Project_1_2017170992_성인규

- problem1_1.PNG (압축 폴더 첨부)

```
user@LAPTOP-JOMLA41T /cygdrive/c/Users/user/Desktop/Project1_uploaded(2)/Cygwin&Compiler/under_graduate
$ ls -l
합계 10
-rwx------+ 1 user 없음  473 3월  14 22:09 bin2hex.perl
-rwx------+ 1 user 없음  244 3월  14 22:09 desktop.ini
-rwx------+ 1 user 없음  129 3월  14 22:09 init.S
-rwx------+ 1 user 없음  376 3월  14 22:09 loop_test.c
-rwx------+ 1 user 없음  838 3월  14 22:10 Makefile
-rwx------+ 1 user 없음  150 3월  14 22:09 sections.lds
-rwx------+ 1 user 없음  170 3월  14 22:09 test_main.c
```

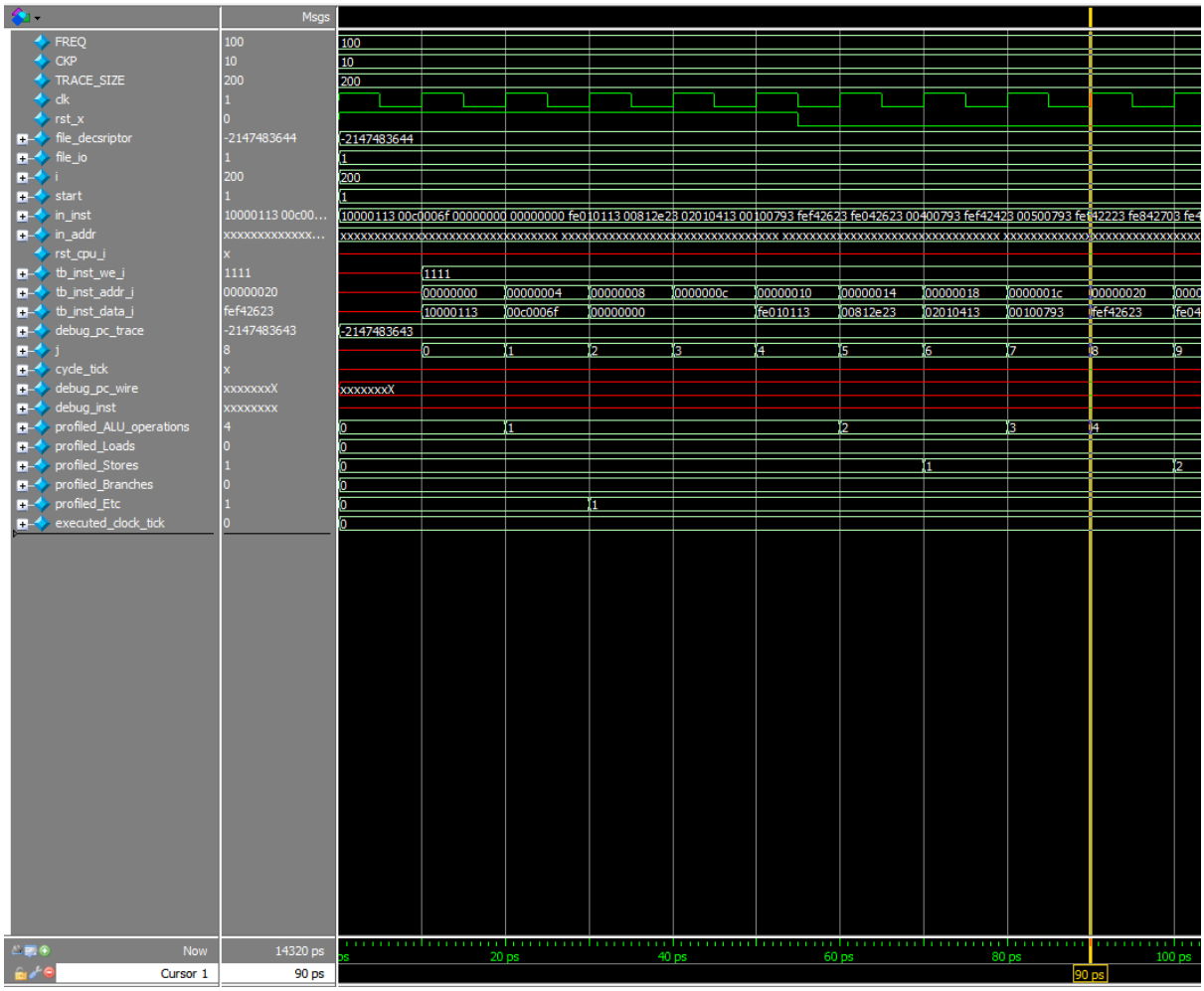
- problem1_2.PNG (압축 폴더 첨부)

```
user@LAPTOP-JOMLA41T /cygdrive/c/Users/user/Desktop/Project1_uploaded(2)/Cygwin&Compiler/under_graduate
$ make
/home/work/riscv32_cross_compiler/bin/riscv32-unknown-elf-gcc -c -g -march=rv32i test_main.c
/home/work/riscv32_cross_compiler/bin/riscv32-unknown-elf-as -g -march=rv32i init.S -o init.o
init.S: Assembler messages:
init.S: Warning: end of file not at end of a line; newline inserted
/home/work/riscv32_cross_compiler/bin/riscv32-unknown-elf-ld -N -X -T sections.lds init.o test_main.o -o out
/home/work/riscv32_cross_compiler/bin/riscv32-unknown-elf-objdump -D  out > out.dump
/home/work/riscv32_cross_compiler/bin/riscv32-unknown-elf-objcopy -O  binary out out.bin
./bin2hex.perl > out.hex

user@LAPTOP-JOMLA41T /cygdrive/c/Users/user/Desktop/Project1_uploaded(2)/Cygwin&Compiler/under_graduate
$ ls -l
합계 46
-rwx------+ 1 user 없음   473 3월  14 22:09 bin2hex.perl
-rwx------+ 1 user 없음   244 3월  14 22:09 desktop.ini
-rw-r--r--+ 1 user 없음  2852 3월  14 22:11 init.o
-rwx------+ 1 user 없음   129 3월  14 22:09 init.S
-rwx------+ 1 user 없음   376 3월  14 22:09 loop_test.c
-rwx------+ 1 user 없음   838 3월  14 22:10 Makefile
-rwxr-xr-x+ 1 user 없음  3148 3월  14 22:11 out
-rwxr-xr-x+ 1 user 없음  1280 3월  14 22:11 out.bin
-rw-r--r--+ 1 user 없음 14531 3월  14 22:11 out.dump
-rw-r--r--+ 1 user 없음  2880 3월  14 22:11 out.hex
-rwx------+ 1 user 없음   150 3월  14 22:09 sections.lds
-rwx------+ 1 user 없음   170 3월  14 22:09 test_main.c
-rw-r--r--+ 1 user 없음  2876 3월  14 22:11 test_main.o
```

Memory clock 0~8 cycle

- problem2_1.PNG (압축 폴더 첨부)



위와 같이 testbench module이 simulation 시작 후, write enable 신호가 1111로 인가되어, memory로 binary instruction을 copy하는 waveform을 확인할 수 있다. testbench의 각 신호를 조금 더 자세히 살펴보면, 다음과 같다.

- write enable
  - `tb_inst_we_i` 신호가 1111임을 알 수 있다. 이 신호가 1111의 값을 유지하는 동안, 전체 instruction data(32bit 전체 값)를 RAM에 copy한다. 이는 `tcm_mem_ram.v`의 코드를 통해 확인하겠다.
- address
  - 각 instruction data의 address에 해당하는 값이다. 즉, `ram[address]`에 instruction data가 copy된다. 이 또한 코드를 통해 확인하겠다.
  - address 값은 한 instruction 마다 4씩 증가함을 알 수 있다. 32-bit processor이므로 address 값이 4 byte(=32bit)씩 증가한다.
- data
  - `input_inst.dat`의 instruction data 값이다. clk마다 값을 읽어오며, `TRACE_SIZE`(200)까지의 data를 읽어온다.
  - 이 값이 `ram[address]`에 copy되는 값이다. 마찬가지로 코드를 통해 확인하겠다.

다음의 코드를 통해 앞서 설명한 내용을 확인할 수 있다.

```
//------------------------------------------------------------------
// tcm_mem_ram.v
// Dual Port RAM 64KB
// Mode: Read First
//------------------------------------------------------------------
// Synchronous write
// wr0_i; tb_inst_we_i, addr0_i; tb_inst_addr_i, data0_i; tb_inst data_i
always @ (posedge clk0_i)
begin
    if (wr0_i[0])
```

```
        ram[addr0_i][7:0] <= data0_i[7:0];
    if (wr0_i[1])
        ram[addr0_i][15:8] <= data0_i[15:8];
    if (wr0_i[2])
        ram[addr0_i][23:16] <= data0_i[23:16];
    if (wr0_i[3])
        ram[addr0_i][31:24] <= data0_i[31:24];

    ram_read0_q <= ram[addr0_i];
end
```

CPI 계산

$$CPI = \frac{\Sigma_{i=1}^{n}(CPI_i \cdot C_i)}{Instruction\ Count} = \frac{Clock\ cycles}{Instruction\ Count}$$

- 위의 식을 통해 CPI를 계산한다. 따라서, program이 실행되는 동안의 총 clock 수, 그리고 program이 실행되는 동안의 실행된 instruction의 개수를 count하여 나눠줌으로써 CPI를 계산할 수 있다.
- Testbench simulation을 통해 Clock cycles는 36, Instrcution Count는 21임을 확인할 수 있었다. 자세한 내용은 아래와 같다.
- $CPI = \frac{36}{21} = 1.71428571$
- 따라서, CPI 값은 1.71428571 임을 알 수 있다.

Instrcution Count

```
VSIM 4> run
# File Read Done!
# Instruction enqueue start
# Reset disable... Simulation Start !!!
# Instruction enqueue end
# Instruction profile result
# Profiled #of ALU operations: 9
# Profiled #of Load operations: 4
# Profiled #of Store operations: 6
# Profiled #of Branches operations: 0
# Profiled #of Etc. operations: 2
# Profiled #of All instrunctions: 21
# Core reset
# Executed clock tick 36
# 1
# Break in Module tb_tcm_riscv_top at C:/Users/user/Desktop/Project1_uploaded(2)/src_project1/tb/tb_tcm_riscv_top.v line 294
```

위와 같이 Transcript에서 log를 통해 `input_inst.dat` 파일의 **전체 instruction의 수는 21임**을 확인할 수 있다.

- Prepare phase에서 `in_inst`에서 값을 읽어 memory에 write할 때, instruction profiling을 수행하였다. 즉, 값을 읽어오는 동시에 operation의 type에 따라 count해주었다.
- `TRACE_SIZE` 만큼 instruction data를 다 읽은 후에, 각 operation 별로 count한 결과와 전체 count 결과를 transcript에 `display`하였다.
- 이에 대한 코드는 다음과 같다.

```
//------------------------------------------------------------------
// tb_tcm_riscv_top.v
//------------------------------------------------------------------

    if     ( (req_instruction[6:0] == `OP_R_TYPE)
            |(req_instruction[6:0] == `OP_I_TYPE_ARITH)
            |(req_instruction[6:0] == `OP_M_TYPE) )          profiled_ALU_operations = profiled_ALU_operations + 1;

    else if ( (req_instruction[6:0] == `OP_I_TYPE_LOAD) )    profiled_Loads = profiled_Loads + 1;
    else if ( (req_instruction[6:0] == `OP_S_TYPE) )         profiled_Stores = profiled_Stores + 1;
    else if ( (req_instruction[6:0] == `OP_B_TYPE) )         profiled_Branches = profiled_Branches + 1;
    else if ( (req_instruction[6:0] == `OP_I_TYPE_JALR)
             |(req_instruction[6:0] == `OP_J_TYPE)
             |(req_instruction[6:0] == `OP_U_TYPE_AUIPC)
             |(req_instruction[6:0] == `OP_U_TYPE_LUI) )     profiled_Etc = profiled_Etc + 1;

// ...


   $display ("Instruction enqueue end");

   $display ("Instruction profile result");
```

```
    $display ("Profiled #of ALU operations: %0d", profiled_ALU_operations);
    $display ("Profiled #of Load operations: %0d", profiled_Loads);
    $display ("Profiled #of Store operations: %0d", profiled_Stores);
    $display ("Profiled #of Branches operations: %0d", profiled_Branches);
    $display ("Profiled #of Etc. operations: %0d", profiled_Etc);
    $display ("Profiled #of All instrunctions: %0d", profiled_ALU_operations + profiled_Loads + profiled_Stores + profiled_Bran
```

input_inst.dat

```
10000113
00C0006F
00000000
00000000
FE010113
00812E23
02010413
00100793
FEF42623
FE042623
00400793
FEF42423
00500793
FEF42223
FE842703
FE442783
00F707B3
FEF42623
FEC42783
00078513
01C12403
02010113
00008067
```

- 위와 같이 직접 `input_inst.dat` 파일을 직접 살펴봐도 확인할 수 있다. `input_inst.dat` 는 프로그램 실행 파일(기계어)이고, 이를 그대로 input으로 넣어준다.
- 이 input에 따라서 instruction이 그대로 execute되기 때문에, 프로그램 실행 종료까지 **총 21개의 instruction이 execute되었음을 확인할 수 있다.**

Clock cycles

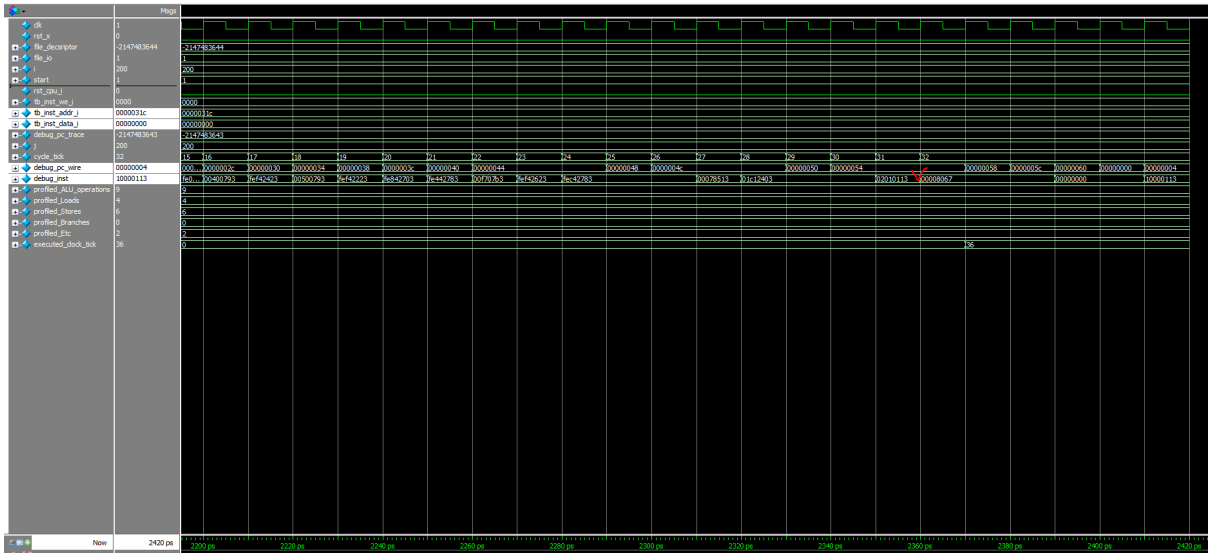
```
VSIM 4> run
# File Read Done!
# Instruction enqueue start
# Reset disable... Simulation Start !!!
# Instruction enqueue end
# Instruction profile result
# Profiled #of ALU operations: 9
# Profiled #of Load operations: 4
# Profiled #of Store operations: 6
# Profiled #of Branches operations: 0
# Profiled #of Etc. operations: 2
# Profiled #of All instrunctions: 21
# Core reset
# Executed clock tick 36
# 1
# Break in Module tb_tcm_riscv_top at C:/Users/user/Desktop/Project1_uploaded(2)/src_project1/tb/tb_tcm_riscv_top.v line 294
```

- Program이 실행되는 동안의 총 clock cycle 수도 위의 log에서 확인할 수 있다.
- 위의 사진을 보면 Prepare Phase 이후, Execution Phase가 시작되면서 Core reset이라는 log가 출력됨을 확인할 수 있다.
- 이때 `rst_cpu_i` 가 set된다. 이에 대한 코드는 아래의 코드와 같다.
- 즉, program execution의 시작 지점이므로 이 지점 부터 CPI 계산을 고려한 clock을 count한다.
- Core reset 이후 출력된 log에서 **Executed clock tick이 36임**을 확인할 수 있다.

```
//Execution Phase
wait_clocks(1);
$display ("Core reset");
rst_cpu_i = 1'b1;
```

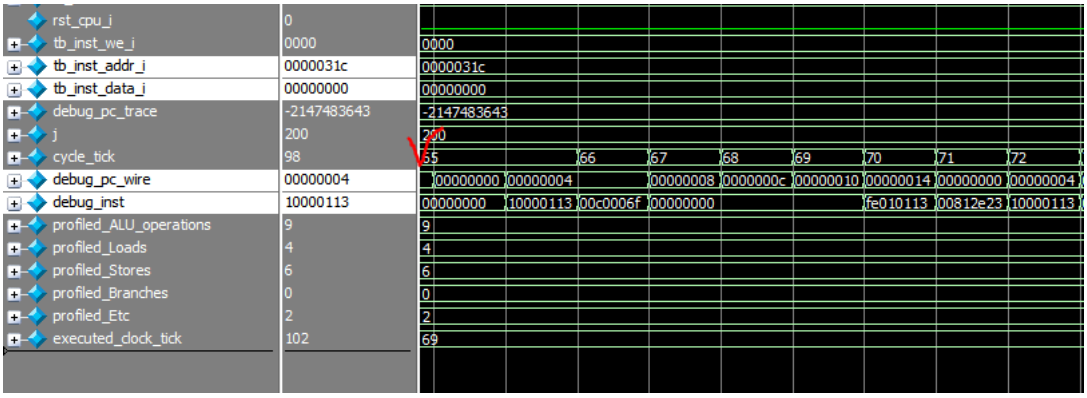
```
wait_clocks(1);
rst_cpu_i = 1'b0;
```

- 이는 testbench simulation을 통해 waveform으로도 확인할 수 있다.

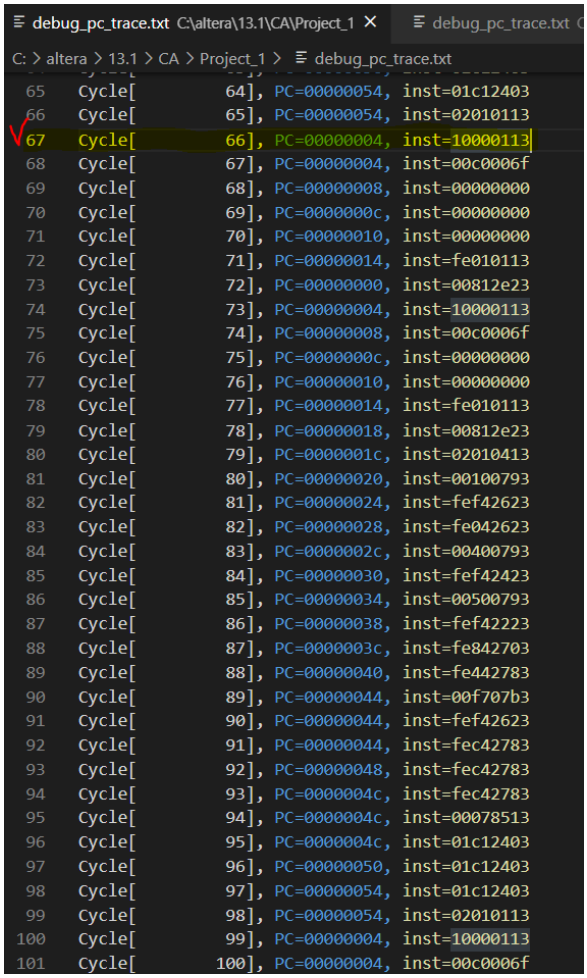


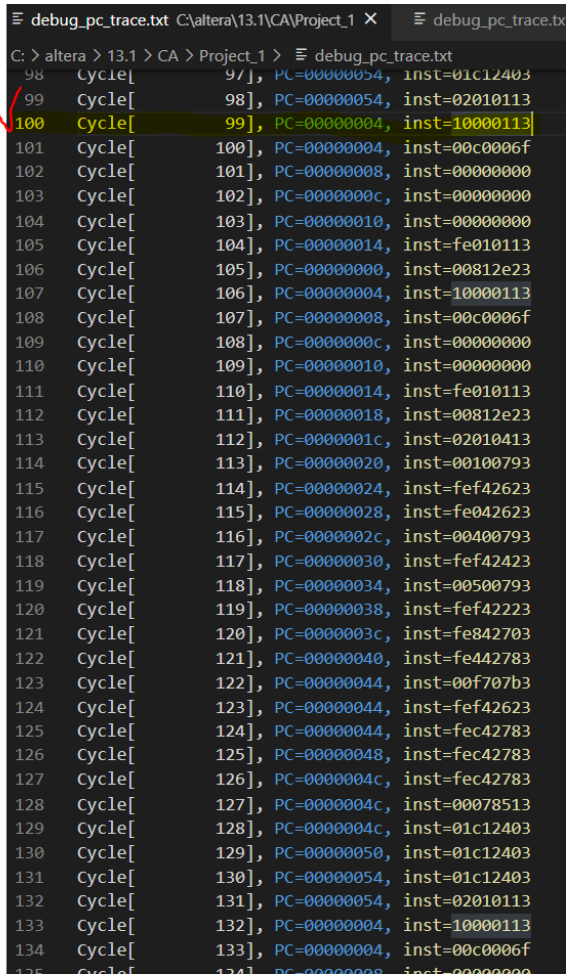
- 빨간색으로 marking한 부분에서, `clock_tick` 값이 32일때 `input_inst.dat`의 마지막 instruction인 0×00008067이 fetch되고, 4 cycle 이후 연산이 종료됨을 확인할 수 있다. (32 + 4 = 36)
- 따라서, 총 **Executed clock cycle은 36**임을 확인할 수 있다.

- 위의 설명을 마지막으로 CPI를 계산하는 과정은 끝났다.
- 허나 종료 이후에도 계속해서 시뮬레이션을 run하면, `debug_pc_wire` 의 값이 다음의 사진과 같이 계속해서 바뀌는 걸 확인할 수 있다. 따라서, 이에 대해 추가적으로 조사해보았다.



- 이를 자세히 살펴보면, 다음과 같이 일정한 주기와 순서를 가지고 반복적으로 값이 변함을 확인할 수 있다.





- 빨간색으로 marking한 부분부터 pc값을 비교해보면 확인할 수 있다.
- 또한, 이는 계속해서 simulation을 진행해도 **infinite하게 반복**됨을 확인할 수 있다.
- 이에 대한 조사를 진행하였으나, 직접적인 해답을 얻을 수 있는 레퍼런스는 찾을 수 없었다.
- 다만 아래의 레퍼런스들을 참고했을 때 이는 execution이 종료된 이후, 무의미하게 반복되는 값으로 추측된다.
  - Does the program counter always have to change (upon a clock tick)?
  - A Computer Clock That Won't Stay Set

- `debug_pc_trace.txt` (압축 폴더 첨부)
  - Clock 0 - Clock 32