

Project 4: Problem 1

Retired instruction 수와 실행 cycle 및 CPI

Retired instruction 수와 실행 cycle은 `riscv_HPC.v` 파일의 `HPC_req_retired` 및 `HPC_exe_cycle` 를 통해 확인할 수 있다.

실제로 그 값을 확인한 결과는 다음과 같다.

HPC_req_retired	5176	5175	5176				
HPC_exe_cycle	7681	7680	7681				

마지막 instruction이 retire될 때의 cycle 수는 7681이었으며, 총 retire된 instruction의 수는 5176이었다. 따라서 총 CPI는 다음과 같이 계산된다.

$$CPI = \frac{Clock\ cycles}{Instruction\ Count} = \frac{7681}{5176} = 1.48396$$

Program 시작 후 종료까지, compute-bound instruction (division)의 수와 이에 따른 stall cycle 수 및 CPI 증가값

먼저 계산과정을 설명하겠다. 다음과 같은 코드를 통해 stall을 발생시키는 compute-bound instruction (division)의 수를 계산하였다.

```
wire stall_sig_rising_edge_ALU;
assign stall_sig_rising_edge_ALU = stall_by_ALU;
reg stall_sig_rising_edge_r_ALU;
always @(posedge rst_i or posedge clk) begin
    if (rst_i)
        stall_sig_rising_edge_r_ALU <= 1'b0;
    else
        stall_sig_rising_edge_r_ALU <= stall_sig_rising_edge_ALU;
end
wire stall_ALU_cause_sig;
assign stall_ALU_cause_sig = (stall_sig_rising_edge_ALU==1'b1)&&(stall_sig_rising_edge_r_ALU==1'b0);

always @(posedge rst_i or posedge clk) begin
    if (rst_i)
        HPC_req_ALU <= 'b0;
    else if(stall_ALU_cause_sig)
        HPC_req_ALU <= HPC_req_ALU + 1'b1;
end
```

간단하게 설명하자면, `stall_by_ALU` 를 wire로 할당하고, 이를 latch시켜서 stall signal이 발생하는 시점을 포착하여 count 했다. 이때 해당 코드가 valid한지 확인하기 위해서는 `stall_by_ALU` 가 정말 ALU에 의한 stall 여부를 의미하는 변수인지 확인할 필요가 있다. 이는 `riscv_pipe_ctrl.v` 에서 다음과 같은 코드를 통해 확인할 수 있었다. Execute stage에서 R-type instruction의 division 연산을 수행하는 경우에 data hazard가 발생하기 때문에 div 연산을 기준으로 stall 발생 여부를 확인하고 있다.

```
ctrl_e1_q[PCINFO_DIV] <= issue_div_i & ~take_interrupt_i;
```

```
504 assign HPC_stall_by_ALU = (ctrl_e1_q[PCINFO_DIV] && ~div_complete_i);
505
```

```
531 .stall_by_ALU (HPC_stall_by_ALU),
```

그 결과, 이는 4의 값을 가짐을 확인할 수 있었다. 이는 주석처리된 다음의 코드를 통해 전체 div instruction 수임을 확인할 수 있다. 즉 모든 div instruction은 stall을 발생했다.

```
always @(posedge rst_i or posedge clk) begin
    if (rst_i) HPC_req_ALU <='b0;
    else if(req_inst_div) HPC_req_ALU <= HPC_req_ALU + 1'b1;
end
```

다음으로 아래의 코드를 통해 compute-bound instruction (division)에 따른 stall cycle 수를 계산하였다.

```
// wire stall_sig_rising_edge_ALU;
// assign stall_sig_rising_edge_ALU = stall_by_ALU;

always @(posedge rst_i or posedge clk) begin
    if (rst_i) HPC_req_ALU_stall_cycle <='b0;
    else if(stall_sig_rising_edge_ALU) HPC_req_ALU_stall_cycle <= HPC_req_ALU_stall_cycle + 1'b1;
end
```

위의 코드를 통해 앞서 `riscv_pipe_ctrl.v`의 코드를 직접 살펴봄으로써, `stall_by_ALU`가 의미하는 바를 정확히 파악하였으므로, 이는 valid한 코드임을 알 수 있다.

그 결과, 이는 132의 값을 가짐을 확인할 수 있었다.

정리하면, 총 4개의 compute-bound instruction (division)이 총 132 cycle의 stall을 발생시켰다. 이에 의한 CPI 증가값을 계산했고, 그 결과는 다음과 같다.

- 총 CPI : 1.4840
- 해당 instruction에 의한 stall이 발생하지 않은 이상적인 경우의 CPI : $\frac{7681-132}{5176} = \frac{7549}{5172} = 1.4585$
- CPI 증가값 : $1.4840 - 1.4585 = 0.0255$

결과에 대한 파형은 다음과 같이 확인할 수 있다.

HPC_exe_cycle	7685	7685
HPC_req_ALU	4	4
HPC_req_ALU_stall_cycle	132	132
HPC_req_MEM	1588	1588

Program 시작 후 종료까지, memory-bound instruction (store, load instruction) 수와 이에 따른 stall cycle 수 및 CPI 증가값

마찬가지로 계산과정을 설명하겠다. 다음과 같은 코드를 통해 stall이 발생한 memory-bound instruction (store, load instruction)의 수를 계산하였다.

```
wire stall_sig_rising_edge_MEM;
assign stall_sig_rising_edge_MEM = stall_by_MEM;
reg stall_sig_rising_edge_r_MEM;
always @(posedge rst_i or posedge clk) begin
    if (rst_i) stall_sig_rising_edge_r_MEM <=1'b0;
```

```

        else
            stall_sig_rising_edge_r_MEM <=stall_sig_rising_edge_MEM;
        end
        wire stall_MEM_cause_sig;
        assign stall_MEM_cause_sig = (stall_sig_rising_edge_MEM==1'b1)&&(stall_sig_rising_edge_r_MEM==1'b0);

        always @(posedge rst_i or posedge clk) begin
            if (rst_i)
                HPC_req_MEM_cause_stall <='b0;
            else if(stall_MEM_cause_sig)
                HPC_req_MEM_cause_stall <= HPC_req_MEM_cause_stall + 1'b1;
        end
    end

```

위는 앞서 살펴본 코드와 동일한 구조를 가지기 때문에, 코드에 대한 자세한 설명은 생략한다. 다만 이전에 `stall_by_ALU` 의 코드를 직접 살펴보았듯이 `stall_by_MEM` 의 코드를 직접 `riscv_pipe_ctrl.v` 에서 확인함으로써 검증할 수 있었다. (Memory stage에서 I-type instruction의 load, store에 대한 cache miss가 발생할 경우 stall이 발생함) 이는 다음과 같다.

```

ctrl_e1_q[PCINFO_LOAD] <= issue_lsu_i & issue_rd_valid_i & ~take_interrupt_i; // TODO: Check
ctrl_e1_q[PCINFO_STORE] <= issue_lsu_i & ~issue_rd_valid_i & ~take_interrupt_i;

```

```

509 assign HPC_stall_by_MEM = ( ((ctrl_e2_q[PCINFO_LOAD] | ctrl_e2_q[PCINFO_STORE]) & ~mem_complete_i));

```

```

535 .stall_by_MEM (HPC_stall_by_MEM),

```

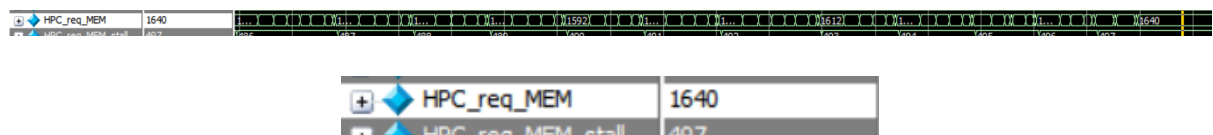
그 결과, 이는 149의 값을 가짐을 확인할 수 있었다.

다음으로 전체 memory instruction 수는 다음의 코드를 사용하여 다음과 같이 계산됨을 확인하였다.

```

always @(posedge rst_i or posedge clk) begin
    if (rst_i)
        HPC_req_MEM <='b0;
    else if(req_inst_load || req_inst_store)
        HPC_req_MEM <= HPC_req_MEM + 1'b1;
end

```



그 값은 위의 파형과 같이 1640으로 카운트 되었다.

다음으로 아래의 코드를 통해 memory-bound instruction (store, load instruction)에 따른 stall cycle 수를 계산하였다.

```

// wire stall_sig_rising_edge_MEM;
// assign stall_sig_rising_edge_MEM = stall_by_MEM;

always @(posedge rst_i or posedge clk) begin
    if (rst_i)
        HPC_req_MEM_stall_cycle <='b0;
    else if(stall_sig_rising_edge_MEM)
        HPC_req_MEM_stall_cycle <= HPC_req_MEM_stall_cycle + 1'b1;
end

```





위의 코드를 통해 앞서 `riscv_pipe_ctrl.v` 의 코드를 직접 살펴봄으로써, `stall_by_MEM` 이 의미하는 바를 정확히 파악하였으므로, 이는 valid한 코드임을 알 수 있다.

그 결과, 이는 497의 값을 가짐을 확인할 수 있었다.

정리하면, 총 149개의 memory-bound instruction (store, load instruction)이 총 497 cycle의 stall을 발생시켰다. 이에 의한 CPI 증가값을 계산했고, 그 결과는 다음과 같다.

- 총 CPI : 1.4840
- 해당 instruction에 의한 stall이 발생하지 않은 이상적인 경우의 CPI : $\frac{7681-497}{5176} = \frac{7184}{5176} = 1.3879$
- CPI 증가값 : $1.4840 - 1.3879 = 0.0961$

결과에 대한 파형은 다음과 같이 확인할 수 있다.

  HPC_req_MEM_stall_cycle	497	496		497				
  HPC_req_MEM_cause_stall	149	148		149				