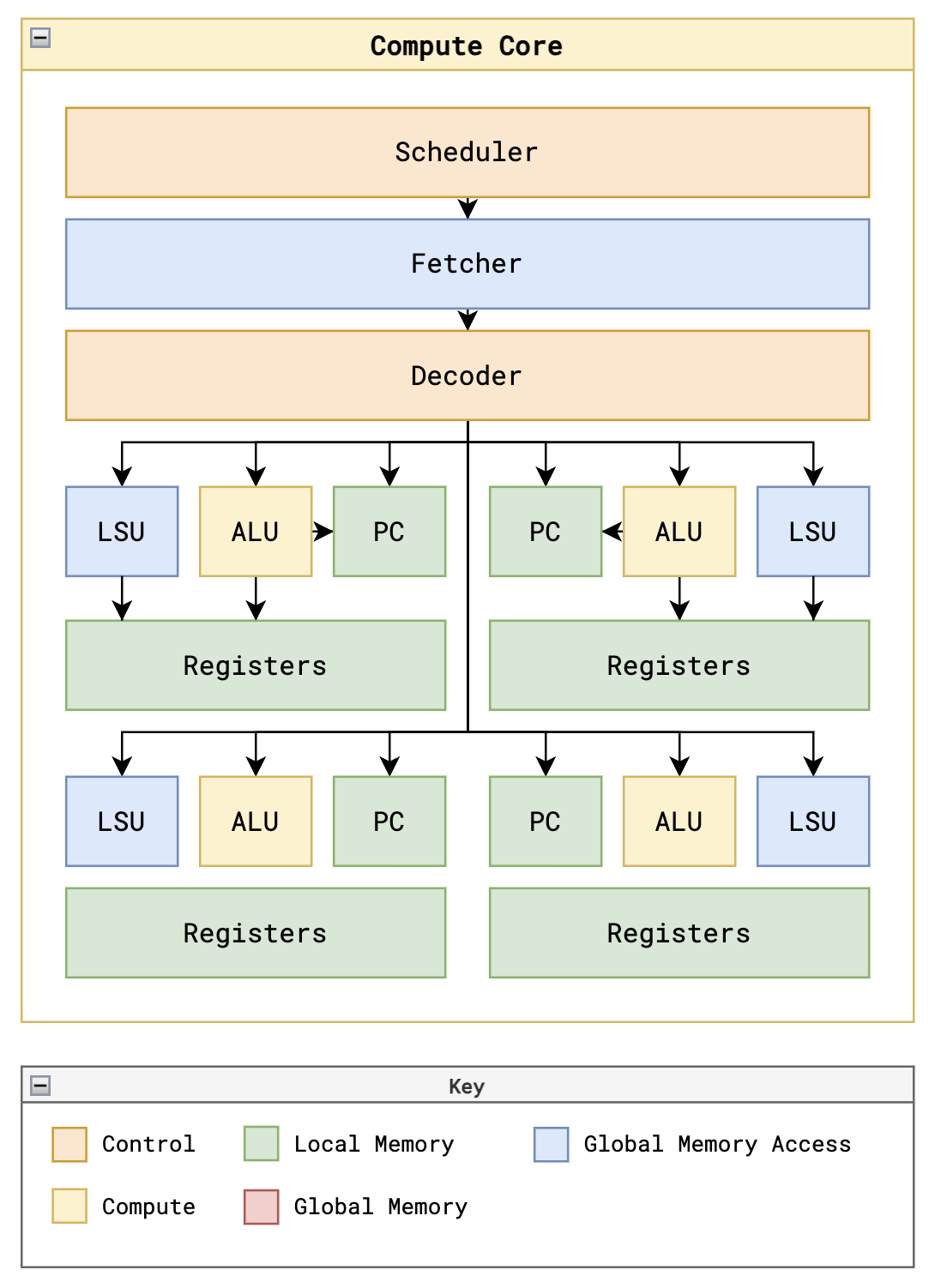
若为非RET指令,更新current\_pc为next\_pc,进入FETCH状态执行下一条指令

若为RET指令, 直接进入DONE,结束执行

wave



core



dispatch

interface

input wire clk,

input wire reset,

input wire start, //启动信号

// Kernel Metadata

input wire [7:0] thread\_count, //输入总线程数

// Core States

input reg [NUM\_CORES-1:0] core\_done, //core的完成信号

output reg [NUM\_CORES-1:0] core\_start, //core的启动信号

output reg [NUM\_CORES-1:0] core\_reset, //core的reset信号

output reg [7:0] core\_block\_id [NUM\_CORES-1:0], //core对应的blcokIdx

output reg [$clog2(THREADS\_PER\_BLOCK):0] core\_thread\_count [NUM\_CORES-1:0], //core内分配的线程数

// Kernel Execution

output reg done //完成信号

logic

total\_blocks = thread\_count // THREADS\_PER\_BLOCK //将线程平分至2个core中

reg [7:0] blocks\_dispatched; // 需要分发的block个数

reg [7:0] blocks\_done; // 执行完成的block个数

reg start\_execution; // 是否已经启动

reset阶段, core\_reset所有位拉高,其他信号置零

start信号为高时

若start\_execution == 0, 拉高start\_execution和core\_reset的所有位

若core\_reset为高,则

拉低core\_reset

若blocks\_dispatched < total\_blocks,

拉高core\_start

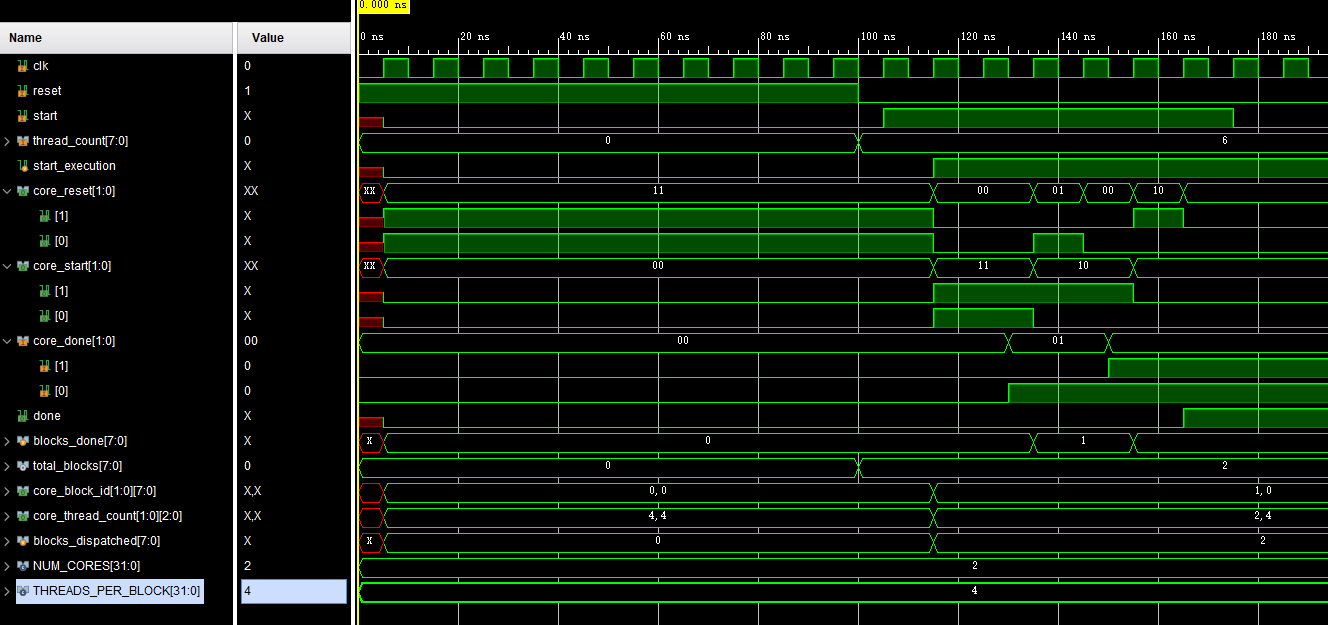
分配blockidx

分配core内有效线程数

若core\_done == 1, core\_start == 1, blocks\_done增加

若blocks\_done == total\_blocks, 所有core全部完成,则拉高done信号

wave



DCR

interface

input wire clk,

input wire reset,

input wire device\_control\_write\_enable,

input wire [7:0] device\_control\_data,

output wire [7:0] thread\_count

logic

device\_control\_write\_enable为高时

将device\_control\_data写入thread\_count寄存器

""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""""

controller(mem driver)

interface

parameter ADDR\_BITS = 8,

parameter DATA\_BITS = 16,

parameter NUM\_CONSUMERS = 4, // 向驱动请求访存的数量, data mem为NUM\_CORES \* THREADS\_PER\_BLOCK, program mem为NUM\_CORES

parameter NUM\_CHANNELS = 1, // 访问内存的通道数量

parameter WRITE\_ENABLE = 1 // 是否允许写内存, data mem为1, program mem为0

input wire clk,

input wire reset,

// Consumer Interface (Fetchers / LSUs)

input reg [NUM\_CONSUMERS-1:0] consumer\_read\_valid,

input reg [ADDR\_BITS-1:0] consumer\_read\_address [NUM\_CONSUMERS-1:0],

output reg [NUM\_CONSUMERS-1:0] consumer\_read\_ready,

output reg [DATA\_BITS-1:0] consumer\_read\_data [NUM\_CONSUMERS-1:0],

input reg [NUM\_CONSUMERS-1:0] consumer\_write\_valid,

input reg [ADDR\_BITS-1:0] consumer\_write\_address [NUM\_CONSUMERS-1:0],

input reg [DATA\_BITS-1:0] consumer\_write\_data [NUM\_CONSUMERS-1:0],

output reg [NUM\_CONSUMERS-1:0] consumer\_write\_ready,

// Memory Interface (Data / Program)

output reg [NUM\_CHANNELS-1:0] mem\_read\_valid,

output reg [ADDR\_BITS-1:0] mem\_read\_address [NUM\_CHANNELS-1:0],

input reg [NUM\_CHANNELS-1:0] mem\_read\_ready,

input reg [DATA\_BITS-1:0] mem\_read\_data [NUM\_CHANNELS-1:0],

output reg [NUM\_CHANNELS-1:0] mem\_write\_valid,

output reg [ADDR\_BITS-1:0] mem\_write\_address [NUM\_CHANNELS-1:0],

output reg [DATA\_BITS-1:0] mem\_write\_data [NUM\_CHANNELS-1:0],

input reg [NUM\_CHANNELS-1:0] mem\_write\_ready

//每个访问内存通道具有一个状态寄存器

reg [2:0] controller\_state [NUM\_CHANNELS-1:0];

IDLE = 3'b000,

READ\_WAITING = 3'b010,

WRITE\_WAITING = 3'b011,

READ\_RELAYING = 3'b100,

WRITE\_RELAYING = 3'b101;

reg [$clog2(NUM\_CONSUMERS)-1:0] current\_consumer [NUM\_CHANNELS-1:0]; // 每个通道在处理哪一个consumer的请求

reg [NUM\_CONSUMERS-1:0] channel\_serving\_consumer // 多个通道之间共享,确保多个通道不会处理请求

logic

reset阶段, 所有信号置为0

每个channel各自独立轮询执行

controller\_state == IDLE

轮询consumer查看是否有consumer发起read\_valid或write\_valid请求, 并且该请求还未被处理(consumer\_read\_valid[j] && !channel\_serving\_consumer[j])

优先编码, 如果轮询到有效请求,将channel\_serving\_consumer对应位拉高, 设置current\_consumer,并在该通道对内存发起访问

进入WAITING状态

controller\_state == READ\_WAITING/WRITE\_WAITING

当对应通道的内存ready信号为高时

拉低mem\_valid信号

拉高consumer\_ready信号,若为读则给consumer\_data赋值

进入RELAYING状态

controller\_state == READ\_RELAYING/WRITE\_RELAYING

当对应consumer的valid信号拉低后,

释放channel\_serving\_consumer

拉低consumer\_ready信号

进入IDLE状态