Hardware System Design, Lab 6

2013-11392 김지현

개요

이번 과제는 Lab 5에서 만든 Processing Element에 타이밍에 맞춰 데이터를 넣어주는 Processing Element Controller를 제작하는것이다. Lab 5에서 만든 PE는 MAC 연산만을 수행할 수 있는데, 여기에 적절한 Controller를 붙여주면 MAC을 벡터 내적기로 확장시킬 수 있다. 벡터 내적기를 구현하고, 테스트하는것이 이번 과제의 목표이다.

구현체 설명

PE

기본적으로 PE는 Lab 5 의 것을 그대로 사용하였으나, Lab 5에서 만들었던 PE에서 reset 관련 처리가 미비하여 이부분을 보강시켰다. Lab 5에선 reset 입력이 구현되어있지 않아도 큰 문제가 없으나, 이번 과제에선 S_DONE state에서 S_IDLE state로 돌아오는 부분때문에 제대로된 reset 구현이 필요했다.

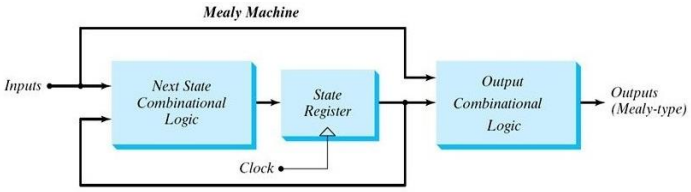

Figure 1. Overview of Mealy FSM

Overview of PE Controller

Lab 5의 PE Controller는 하나의 거대한 Mealy FSM으로 볼 수 있다. PE controller의 input과 PE의 output이 이 FSM의 입력이며, PE controller의 output과 PE의 input이 이 FSM의 output이라고 볼 수 있다. 이 FSM을 한번에 구현하면 너무 복잡하다. 이 때문에 나는 Figure 1과 같이 이 FSM을 Next state를 알아내는 조합논리와, Output을 결정하는 조합논리 크게 두 부분으로 나누어 개발하였다.

PE Controller, definition of state

과제에서 state로 S_IDLE, S_LOAD, S_CALC, S_DONE 이 네가지를 사용하라고 명시적으로 요구하였으나, 실제로는 한 state 안에서 고정된 횟수만큼 특정 연산을 반복하는 등의 처리가 필요하기 때문에, state가 실제로 저 네개가 전부는 아니다. 이 때문에 과제에선 counter 등을 state와 함께

사용하였는데, 이 과제에선 counter를 별도로 만들지는 않았고 state 변수 안에 counter를 내장시켜 구현하였다.

```
//
// FSM
//
reg [L_RAM_SIZE+2:0] state;
// S_IDLE               : state == 0
// S_LOAD (peram)       : 1 <= state < 1 + (1<<L_RAM_SIZE)
//                        counter = state - 1
localparam S_LOAD_peram_bound = 1 + (1<<L_RAM_SIZE);
// S_LOAD (global_dram) : 1 + (1<<L_RAM_SIZE) <= state < 1 + (1<<(L_RAM_SIZE+1))
//                        counter = state - S_LOAD_peram_bound
localparam S_LOAD_global_dram_bound = 1 + (1<<(L_RAM_SIZE+1));
// S_CALC               : 1 + (1<<(L_RAM_SIZE+1)) <= state < 1 + (1<<(L_RAM_SIZE+2))
//                        counter = (state - S_LOAD_global_dram_bound)>>1
localparam S_CALC_bound = 1 + (1<<(L_RAM_SIZE+2));
// S_DONE               : 1 + (1<<(L_RAM_SIZE+2)) <= state < 6 + (1<<(L_RAM_SIZE+2))
//                        counter = state - S_CALC_bound
```

Figure 2. How I encoded the "state"

이 과제에선 state를 N+3 비트 정수로 정의하여, state를 아래와 같이 정의하였다. 아래와 같이 하나의 S_LOAD, S_CALC, S_DONE state를 여러 개로 나누면 counter를 사용할 필요가 없어지고, state 변수 하나로 PE controller의 모든 상태를 관리할 수 있어진다.

- If state = 0, PE is in S_IDLE state
- If $1 \le state < 1 + 2^N$, PE is in S_LOAD (peram) state
  - read address "state – 1" and store it into "peram[state – 1]"
- If $1 + 2^N \le state < 1 + 2^{N+1}$, PE is in S_LOAD (global_dram) state
  - read address "state – 1" and store it into "global_dram[state – 1 - 2^N]"
- If $1 + 2^{N+1} \le state < 1 + 2^{N+2}$, PE is in S_CALC state
  - If "state – 1 - 2^{N+1}" is even, calculate "peram[(state – 1 - 2^{N+1})/2] × global_dram[(state – 1 - 2^{N+1})/2]"
  - Otherwise, wait until previous MAC is done
- If $1 + 2^{N+2} \le state < 6 + 2^{N+2}$, PE is in S_DONE state
  - After "6 + 2^{N+2} – state" cycle, S_DONE state will be finished

PE Controller, "next state" combination logic

Figure 2와 같이 상태를 정의하고나면, next state를 찾는 조합논리는 생김새가 비교적 간단하다.

- On S_IDLE: start가 1일때에만 state += 1
- On S_LOAD: 항상 state += 1
- On S_CALC:
  - If "state – 1 - 2^{N+1}" is even, state += 1
  - Otherwise, MAC 계산이 끝난 경우에만 state += 1
- On S_DONE:
  - If "state = 5 + 2^{N+2}", **state = 0** (S_IDLE로 돌아감)
  - Otherwise, state += 1

이와 같이 state를 진행이 가능한 경우에만 1씩 더해지고, 진행이 불가능한경우엔 멈춰있는 counter로 볼 수 있다. 위의 로직에다가 reset이 활성화되어있을 경우 state를 0으로 만드는 부분만 추가하면 과제의 next() 함수가 완성된다.

```
wire [L_RAM_SIZE+2:0] next_state;
assign next_state = next(state, aresetn, start, pe_ready);
function [L_RAM_SIZE+2:0] next(input [L_RAM_SIZE+2:0] state, input aresetn, start, pe_ready);
    if (!aresetn) next = 0;
    else begin
        /* S_IDLE */ if (state == 0) next = start;
        /* S_LOAD */ else if (state < S_LOAD_global_dram_bound) next = state + 1;
        /* S_CALC */ else if (state < S_CALC_bound) begin
            if (!((state - S_LOAD_global_dram_bound)&1)) next = state + 1;
            else next = state + pe_ready;
        end
        /* S_DONE */ else begin
            if (state - S_CALC_bound < 4) next = state + 1;
            else next = 0;
        end
    end
endfunction
```

Figure 3. Simplified code of "next state" logic

PE Controller, “output” combinational logic of rising edge

상태와 입력이 주어졌을 때 어떻게 행동할지 결정하는 조합논리를 아래와 같이 작성하면 된다.

- On S_IDLE: PE가 아무 일도 하지 않도록 설정
  - pe_we = 0, pe_valid = 0
- On S_LOAD (peram): 입력으로 들어오는 데이터가 peram에 저장되도록
  - pe_din = rddata, pe_addr = state – 1, pe_we = 1, pe_valid = 0
- On S_LOAD (global dram): 입력으로 들어오는 데이터가 global DRAM에 저장되도록
  - pe_we = 0, pe_valid = 0
  - global_bram[state - 1 - 2^N] = rddata
- On S_CALC:
  - If “state – 1 - 2^{N+1}” is even, “peram[(state – 1 - 2^{N+1})/2] × global_dram[(state – 1 - 2^{N+1})/2]” 계산 수행
    - pe_ain = global_dram[(state – 1 - 2^{N+1})/2]
    - pe_addr = (state – 1 - 2^{N+1})/2
    - pe_we = 0, **pe_valid = 1**
  - Otherwise, MAC 계산이 끝날 때 까지 아무것도 하지 않음
    - pe_we = 0, pe_valid = 0

PE Controller, “output” combinational logic of falling edge

위에서 하나 특기할 점은 rdaddr을 결정하는 코드가 위에 존재하지 않는다는 점이다. PE Controller에서 rdaddr의 역할은, PE Controller의 사용자가 정의한 메모리를 PE Controller에 연결시키도록 하여, PE Controller가 rdaddr의 값을 바꿀 때마다 PE Controller가 읽는 메모리 값도 자동으로 바뀌도록 만들어 PE Controller가 원하는 메모리 영역을 자유 자재로 읽을 수 있게 하기 위함이다.

이 때 rdaddr를 바꿀 경우, rdaddr의 변화가 rddata에 반영되기까지 시간이 필요하여서, rddata를 바꾼 시점에선 rdaddr를 즉시 사용해선 안된다. 이를 해결하기 위해선 rdaddr를 수정한 다음 클락 사이클에 rddata를 사용하는 등의 방식으로 rdaddr과 rddata를 사용하는 시점 사이에 차이를 두어야한다.

```
//
// Output (falling edge)
//
always @(negedge aclk) begin
    if (0 < state && state < S_LOAD_global_dram_bound) begin
        // S_LOAD
        rdaddr = state - 1;
    end
end
```

Figure 4. Updating rdaddr at falling clock edge

본 과제에선 rdaddr를 Rising Clock Edge가 아니라 rddata를 사용하는 시점보다 반 클락 앞선 Falling Clock Edge 시점에 업데이트하는 방식으로 이러한 타이밍 문제에서 벗어났다. 이를 코드로 표현하면 Figure 4와 같다.

PE Controller, 그 외 주의할 점

여기까지 구현하면 대부분의 로직이 완성되는데, 아래 부분을 놓치지 않도록 주의해야한다.

```
assign done = S_CALC_bound ≤ state;
```

Figure 5. Implementing "done" output

- 매 Rising Clock Edge에서의 Output 처리 이후에 state가 nextstate로 진행되도록 할 것
- “done” output 구현. Output에서 done 출력을 구현해도 되지만, done은 단순히 state가 S_DONE일경우 1이고, 아닐경우 0을 뱉는 핀이기 때문에 Figure 5와 같이 조합논리로 구현해도 된다.

```
// pe_ready: Is PE ready for next MAC input?
reg pe_ready;
always @(negedge aclk) begin
    pe_ready = pe_dvalid;
end
```

Figure 6. "pe_ready" register

- MAC 계산의 완료 여부를 PE의 dvalid 핀에서 바로 읽는 방식으로 구현할 경우, PE의 dvalid값도 rising edge에 업데이트되고 PE의 dvalid값을 읽는 시점도 rising edge이기 때문에 타이밍 문제가 발생할 수 있다. Figure 6과 같이 레지스터를 하나 만들어야 한다.
- S_IDLE state일경우 PE가 reset되도록 구현해야한다.

실험 결과 분석

S_IDLE state부터, S_LOAD state까지의 waveform, S_CALC state부터 S_DONE state까지의 waveform을 각각 Figure 7과 Figure 8과 같이 확인할 수 있었다.

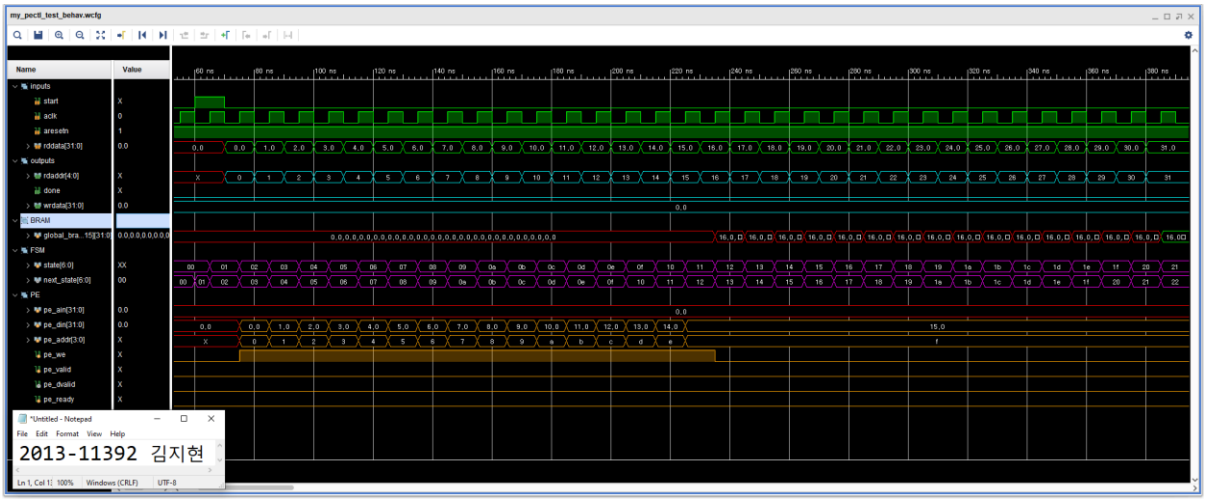

Figure 7. S_IDLE state부터, S_LOAD state까지의 waveform

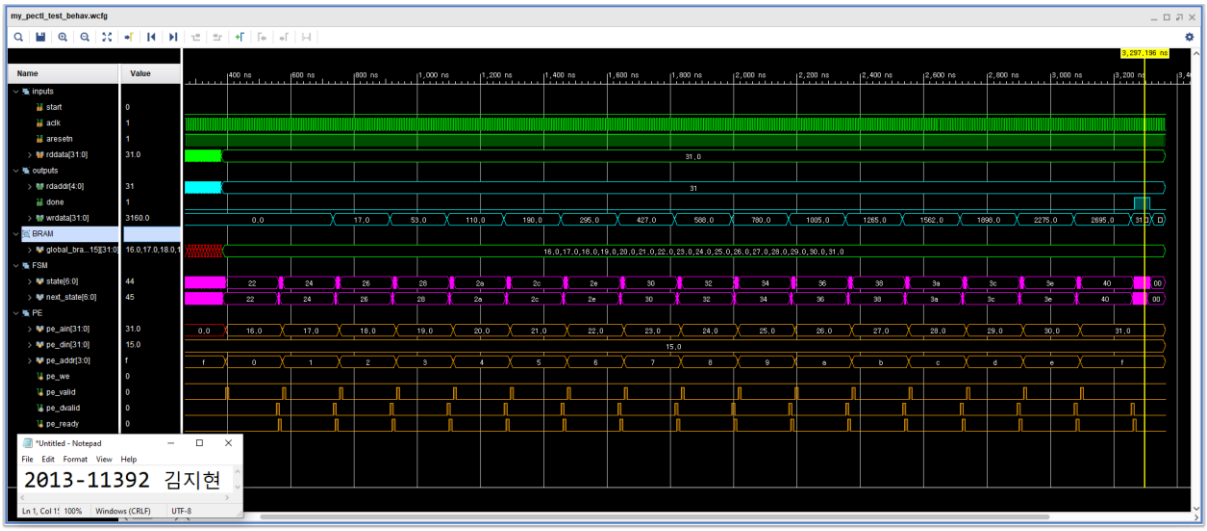

Figure 8. S_CALC state부터 S_DONE state까지의 waveform

동작이 몹시 복잡해보일 수 있으나, PE Controller를 사용하는 testbench의 입장에선 사용법이 몹시 간단하다. 데이터가 적절하게 세팅된 메모리를 준비하고 이를 PE Controller과 연결시킨 뒤, start에 신호만 넣으면, 이후부터는 dvalid가 1이 될 때 까지 기다리기만 하면 된다.

결론

지금까지 만들었던 FMA 회로나 MAC 회로와는 다르게, 벡터 내적이라는 훨씬 흔히 쓰이는 연산을 Verilog로 작성하는것이어서, 훨씬 재밌었고 하드웨어 가속기가 어떤 형태로 동작하는지 훨씬 쉽게 가늠할 수 있는 과제였다. 그리고 비록 지금은 벡터 내적 계산기이지만, 약간만 확장하면 벡터*행렬 곱셈기로 확장할 수 있는 형태이기 때문에, Final project가 어떤 형태일지 가늠할 수 있었다. 미래의 과제에서 이 PE를 소프트웨어에 직접 연결하는 때가 기대된다.