ELEC3608 - Computer Architecture
Assignment Report - Pipelined Processor

Kevin Nguyen - 311254039 and Khanh Nguyen - 311253865

Part 1: Implementation of a hazard-free function nfib(unsigned int n)

Design and Implementation

This part of the assignment required the nfib function to be written and tested on the pipelined PS_SPIM processor. The nfib function is limited to the available instructions in the given processor.

The nfib(unsigned int n) function returns the largest i for which fib(i) <= n.
(e.g. nfib(0)=0, nfib(1)=2, nfib(100,000)=25)

Our initial design of nfib(n) was to calculate each fibonacci number upwards and count how many time we called fib(n). We had previously implemented a recursive fibonacci function in MIPS, however it exponentially slowed down as the input value increased. There was also mentions that there is a lack of stack memory in the given processor. This lead us to a sequential design as seen in **Figure 1**, which was more simplistic and efficient.

```
function nfib(int n){
    int a = 0
    int b = 1
    int counter = 1
    int temp = 0

    while (b < n){
        temp = b
        b = a + b
        a = temp
        counter++
    }
    return counter
}
```

Figure 1: Pseudocode for nfib(n)

The pseudocode in **Figure 1** is our design of nfib(n). We start by initialising variables *a* which is the placeholder for the previous fibonacci number*, b* which is the placeholder for the current fibonacci number*, counter* and *temp* which is a temporary variable used for swapping variables.

The function then loops until the current fibonacci number *b* is greater or equal to the input value *n*, each time calculating the next fibonacci number, rearranging the variables for the next loop and incrementing the counter. You can see that the fibonacci series is visible in the *b* row of the table in **Figure 2**.

a	0	1	1	2	3	5	8	13	21	34	55	89	144	233	...
b	*1*	*1*	*2*	*3*	*5*	*8*	*13*	*21*	*34*	*55*	*89*	*144*	*233*	*377*	...
temp	0	1	1	2	3	5	8	13	21	34	55	89	144	233	...
counter	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Figure 2: Values of nfib(n) at the start and after each loop

```
.data
msg1:.asciiz "Give a number: "
.text
.globl main
main:      li $s0, 0 #a = 0
     li $s1, 1   #b = 1
     li $s2, 0  #counter = 0
     li $a0, 123  #input
     addi $a0, $a0, 1

fib: slt $t1, $s1, $a0
     beq $t1, $0, exit
     add $t0, $s1, $0   #t = b
     add $s1, $s0, $s1  #b = a+b
     add $s0, $t0, $0   #a = t
     addi $s2, $s2, 1
     j fib

exit:      li $v0, 10
     syscall
```

Figure 3: MIPS assembly of nfib(n) as written on Qtspim.exe

This was not compatible with the given PS_SPIM processor because:

- No i format instructions, initialised memory then load into registers
- To increment counter, because there is no i format, had to lw one of the registers with the value of 1 and used that to increment.
- To j instruction, used beq $0, $0, xx as unconditional jump
- Could not use slt so we used ‘sub’ and ‘and’ instead
    - Given value A and B, we did ‘B - A’ then AND it with 0x8000000. If B was smaller than A then B-A would return a negative result. This result AND 0x8000000 will return 0x80000000. However if it was a positive result, then it will return 0x0. So the result is always 0x0 when B is larger or equal to A and from there we can branch to the exit.

```
1       X"8C110004",        -- lw $s1, 4($0)
2       X"8C040008",        -- lw $a0, 8($0)
3       X"00008020",        -- add $s0, $zero, $zero
4       X"00009020",        -- add $s2, $0, $0
5       X"02242020",        -- add $a0, $a0, $s1
6       X"8C130004",        -- lw $s3, 4($0)
7       X"8C140010",        -- lw $s4, 16($0)
8       X"02244822",        -- sub $t1, $s1, $a0
9       X"00000000",
10      X"00000000",
11      X"01344824",        -- and $t1, $t1, $s4
12      X"00000000",
13      X"00000000",
14      X"11200008",        -- beq $t1, $0, 24
15      X"00000000",
16      X"00000000",
17      X"02204020",        -- add $t0, $s1, $0
18      X"02118820",        -- add $s1, $s0, $s1
19      X"00000000",
20      X"01008020",        -- add $s0, $t0, $0
21      X"02729020",        -- add $s2, $s3, $s2
22      X"1000FFF1",        -- beq $0, $0, -48 (PC + 4 - 60 = -56)
23      X"00000000",
24      X"00000000",
25      X"00000000"
```

Figure 4: Translated MIPS assembly of nfib(n)

Lines 1 to 8 are arranged to avoid data hazards, e.g. Line 2 (lw $a0, 8($0)) rd is equal to line 5(add $a0, $a0, $s1) rs (type 1a hazard) but line 2 is already in the write back stage when line 5 is in the instruction decode stage so no data hazard occurs. The same applies for line 1 and line 5(lw $s1, 4($0) , add $a0, $a0, $s1)

Lines 8, 11 and 14 is our conditional branch section and cannot be re arranged. So it data hazards are avoided by using ‘nop’. It waits for the results to be written back before the instruction decode of the next operation occurs. Each beq instruction is followed by nops to prevent control hazards, as we cannot fetch the next instruction until we know the result of the beq operation.

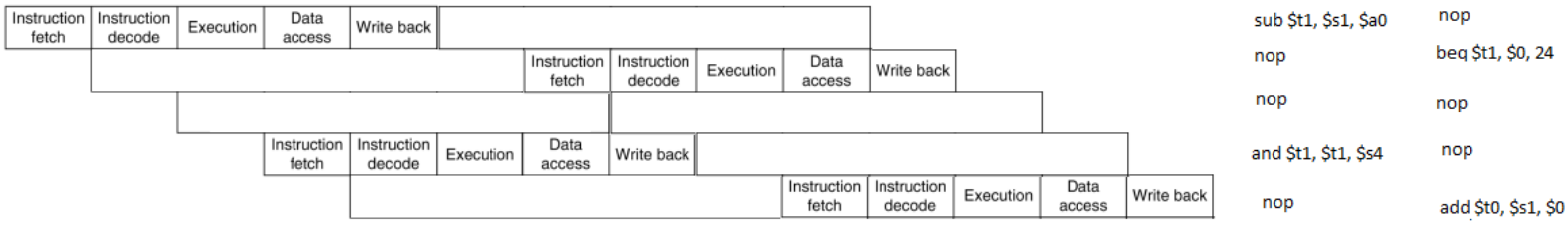


Figure 5: Multi-Cycle pipeline diagram, lines 8 to 17

Changes to PS_SPIM (vhdl)

```
62  BEGIN
63  -- access instruction pointed to by current PC
64  -- and increment PC by 4. This is combinational
65
66  Instruction <=  iram(CONV_INTEGER(PC(7 downto 2)));
67  PC_out<= (PC + 4);
68
```

Figure 6: Changes to ps_fetch.vhd included the PC being changed to accommodate for the larger nfib program(figure 4). This avoids any wrap around issues with the array when address is out of bounds. The size if the INST_MEM array was also changed to fit the instructions.

```
46
47  ALUop(1 downto 0) <= "01" WHEN Beq = '1' ELSE rformat & '0';
48
```

Figure 7: ps_control.vhd was changed so that the correct ALUop would be generated when we used beq.

The change in figure 7 was required so that the BEQ instruction would work.

```
26  architecture behavioral of memory is
27
28  TYPE DATA_RAM IS ARRAY (0 to 31) OF STD_LOGIC_VECTOR (31 DOWNTO 0);
29     SIGNAL dram: DATA_RAM := (
30        X"00000000",              --A value
31        X"00000001",              --B value
32        X"075bcd15",              -- 123456789 input
33        X"0000007B",              -- 123 input
34        X"80000000",              -- to get most sig. bit
```

Figure 8: ps_memory.vhd was initialised to the required values for the program. Only entries 0-4 were changed in the DATA_RAM array.

Testing

- **Testing Branch on Equal (BEQ) - generate correct aluOp and ALU_clt**

Although this was not part of the assignment, the given BEQ instruction was not functional, so we had to fix and test it. To test the BEQ instruction we used the following program:

```
X"11200007",  -- beq $t1, $0, 32
X"00000000",  -- nop
X"00000000"   -- nop
```

Figure 9: Program to test BEQ

Expected Results:

1. the Branch_PC to be of value 0x20 (pc counter jumped up by 32)
2. the function opcode to be 0b000111
3. the ALUOp to be 0b01 if our changes in fig 7 are correct.

From there the correct ALU_ctl can be generated from function_opcode and ALUOp to tell ALU to subtract, expected value of ALU_ctl is 0b01.

Results:

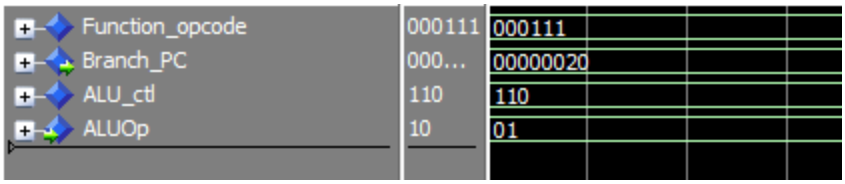


Figure 10: Waveform of BEQ test program

As seen in figure 10 above, the BEQ test program passes and the instruction works as expected.

- **Testing nfib(n) - see if final register values are correct**

To test our nfib(n) function, we used the following test program, we ran it and had a look at the final values of the registers.

```
X"8C110004",        -- lw $s1, 4($0)
X"8C040008",        -- lw $a0, 8($0)
X"00008020",        -- add $s0, $zero, $zero
X"00009020",        -- add $s2, $0, $0
X"02242020",        -- add $a0, $a0, $s1
X"8C130004",        -- lw $s3, 4($0)
X"8C140010",        -- lw $s4, 16($0)
X"02244822",        -- sub $t1, $s1, $a0
X"00000000",
X"00000000",
X"01344824",        -- and $t1, $t1, $s4
X"00000000",
X"00000000",
X"11200008",        -- beq $t1, $0, 32 (PC + 4 + 28 = 32)
X"00000000",
X"00000000",
X"02204020",        -- add $t0, $s1, $0
X"02118820",        -- add $s1, $s0, $s1
X"00000000",
X"01008020",        -- add $s0, $t0, $0
X"02729020",        -- add $s2, $s3, $s2
X"1000FFF1",        -- beq $0, $0, -54 (PC + 4 - 60 = -54)
X"00000000",
X"00000000",
X"00000000"
```

Figure 11: Test program for nfib(n)

Expected Results:

- register $s2 (=$18) to contain the value 40

Results:

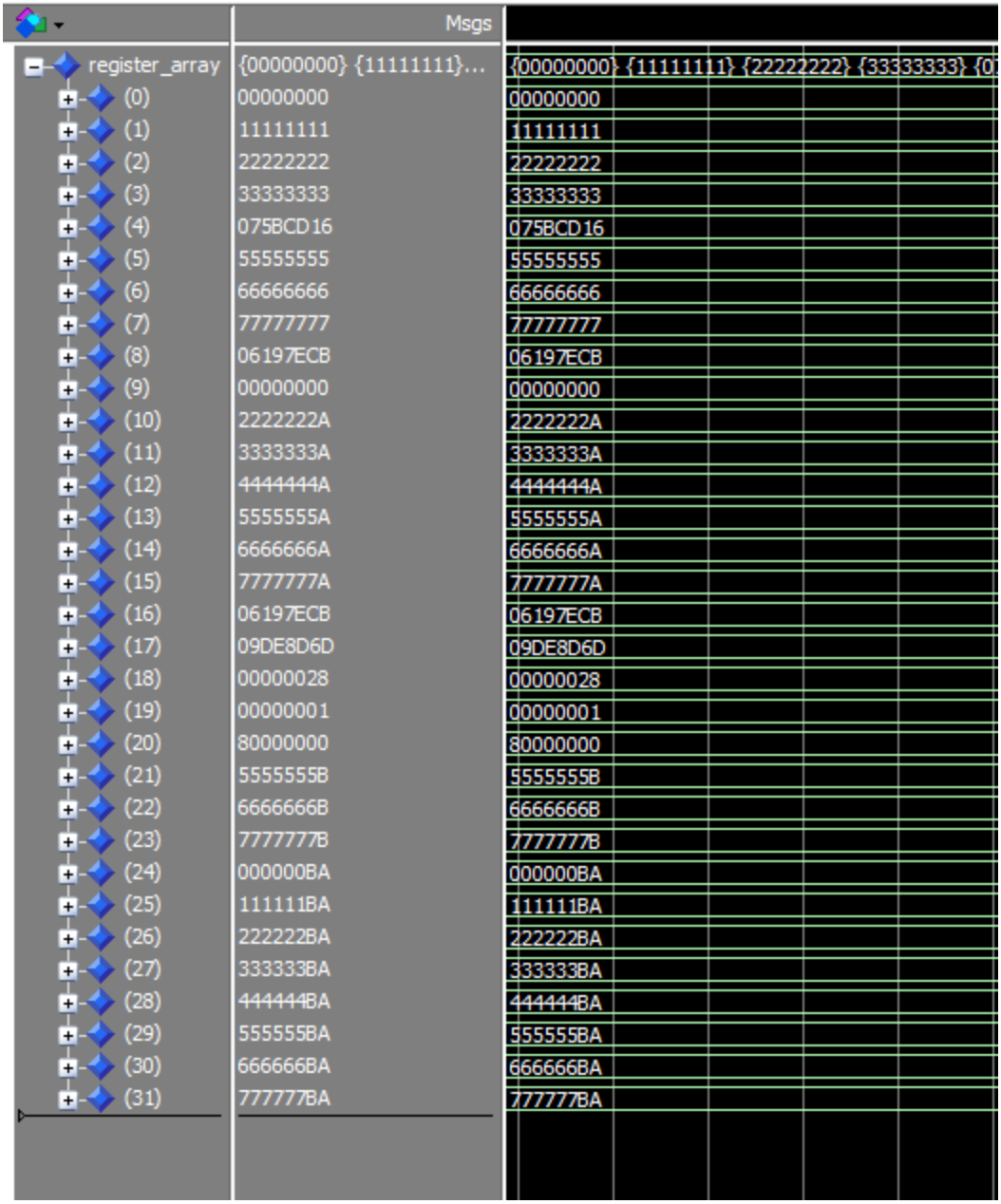


Figure 12: Waveform results of simulation of nfib(n) test program

As seen above in figure 12:

- register $4 containing our input 0x75BCD16 (123456789)
- register $16 and $17 are our a and b variables, respectively. B is our fib sequence
- register $18 is our counter and we can see that it contains 0x28 (40), which is what we expected.

Part 2: Add slt and slti instructions to PS_SPIM, and modify nfib to use slt/slti.

Design and Implementation

This part of the assignment required the *set on less than* (SLT) and the *set on less than immediate* (STLI) instructions to be added to the PS_SPIM processor.

```
89          WHEN "111"  =>  if(Ainput < Binput) then
90                              ALU_internal <= X"00000001";
91                          else
92                              ALU_internal <= X"00000000";
93                          end if;
```

Figure 13: In ps_fetch.vhd, the switch case is modified so that the correct value of either 1 or 0, depending on the result, is the output for the ALU when the slt operation is performed.

```
27      rformat     <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
28      Lw          <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
29      Sw          <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
30      Beq         <=  '1'  WHEN  Opcode = "000100"  ELSE '0';
31      iformat     <=  '1'  WHEN  Opcode = "001010"  ELSE '0';
32  --
33  -- implement each output signal as the column of the truth
34  -- table  which defines the control
35  --
36
37  RegDst <= rformat;
38  ALUSrc <= (lw or sw or iformat) ;
39
40  MemToReg <= lw ;
41  RegWrite <= (rformat or lw or iformat);
42  MemRead <= lw ;
43  MemWrite <= sw;
44  Branch <= beq;
45
```

Figure 14: Changes made to ps_control.vhd

We added an iformat flag to warn us when it is an i format instruction, edited RegWrite flag to work for i format instructions. This enables i format instructions to right back to register.

```
46
47  ALUop(1 downto 0) <= "01" WHEN Beq = '1' ELSE
48                       "11" WHEN iformat = '1' ELSE
49                       "10" WHEN rformat ='1' ELSE
50                       "00";
```

Figure 15: ps_control.vhd, new ALUop logic to allow i format instructions.

For the changes we made to ps_control as seen in figure 15 will set ALUop = 0b11 for i format instructions. This is used for the SLTI instruction.

```
61    ALU_ctl( 0 ) <= '1' WHEN ALUOp = "11" ELSE ( Function_opcode( 0 ) OR Function_opcode( 3 ) ) AND ALUOp(1) ;
62    ALU_ctl( 1 ) <= '1' WHEN ALUOp = "11" ELSE ( NOT Function_opcode( 2 ) ) OR (NOT ALUOp( 1 ) );
63    ALU_ctl( 2 ) <= '1' WHEN ALUOp = "11" ELSE ( Function_opcode( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );
64
```

Figure 16: ps_control.vhd, new ALU_ctl logic to generate correct ALU_ctl signal when using slti

We then removed the AND logic in place for the SLT instruction in the nfib(n) function.

```
X"8C110004",     -- lw $s1, 4($0)
X"8C040008",     -- lw $a0, 8($0)
X"00008020",     -- add $s0, $zero, $zero
X"00009020",     -- add $s2, $0, $0
X"02242020",     -- add $a0, $a0, $s1
X"8C130004",     -- lw $s3, 4($0)
X"00000000",     -- nop
X"0224482A",     -- slt $t1, $s1, $a0
X"00000000",     -- nop
X"00000000",     -- nop
X"11200008",     -- beq $t1, $0, 32
X"00000000",     -- nop
X"00000000",     -- nop
X"02204020",     -- add $t0, $s1, $0
X"02118820",     -- add $s1, $s0, $s1
X"00000000",     -- nop
X"01008020",     -- add $s0, $t0, $0
X"02729020",     -- add $s2, $s3, $s2
X"1000FFF4",     -- beq $0, $0, -48 (PC + 4 - 48 = -44)
X"00000000",     -- nop
X"00000000",     -- nop
X"00000000"      -- nop
```

Figure 17: Translated MIPS assembly of nfib(n) using slt

Testing

- **Testing slt**

To test set on less than (slt), we used the following test program:

```
X"0001102A",          -- slt $2, $0, $1
X"00000000",
X"00000000",
X"00000000",
X"00000000",
X"00000000"
```

Figure 18: Test program for set on less than

The registers are already initialised, register $1 contains 0x11111111 and $0 contains 0x0 so $1 is larger.

Expected Results:

- regWrite = 1
- alu_result = 1
- ALUOp = 10
- ALUCtl = 111
- reg $2 to b 0x1

Result:

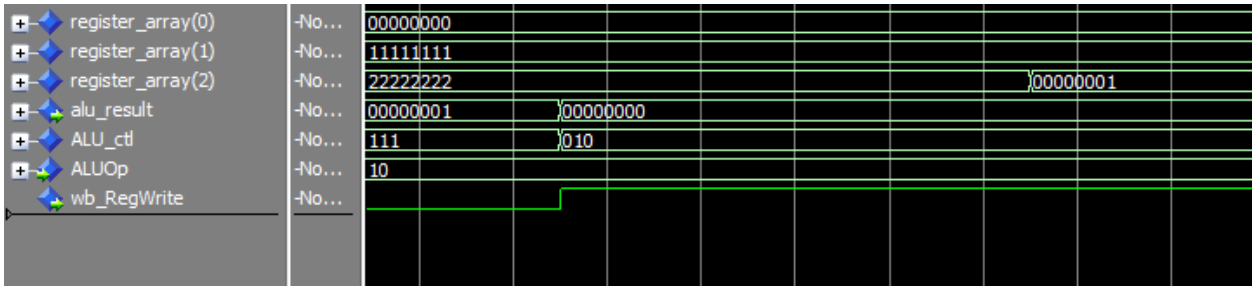


Figure 19: Waveform results from simulation of slt test program

As expected, we can see alu_result = 0b1 , ALU_clt = 0b111, ALUOp = 10.
Then we can see wb_RegWrite going high followed by the alu_result going into $2.

- **Testing slti**

To test set on less than immediate (slti), we used the following test program:

```
X"2802007B",          -- slti $2, $0, 123
X"00000000",
X"00000000",
X"00000000",
X"00000000",
X"00000000"
```

Figure 20: Test program for set on less than immediate

since $0 is 0, we expect results to be
regWrite = 1, alu_result = 1, ALUOp = 11 (since it is iformat) , ALUCtl = 111, reg $2 to b 0x1.

Results:

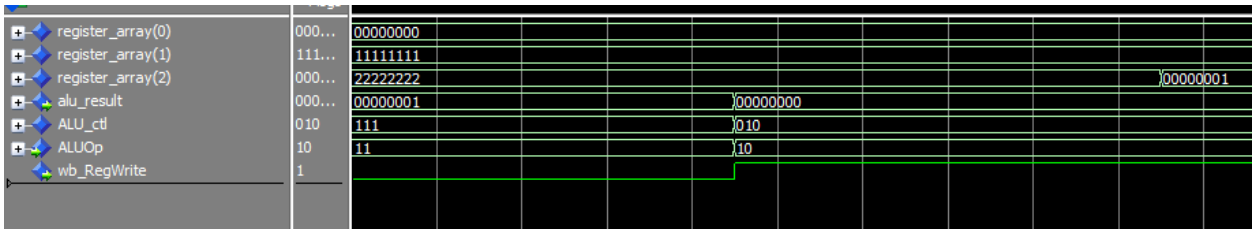


Figure 21: Waveform result from simulation of test program

as expected, ALUop = 11, ALU_ctl = 111, ALU_result = 0x1. Then we can see wb_RegWrite go high and then the value 0x1 being written into reg $2

Testing the nfib(n) program with slt:
The test program for this part that was used for testing can be seen in figure 11.

Expected Results:

- the final register value of $18 to be 0x28 (40).

Results:

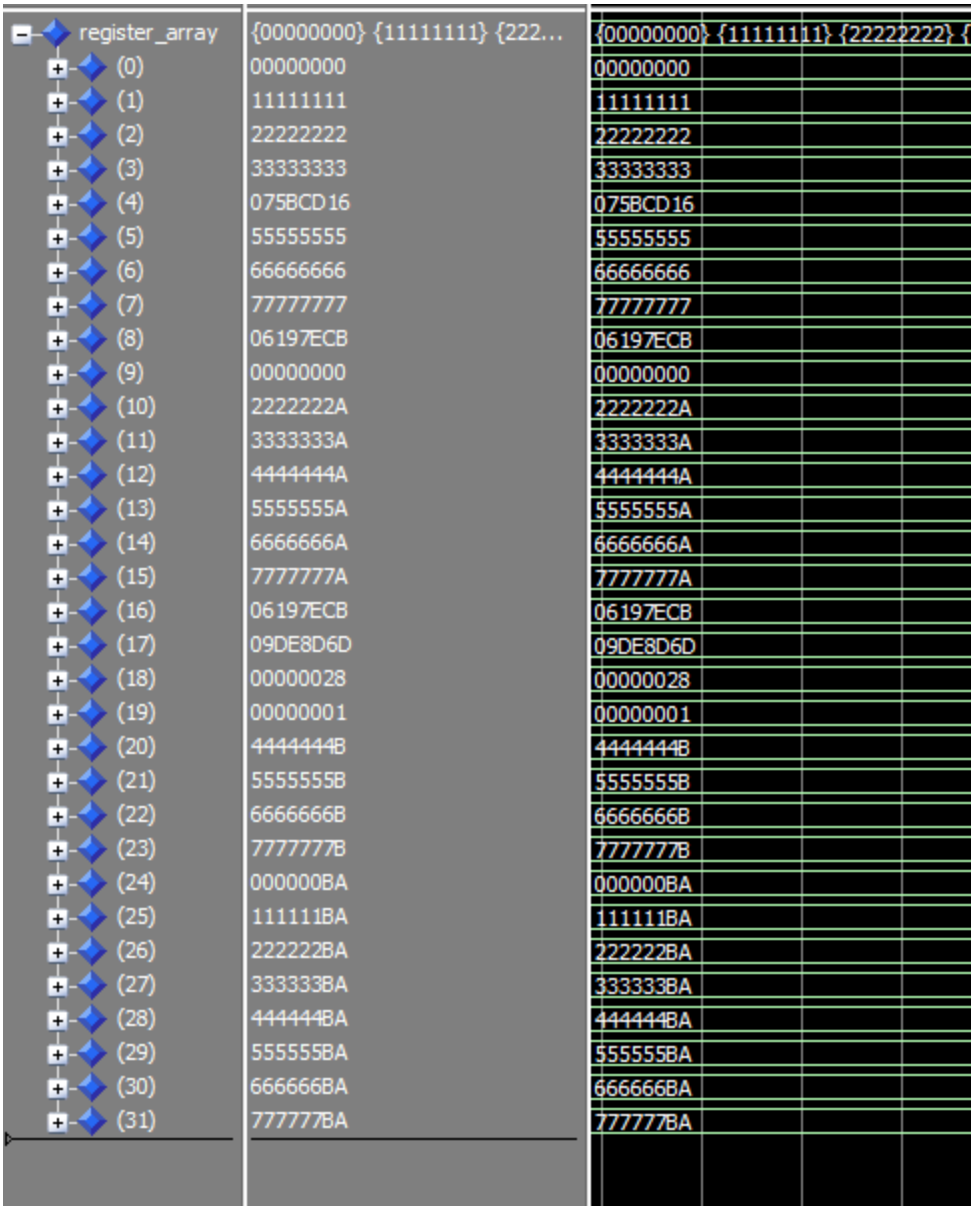


Figure 22: Waveform result from simulation of test program nfib(n) with slt

Our results show reg $18 containing 28, so it it working as planned. The results are the same as part one. Only difference being that it is faster.

Part 3: Forwarding

Design and Implementation

For this part of the assignment, we were required to implement forwarding for all all R-type instructions implemented in the PS_SPIM processor along with the SLT/SLTI instructions so that they can proceed as early as possible.

We created a forward unit that is able to detect when a forward is necessary, where to forward to and what data to forward.

```
19  architecture behavioral of forward is
20      signal mem_fwd_rs, mem_fwd_rt, ex_fwd_rs, ex_fwd_rt: std_logic;
21  begin
22      ex_fwd_rs <=    '0' when (ex_mem_rd = "00000") else
23                      ex_mem_regWrite when (ex_mem_rd = id_ex_rs) else
24                      '0';
25
26      ex_fwd_rt <=    '0' when (ex_mem_rd = "00000") else
27                      ex_mem_regWrite when (ex_mem_rd = id_ex_rt) else
28                      '0';
29
30      mem_fwd_rs <=   '0' when (mem_wb_rd = "00000") else
31                      '0' when (ex_fwd_rs = '1') else
32                      mem_wb_regWrite when (mem_wb_rd = id_ex_rs) else
33                      '0';
34
35      mem_fwd_rt <=   '0' when (mem_wb_rd = "00000") else
36                      '0' when (ex_fwd_rt = '1') else
37                      mem_wb_regWrite when (mem_wb_rd = id_ex_rt) else
38                      '0';
39
40      forwardRs <= ex_fwd_rs & mem_fwd_rs;
41      forwardRt <= ex_fwd_rt & mem_fwd_rt;
42
43
44  end behavioral;
```

Figure 23: Behavior of the forward entity

Figure 23 shows the behavior of our forward unit. It uses the 4 hazard conditions:
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

To set the flags for either a forward from EX/MEM pipeline reg or a forward from MEM/WB pipeline reg.

In order to compare the Rs register in the ID/EX stage with the Rd register, we had to create some modifications so the Rs address would propagate through.

```
25    -- outputs
26    --
27          register_rs, register_rt :out std_logic_vector(31 downto 0);
28          Sign_extend :out std_logic_vector(31 downto 0);
29          wreg_rd, wreg_rt, wreg_rs : out std_logic_vector (4 downto 0));
30    end decode;
31
```

```
103         -- move possible write destinations to execute stage
104            wreg_rd <= instruction(15 downto 11);
105           wreg_rt <= instruction(20 downto 16);
106           wreg_rs <= instruction(25 downto 21);
```

Figure 24: Changes made in ps_decode.vhd, added output wreg_rs

```
4   entity pipe_reg2 is
5   port (id_MemToReg, id_RegWrite, id_MemWrite, id_MemRead: in std_logic;
6         id_ALUSrc, id_RegDst, clk, reset, id_branch : in std_logic;
7         id_ALUOp : in std_logic_vector(1 downto 0);
8         id_PC4: in std_logic_vector(31 downto 0);
9         id_register_rs, id_register_rt, id_sign_extend: in std_logic_vector(31 downto 0);
10        id_wreg_rd, id_wreg_rt, id_wreg_rs : in std_logic_vector(4 downto 0);
11
12        ex_MemToReg, ex_RegWrite, ex_MemWrite, ex_MemRead, ex_branch: out std_logic;
13        ex_ALUSrc, ex_RegDst : out std_logic;
14        ex_ALUOp : out std_logic_vector(1 downto 0);
15        ex_PC4: out std_logic_vector(31 downto 0);
16        ex_register_rs, ex_register_rt, ex_sign_extend: out std_logic_vector(31 downto 0);
17        ex_wreg_rd, ex_wreg_rt, ex_wreg_rs : out std_logic_vector(4 downto 0));
18  end pipe_reg2;
19
20
```

Figure 25: In pipe_reg2.vhd, we added in id_wreg_rs and ex_wreg_rs, lines 10 and 17 respectively. in_wreg_rs <= ex_wreg_rs

Now ps_execute.vhd needs to be changed to make forwarding work. It needs to when to use the register values or the forwarded data for its inputs for the ALU. On top of that, it needs to know where the forwarded data is coming from, i.e.EX/MEM stage or MEM/WB stage.

```
-- compute the two ALU inputs
Ainput <=    fwd_data_wb when forwardRs(0) = '1' ELSE
             fwd_data_mem when forwardRs(1) = '1' ELSE
             register_rs;

-- ALU input mux
Binput <=    fwd_data_wb when forwardRt(0) = '1' ELSE
             fwd_data_mem when forwardRt(1) = '1' ELSE
             register_rt WHEN ( ALUSrc = '0' ) else
             Sign_extend(31 downto 0) when ALUSrc = '1' else
             X"BBBBBBBB";
```

Figure 26: the mux for the inputs of the ALU. Given the forward flags set by the unit, it will choose its inputs accordingly.

```
11  entity execute is
12  port(
13  --
14   -- inputs
15   --
16        PC4 : in std_logic_vector(31 downto 0);
17        forwardRs, forwardRt: in std_logic_vector (1 downto 0);
18        register_rs, register_rt, fwd_data_wb, fwd_data_mem :in std_logic_vector (31 downto 0);
19        Sign_extend :in std_logic_vector(31 downto 0);
20        ALUOp: in std_logic_vector(1 downto 0);
21        ALUSrc, RegDst : in std_logic;
22        wreg_rd, wreg_rt : in std_logic_vector(4 downto 0);
23
24   -- outputs
```

Figure 27: the added inputs , fwd_data_wb and fwd_data_mem, for ps_execute.vhd (line 18).

With all the entities modified and ready, we now must connect the forwarding unit in the top level entity.

Inside the top level entity, spim_pipe we added the forward unit in. Then did the necessary internal signals and port mapping.

```
36        -- forwarding unit
37  component forward
38  port(ex_mem_regWrite:       in std_logic;
39       mem_wb_regWrite:       in std_logic;
40       mem_wb_rd:             in std_logic_vector(4 downto 0);
41       ex_mem_rd:             in std_logic_vector(4 downto 0);
42       id_ex_rs:              in std_logic_vector(4 downto 0);
43       id_ex_rt:              in std_logic_vector(4 downto 0);
44       forwardRs:             out std_logic_vector(1 downto 0);
45       forwardRt:             out std_logic_vector(1 downto 0));
46  end component;
```

Figure 28: Forwarding component

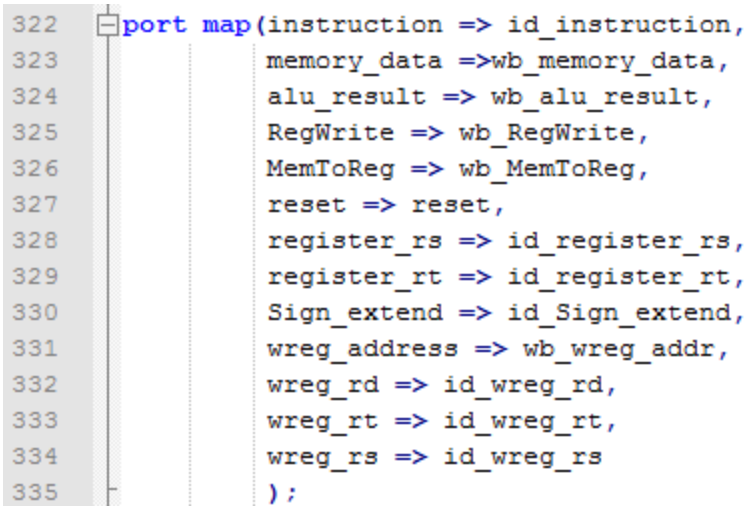
```
230  --
231  -- FW
232  --
233  signal forwardRt, forwardRs : std_logic_vector(1 downto 0);
273  -- instantiate forwarding
274  forward_unit : forward
275  port map(
276      ex_mem_regWrite => mem_RegWrite,
277      mem_wb_regWrite => wb_RegWrite,
278      mem_wb_rd => wb_wreg_addr,
279      ex_mem_rd => mem_wreg_addr,
280      id_ex_rs => ex_wreg_rs,
281      id_ex_rt => ex_wreg_rt,
282      forwardRs => forwardRs,
283      forwardRt => forwardRt);
```

Figure 29: Portmap for the forwarding component

```
372  EX: execute  -- instantiate the component EX?
373
374  port map(PC4 => ex_PC4,
375           forwardRs => forwardRs,
376           forwardRt => forwardRt,
377           fwd_data_wb => wb_alu_result,
378           fwd_data_mem => mem_alu_result,
379           register_rs => ex_register_rs,
380           register_rt => ex_register_rt,
381           sign_extend => ex_sign_extend,
382           RegDst => ex_RegDst,
383           ALUOp => ex_ALUOp,
384           ALUSrc => ex_ALUSrc,
385           alu_result => ex_alu_result,
386           wreg_rd => ex_wreg_rd,
387           wreg_rt => ex_wreg_rt,
388           wreg_address => ex_wreg_addr,
389           branch_pc => ex_branch_PC,
390           zero => ex_zero);
391
```

Figure 30: forwarding flags from the forward unit connected to EX component (line 375, 376).

The fwd_data_wb and fwd_data_mem registers take the alu_results from the MEM/WB stage and EX/MEM stage respectively.



```
322   port map(instruction => id_instruction,
323            memory_data =>wb_memory_data,
324            alu_result => wb_alu_result,
325            RegWrite => wb_RegWrite,
326            MemToReg => wb_MemToReg,
327            reset => reset,
328            register_rs => id_register_rs,
329            register_rt => id_register_rt,
330            Sign_extend => id_Sign_extend,
331            wreg_address => wb_wreg_addr,
332            wreg_rd => id_wreg_rd,
333            wreg_rt => id_wreg_rt,
334            wreg_rs => id_wreg_rs
335            );
```

Figure 31: port map for decode component. wreg_rs passes the address to the internal signal id_wreg_rs.

```
337   id_ex: pipe_reg2 -- instantiate the pipeline register ID/EX
338   port map(clk => clock,
339   reset => reset,
340        id_branch => id_branch,
341        id_MemToReg => id_MemToReg,
342        id_RegWrite => id_RegWrite,
343        id_MemWrite => id_MemWrite,
344        id_MemRead => id_MemRead,
345        id_ALUSrc => id_ALUSrc,
346        id_RegDst => id_RegDst,
347        id_ALUOp  => id_ALUOp,
348        id_PC4 => id_PC4,
349        id_register_rs  => id_register_rs,
350        id_register_rt  => id_register_rt,
351        id_sign_extend  => id_sign_extend,
352        id_wreg_rd  => id_wreg_rd,
353        id_wreg_rt  => id_wreg_rt,
354        id_wreg_rs => id_wreg_rs,
355
356        ex_branch => ex_branch,
357        ex_MemToReg => ex_MemToReg,
358        ex_RegWrite => ex_RegWrite,
359        ex_MemWrite => ex_MemWrite,
360        ex_MemRead => ex_MemRead,
361        ex_ALUSrc => ex_ALUSrc,
362        ex_RegDst => ex_RegDst,
363        ex_ALUOp => ex_ALUOp,
364        ex_PC4  => ex_PC4,
365        ex_register_rs  => ex_register_rs,
366        ex_register_rt  => ex_register_rt,
367        ex_sign_extend  => ex_sign_extend,
368        ex_wreg_rd  => ex_wreg_rd,
369        ex_wreg_rt  => ex_wreg_rt,
370        ex_wreg_rs  => ex_wreg_rs);
371
372   EX: execute  -- instantiate the component EX?
```

Figure 32: port map for ID/EX pipeline registers. Added in lines 354 and 370.

Testing

To test whether our forwarding unit works or not, we took the code from part 2 (figure 11) and removed all the nops. If the forwarding unit is performing as intended, then the results should be the same as part 2 but achieved in fewer clock cycles. We cannot remove all the nops however because we only forwarded R type instructions, so nops are still needed after beq instructions.

We used the following test program to test:

```
X"8C110004",  -- lw $s1, 4($0)
X"8C040008",  -- lw $a0, 8($0)
X"00008020",  -- add $s0, $zero, $zero
X"00009020",  -- add $s2, $0, $0
X"02242020",  -- add $a0, $a0, $s1
X"8C130004",  -- lw $s3, 4($0)
X"0224482A",  -- slt $t1, $s1, $a0
X"11200007",  -- beq $t1, $0, 28
X"00000000",  -- nop
X"00000000",  -- nop
X"02204020",  -- add $t0, $s1, $0
X"02118820",  -- add $s1, $s0, $s1
X"01008020",  -- add $s0, $t0, $0
X"02729020",  -- add $s2, $s3, $s2
X"1000FFF7",  -- beq $0, $0, -28 (PC + 4 - 32 = -28)
X"00000000",  -- nop
X"00000000",  -- nop
X"00000000"   -- nop
```

Figure 33: Test program for nfib(n) with forwarding

We expect the results to be exactly the same as part 2. But we also expect to see the amount of clock cycles needed to achieve the result to be less that parts 1 and 2. That is register $18 to contain 0x28 and cycles to compute > 617.

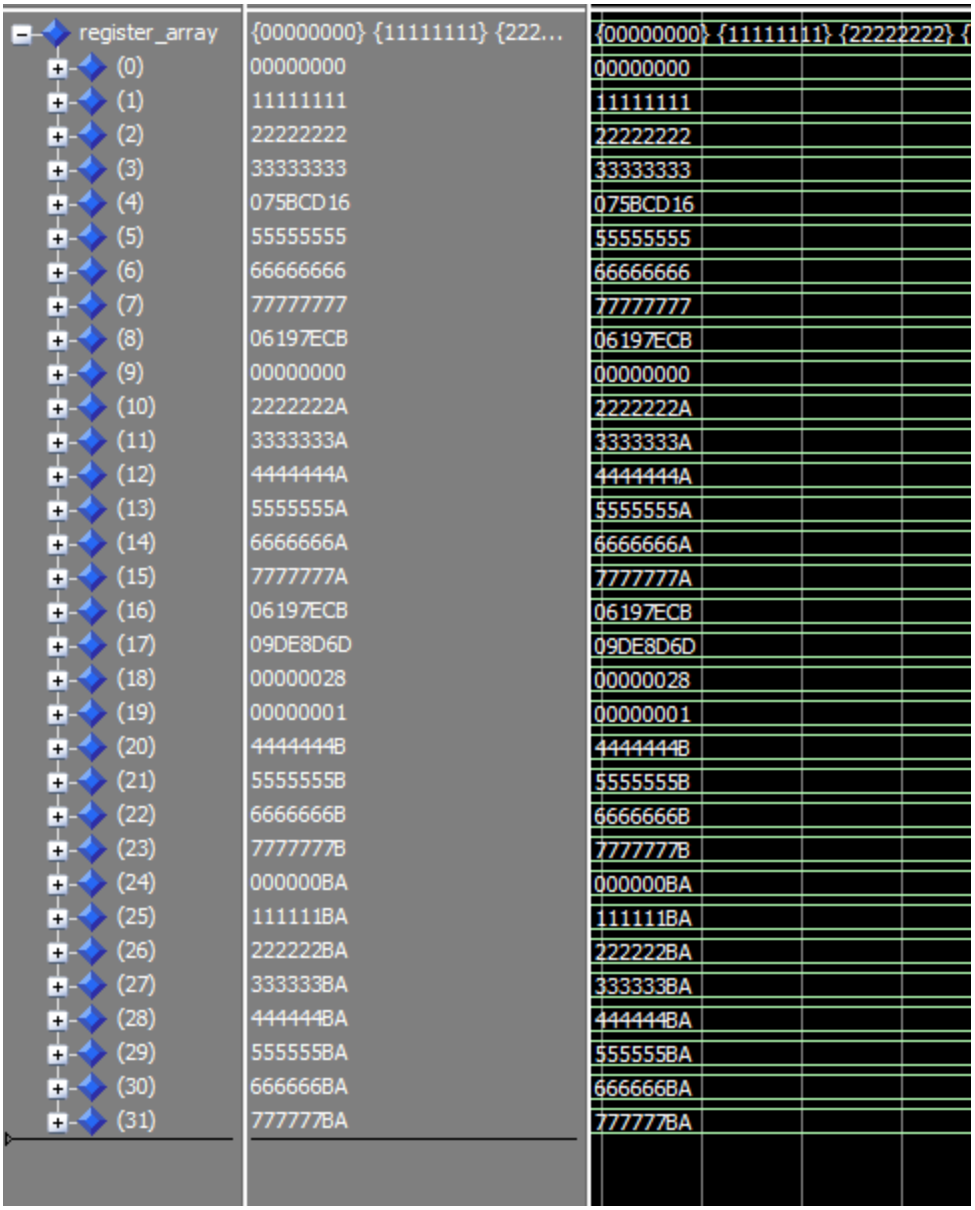


Figure 34: Result waveform of simulation of the test program with forwarding

As we can see, register $18 contains the value 0x28. The amount of cycles to compute with the forwarding of all R type instructions is 494 cycles. This confirms that our forwarding unit is working as intended.

	Cycles to compute nfib(123)	**Cycles to compute nfib(123456789)**
Question 1	218	740
Question 2	182	617
Question 3	146	494

Figure 35: Cycles needed to compute nfib(n), where n = 123 and 123456789

There is a significant speedup from Question 1 to Question 3 as seen in Figure 35.

We save 72 cycles for nfib(123) with a **speedup of 1.4931**

We save 246 cycles for nfib(123456789) with a **speedup of 1.4980**