Block Ram

FPGA 내부 램 구조를 가진 block라 block ram이라고 부름

FPGA 데이터를 저장하는데 사용. 내부에 존재하기에 레이턴시가 낮음

Block ram은 유한한 크기를 가지며 width와 depth를 조절해 원하는 크기로 사용할 수 있음

텍스트, 폰트, 스크린샷, 화이트이(가) 표시된 사진

자동 생성된 설명 텍스트, 폰트, 화이트, 라인이(가) 표시된 사진

자동 생성된 설명

Width와 depth에 따라 address bus와 data bus의 신호개수가 정해진다

듀얼포트를 이용하며 각 포트별 비트를 다르게 정하면 A에서 6비트로 저장한 값을 B에서 8비트로 변환하는 방식도 가능하다.

텍스트, 폰트, 스크린샷, 영수증이(가) 표시된 사진

자동 생성된 설명

이 포트는 FIFO BRAM으로 비동기회로에서 사용, 서로 clock 주파수가 다른 두 모듈을 연결해줄 때 사용하는

Xilinx FPGA에서 BRAM은 최대 36 Kbits를 저장 이 메모리는 두 개의 18 Kb RAM 또는 한 개의 36 Kb RAM으로 구성가능

<https://nandland.com/lesson-15-what-is-a-block-ram-bram/>

<https://docs.amd.com/r/en-US/ug901-vivado-synthesis/Initializing-Block-RAM-Verilog>

<https://community.element14.com/challenges-projects/design-challenges/summer-of-fpga/b/blog/posts/number-plate-recognition-3-implementing-block-ram-using-verilog>

Addressing mode

1. immediate addressing

데이터가 명령어 내에 직접 포함된, 레지스터에 숫자 5를 더해라 (add R1, #5)

데이터의 크기가 명령어 크기로 인해 제한됨

2. Direct addressing

명령어에 메모리 주소가 포함된, 메모리 주소 1000에 저장된 값을 R1에 저장해라(load R1, 1000)

메모리에 접근하므로 느림, 주소가 명령어 크기로 인해 제한됨

3. indirect addressing

명령어가 지정한 메모리 주소에 데이터 주소가 저장되어 있는, 메모리 주소 1000에 저장된 주소를 통해 데이터를 불러와라(load R1, (1000))

메모리 접근하므로 느림, 단 명령어 크기 제한이 사라짐

4. register addressing

레지스터에 저장된 데이터를 참조해 연산하는, 레지스터 데이터간 연산 (add R1, R2)

레지스터 개수가 제한적

5. register indirect addressing

명령어에 적힌 레지스터의 데이터를 메모리 주소로 사용해 메모리 접근, R2 레지스터 데이터 값을 주소로 하여 메모리 접근해 데이터를 불러와라 (load R1, (R2))

메모리 접근으로 느림, 단 메모리 크기 제한 사라짐, 레지스터 개수 제한적인 것도 문제

6. displacement addressing

기본 주소와 변위를 합쳐 메모리 주소 생성하는, 기준 레지스터 주소지정은 레지스터 데이터와 변위 더해 실제 주소 구하는, 인덱스 주소 지정은 인덱스 레지스터 값과 변위를 더해 주소를 구하는, 기준 인덱스 주소 지정은 기준 레지스터 값과 인덱스 레지스터 값을 더하여 실제 주소를 계산. R2 데이터 값과 변위 1000을 더해 메모리 주소로 이용해라 (Load R1, 1000(R2))

주소 계산하려고 연산이 들어감, 단 메모리 접근이 더욱 강력해짐

7. relative addressing

프로그램 카운터인 pc 값을 기준으로 변위를 더해 주소를 구하는, pc에 대하여 얼마만큼 떨어진 위치로 점프할지를 정해주는 (jmp 100)

상대적 위치 기반으로 코드 위치가 바뀌면 값 변환 필요

module bram\_example (

input wire clk,

input wire we,

input wire [ADDR\_WIDTH-1:0] addr,

input wire [DATA\_WIDTH-1:0] din,

output reg [DATA\_WIDTH-1:0] dout

);

parameter ADDR\_WIDTH = 10;

parameter DATA\_WIDTH = 16;

// Define the memory array

reg [DATA\_WIDTH-1:0] bram [(2\*\*ADDR\_WIDTH)-1:0];

always @(posedge clk) begin

if (we) begin

bram[addr] <= din;

end

dout <= bram[addr];

end

endmodule

module testbench;

// Parameters

parameter ADDR\_WIDTH = 10;

parameter DATA\_WIDTH = 16;

// Signals

reg clk;

reg we;

reg [ADDR\_WIDTH-1:0] addr;

reg [DATA\_WIDTH-1:0] din;

wire [DATA\_WIDTH-1:0] dout;

// Instantiate the BRAM module

bram\_example #(.ADDR\_WIDTH(ADDR\_WIDTH), .DATA\_WIDTH(DATA\_WIDTH)) uut (

.clk(clk),

.we(we),

.addr(addr),

.din(din),

.dout(dout)

);

// Clock generation

always #5 clk = ~clk;

initial begin

// Initialize signals

clk = 0;

we = 0;

addr = 0;

din = 0;

// Write data to BRAM

#10;

we = 1;

addr = 10;

din = 16'hABCD;

#10;

we = 0;

// Read data from BRAM

#10;

addr = 10;

#10;

// Erase data by writing zeros

we = 1;

addr = 10;

din = 16'h0000;

#10;

we = 0;

// Read data again to confirm erase

#10;

addr = 10;

#10;

// Finish simulation

$stop;

end

endmodule

module bram\_controller (

input wire clk,

input wire we,

input wire [ADDR\_WIDTH-1:0] addr,

input wire [DATA\_WIDTH-1:0] din,

output reg [DATA\_WIDTH-1:0] dout,

input wire [1:0] mem\_select // 메모리 공간 선택: 00-캐시, 01-메인 메모리, 10-버퍼

);

parameter ADDR\_WIDTH = 12; // 총 4KB (2^12 바이트)

parameter DATA\_WIDTH = 16;

// 전체 4KB BRAM을 각 1KB씩 3개로 분할

reg [DATA\_WIDTH-1:0] bram\_cache [0:1023]; // 1KB for Cache

reg [DATA\_WIDTH-1:0] bram\_main\_memory [0:1023]; // 1KB for Main Memory

reg [DATA\_WIDTH-1:0] bram\_buffer [0:1023]; // 1KB for Buffer

always @(posedge clk) begin

case (mem\_select)

2'b00: begin // Cache

if (we) begin

bram\_cache[addr] <= din;

end

dout <= bram\_cache[addr];

end

2'b01: begin // Main Memory

if (we) begin

bram\_main\_memory[addr] <= din;

end

dout <= bram\_main\_memory[addr];

end

2'b10: begin // Buffer

if (we) begin

bram\_buffer[addr] <= din;

end

dout <= bram\_buffer[addr];

end

default: dout <= 16'hXXXX; // Invalid selection

endcase

end

endmodule