Hardware System Design, Lab 5

2013-11392 김지현

구현체 설명

BRAM

BRAM은 과제에서 시키는대로 BRAM_EN이 켜졌을 때 WE이 0일때엔 읽기를 수행하고, WE가 non-zero일 경우 쓰기를 수행하면 된다. 그러나 쓰기에 1 cycle이 소모되고, 읽기에 2 cycle이 소모되도록 구현해야한다는 지점 때문에 구현이 까다로워진다.

나는 Cycle을 구현하기 위해 BRAM 안에 읽기 큐, 쓰기 큐 두개를 유지하는 방식으로 구현하였다. BRAM 안에 작은 FSM을 하나 구현한것과 개념적으로는 동일한데, 프로그래밍하기 보다 쉬운 형태를 택했다.

```
reg [31:0] read_q[0:1];

/* ... */

always @(posedge BRAM_CLK) begin
    /* ... */

    // Process delayed read: Read mem[addr] into BRAM_RDDATA
    BRAM_RDDATA = read_q[0];

    /* ... */

    // Advance read queue
    read_q[0] = read_q[1];

    /* ... */

    // Enqueue new command if BRAM_EN is true
    if (BRAM_EN) begin
        if (BRAM_WE == 0) begin
            read_q[1] = mem[addr];
        end else begin
            /* ... */
        end
    end
end
```

Figure 1. Simplified view of how `read_q` work

Figure 1는 my_bram에서 읽기 큐와 관련된 부분만을 남기고 나머지 코드를 생략한것이다. 읽기 요청이 들어오면 해당 요청을 바로 처리하는 대신, 일단 read_q에 enqueue시킨다. 그리고 매 Clock Rising Edge마다 읽기 큐는 한칸씩 Advance하며, 제일 오래된 entry가 밀려난다. 밀려나게된 큐의 제일 오래된 entry는 없어지기 전에 실행된다.

과제 스펙에 읽기요청과 쓰기요청이 뒤섞이는 경우 어떻게 동작할지 명시되어있지는 않았으나, 읽기는 커맨드가 enqueue된 시점의 데이터를 2 cycle 뒤에 반환하도록 구현하였고, 쓰기는 커맨드가 dequeue된 시점에 발효되도록 구현하였다.

PE

PE는 Lab 4에서 사용했던 Floating point FMA를 사용해 구현하면 된다. 다만 한가지 처리가 까다로웠던 점이, FMA의 입력이 출력으로 다시 들어가는 부분의 구현이었다.

내 과제에서는 psum이라는 레지스터를 사용해 accumulate된 결과를 기록하고있다. 처음에는 단순히 FMA의 결과에 sensitive한 always block을 만들어 psum 레지스터를 update시켜줬으나, 실험결과 FMA의 두 출력인 valid와 result가 update되는 시점이 일치하지 못하다는 문제가 있어서 psum에 쓰레기값이 잘못 대입되는 문제가 발생했다.

```
wire [31:0] result;
reg [31:0] psum;

floating_point_0 FMA(
    /* ... */
    .s_axis_c_tdata(psum),
    .m_axis_result_tdata(result)
);

/* ... */

always @(negedge aclk) begin
    if (result_valid) begin
        // Accumulate
        psum = result;
    end else begin
        /* ... */
    end
end
```

Figure 2. Simplified view of how psum work

이런 미세한 타이밍 문제에서 아예 벗어나기 위해, Falling Clock Edge에서 FMA의 output valid 값이 true일경우 psum 레지스터를 업데이트하는 방식으로 구현을 고쳤다.

실험 결과 분석

BRAM

Synthesis, Implement 모두 가능했고, Waveform은 아래와 같았다. 5ns 시점에 BRAM1에 첫 읽기 요청이 발생하는데, 15ns 시점에 “result1”의 값이 변하면서 과제의 요구사항인 2사이클 뒤에 읽기처리가 끝나는 것을 알 수 있다. 과제가 요구하는 “my_bram”의 입출력 형태상 쓰기가 1사이클 뒤에 일어났다는 것은 waveform으로 확인할 수 없는데, 이는 “$display” 명령어를 사용해 별도로 확인해야만 했다.

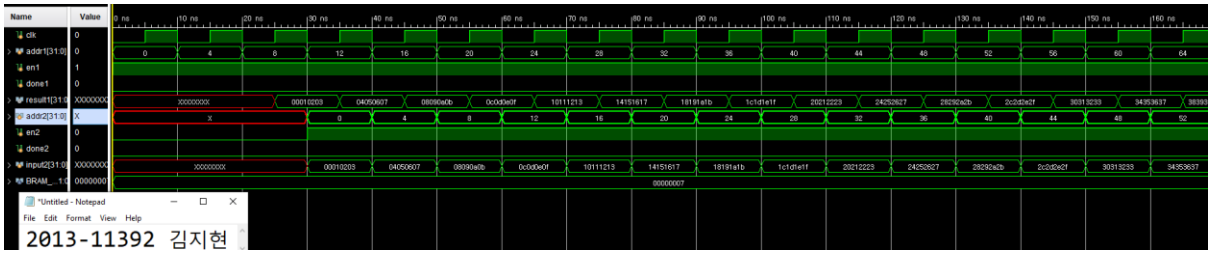

Figure 3. Waveform of my_bram_test.v

또한 실행이 끝난 뒤 “lab-05.sim/sim_1/behav/xsim” 디렉토리를 확인하여 output1.txt, output2.txt 가 “00010203 …”의 꼴로 같은 값을 갖고있어, 과제에서 요구하는 테스트 시나리오인 BRAM1에서 BRAM2로의 값 복사가 정상적으로 이뤄졌다는 것을 확인할 수 있었다.

PE

역시 Synthesis, Implement 모두 성공했고, Waveform은 아래와 같았다. PE의 경우 floating point unit의 계산 delay때문에 시뮬레이션 시간 제한을 기본값보다 늘려야 16번의 덧셈이 모두 정상적으로 이뤄진다는 것을 확인할 수 있었다.

Waveform 파일의 크기가 너무 커서 보고서에는 정상적인 형태로 첨부하지 못했으니, 압축 파일의 “pe.png”를 확인하면 값이 올바르게 계산되었다는 것을 확인할 수 있다.

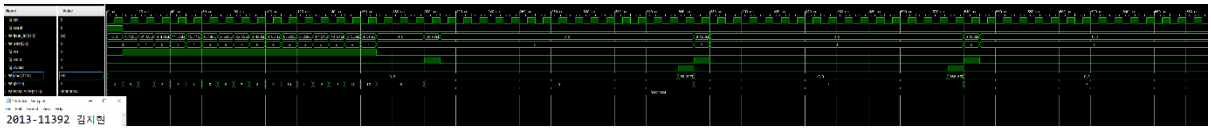

Figure 4. Waveform of my_pe_test.v

10ns부터 170ns까지는 peram에 16개의 숫자를 메모리에 입력시키는 과정이고, 200ns가 처음으로 계산을 수행하는 시점이다. 205ns에 입력된 4.125*20.69가 365ns에 85.39로 정상적으로 계산되는 것을 볼 수 있고, 375ns에 입력된 85.39 + 31.85*9.520이 535ns에 388.6으로 정상적으로 계산되는 것을 볼 수 있다. 또한 올바른 계산 결과를 내고있을때에만 dvalid가 1로 설정되고, dvalid가 0일때엔 항상 dout가 0으로 설정되어, 과제의 요구조건을 만족한다.

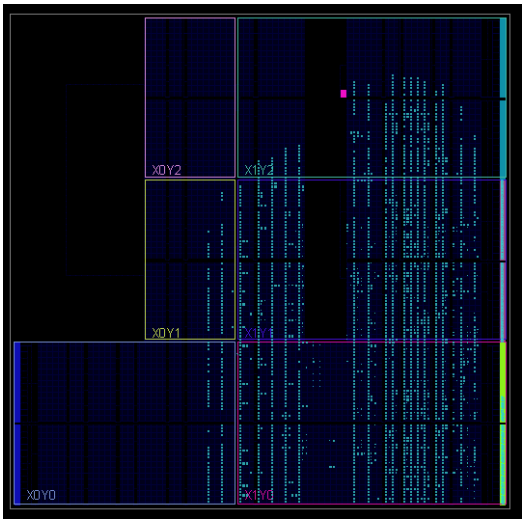

Figure 5. BRAM

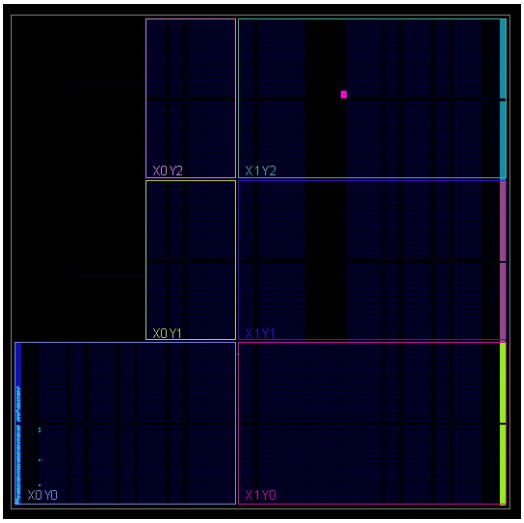

Figure 6. PE

Figure 5와 6은 각각 BRAM과 PE의 implement한 결과이다. BRAM이 사용하는 레지스터 수가 많아 많은 영역이 표시되어있는 것을 볼 수 있다.

결론

간단한 BRAM을 직접 만들어보면서, BRAM의 읽고 쓰기에 딜레이가 있다는 사실을 알 수 있었고, work 단위로만 읽고 쓸 수 있는 BRAM이 어떻게 write mask를 사용해 1바이트 쓰기와 같은 동작을 지원할 수 있는지 직접 만들면서 익힐 수 있었다.

PE를 만들면서는 딜레이가 긴 IP들을 어떻게 조합하여 사용하는지 익힐 수 있었고, 타이밍 이슈를 어떻게 해결하는지도 실습으로 익힐 수 있었다. 본격적으로 Floating point 숫자를 계산하며 Final Project의 PE를 어떻게 만들지 감을 잡을 수 있었던 실습이었다.