There are four cycles in our previous CPU design, Fetch, Decode, Execute and Store (aka 'Write back'). In our current cpu design these 4 cycles are performed one at a time, thus our CPU executes an instruction from memory for every 4 cycles of the CPU clock. With a few changes we can increase our performance substantially to almost achieve a single instruction per clock cycle.

There is a common analogy used when explaining CPU pipelining and also where the name pipelining is borrowed from. You can think of each of these cycles like steps in a manufacturing pipeline. Each step performs it's function and the result is passed to the next step. Like a toy that is assembled along a conveyor from station to station, each worker in turn adds a piece to the final assembly. In any moment, there is 1 toy being finished, and N-1 toys partially completed, where N is the number of assembly stations.

A similar semantic is used to reduce the 4 clock cycles per instruction (so N=4) down to an apparent single clock cycle per instruction, albiet with a few caveats we will discuss later. We turn the standard design on it's head, and each "station" of the pipeline performs a single operation each clock cycle on one of the N instructions in the pipeline. Each instruction still takes 4 clock cycles to fetch, decode, execute and write back, however each clock cycle will result in a finished instruction. Seen another way, at any particular instant 3 instructions will be in the process of executing and 1 will complete.

Pipeline Stalls

There are some problems that arise when implementing pipelining. These problems will prevent the CPU from reaching the theoretical maximum cpu speed of 1 instruction for each clock cycle. The problems arise most commonly during branching operations when a change in the PC (program counter) register occurs. In such cases there are partially finished instructions within the pipeline that assumed no branch will occur. In other words, instructions immediately after the branch instruction entered the pipeline and could not have advance knowledge of the outcome of the branch condition. Therefor, if and when a branch occurs, existing operations in the pipeline must be invalidated. There are numerous ways to handle these situations. For now, we will leave this up to the programmer or compiler to always put 3 NOP instructions immediately preceeding any branch instruction. These NOPS will ensure proper cpu behavior no matter the outcome of the branch condition. This essentially makes all branch instructions take 4 cycles, and all other instructions take a single cycle.

We also have issues with two consecutive instructions accessing the same memory if the first instruction intends to write a value back to that memory location. The second instruction may get an old value as it fetched it's data before the write operation finished. Again, as a programmer we can insert a NOP in these situations, or as a compiler writer we can catch them during code generation.

There are also hardware methods to handling pipleline stalls. For example, we can add a single bit to each pipleline step that indicates if the output of that step is valid. Then if the PC changes because of a branch instruction, we can set that bit to invalid consequently turning those pending operations into a NOP. We could also go further and duplicate part of the pipleline so that a branch traverses both outcomes until such time as the condition can be evaluated. The latter would keep with 1:1 instruction per clock timing at the cost of a more complex design requiring more logic.

Verilog Source

There are two areas in our design reguired to implement simple pipelining. (1) We must remove the outer state machine so instead of sequencing through the four instruction cycles each one is executed in parallel. (2) To link each cycle together like a pipelining assembly line we must also add some register storage in between to store the result from each cycle until the next cycle occurs.

```
  module yfcpu(clk, rst, PC);    // our cpu core parameters  parameter im_size = 8;  // 2^n
instruction word memory  parameter rf_size = 4;  // 2^n word register file    // a parameter for the
bus width of our cpu  parameter bw = 8;    input clk; // our system clock  output PC; // our
program counter  input rst; // reset signal    // the cycle states of our cpu, i.e. the Control Unit
states  parameter s_fetch   = 3'b000; // fetch next instruction from memory  parameter s_decode
 = 3'b001; // decode instruction into opcode and operands  parameter s_execute = 3'b010; //
execute the instruction inside the ALU  parameter s_store   = 3'b011; // store the result back to
memory  parameter s_nostore = 3'b100; // dont store any result, skips a cycle    // the parts of
our instruction word  parameter opcode_size = 4;  // size of opcode in bits    // Mnemonic Op
Codes  parameter LRI   = 4'b0001;  parameter ADD   = 4'b0100;  parameter SUB   = 4'b0101;
parameter OR    = 4'b0110;  parameter XOR   = 4'b0111;    // our new branch mnemonics!
parameter BRA    = 4'b1000;   // branch to address in memory location RB  parameter BRANZ
= 4'b1001;   // branch to address in memory location RB, if RA is zero  parameter BRAL   =
4'b1010;   // branch to literal address RB  parameter BRALNZ = 4'b1011;   // branch to literal
address RB, if RA is zero  parameter CALL   = 4'b1100;   // call subroutine; store current PC into
RD and jump to address in literal location RB    parameter HALT  = 4'b1111;        // our memory
core consisting of Instruction Memory, Register File and an ALU working (W) register  reg [
opcode_size + (rf_size*3) -1 : 0 ] IMEM[0: (1 << im_size) -1 ] ; // instruction memory  reg [
bw-1:0 ] REGFILE[0: (1 << rf_size) -1 ]; // data memory  reg [ bw-1:0 ] W; // working
(intermediate) register    // our cpu core registers  reg [ im_size-1 : 0 ] PC;   // program counter
reg [ opcode_size + (rf_size*3) -1 : 0 ] IR; // instruction register    /* Control Unit registers
The control unit sequencer cycles through fetching the next instruction        from memory,
decoding the instruction into the opcode and operands and        executing the opcode in the
ALU.  */  reg [ 2:0 ] current_state;  reg [ 2:0 ] next_state;    // our instruction registers  // an
opcode typically loads registers addressed from RA and RB, and stores  // the result into
destination register RD. RA:RB can also be used to form  // an 8bit immediate (literal) value.
```

```
reg [ opcode_size-1 : 0 ] OPCODE;  reg [ rf_size-1 : 0 ] RA;   // left operand register address
reg [ rf_size-1 : 0 ] RB;   // right operand register address  reg [ rf_size-1 : 0 ] RD;   // destination
register      /* Pipeline flags */  reg F_STORE; // execute cycle sets if W should be written to RD
in writeback cycle  reg F_HALTED; // if set, we halt execution until a reset instruction (in a sim,
we finish)    // the initial cpu state bootstrap  initial begin   PC = 0;   current_state = s_fetch;     //
LOADROM: here we load the rom file generated by our assembler!!!
$readmemh("test.rom",IMEM,0,14);     wait(OPCODE == HALT)
$writememh("regfile",REGFILE);   end          // at each clock cycle we sequence the Control
Unit, or if rst is  // asserted we keep the cpu in reset.  always @ (clk, rst)  begin   if(rst) begin
current_state = s_fetch;    PC = 0;    F_HALTED = 0;    end   else if(!F_HALTED)      begin      //
perform a fetch, decode, execute and writeback cycle concurrently    // FETCH :  fetch
instruction from instruction memory    IR = IMEM[ PC ];    PC = PC + 1;  // increment program
counter (instruction mem pointer)      // DECODE : decode the opcode and register operands
OPCODE = IR[ opcode_size + (rf_size*3) -1 : (rf_size*3) ];    RA = IR[ (rf_size*3) -1 : (rf_size*2)
];    RB = IR[ (rf_size*2) -1 : (rf_size  ) ];    RD = IR[ (rf_size  ) -1 : 0 ];      // EXECUTE : Execute
ALU instruction, process the OPCODE    case (OPCODE)     LRI: begin         // load register RD
with immediate from RA:RB operands        W = {RA, RB};        F_STORE = 1;        end
ADD: begin         // Add RA + RB        W = REGFILE[RA] + REGFILE[RB];        F_STORE = 1;
  end            SUB: begin         // Sub RA - RB        W = REGFILE[RA] - REGFILE[RB];
F_STORE = 1;       end       OR: begin         // OR RA + RB        W = REGFILE[RA] |
REGFILE[RB];        F_STORE = 1;        end            XOR: begin         // Exclusive OR RA ^ RB
  W = REGFILE[RA] ^ REGFILE[RB];        F_STORE = 1;        end            HALT: begin        //
Halt execution, loop indefinately        F_HALTED = 1;        $finish; // quit the sim        end
BRA: begin        // branch to REGFILE[RB]        PC = REGFILE[RB];        F_STORE = 0;
end       BRAL: begin        // branch to RB        PC = RB;        F_STORE = 0;        end
BRANZ: begin        // branch to REGFILE[RB] if REGFILE[RA] is zero        if(REGFILE[RA]
!=8'd0)          PC = REGFILE[RB];        F_STORE = 0;        end       BRALNZ: begin        //
branch to RB if REGFILE[RA] is zero        if(REGFILE[RA] !=8'd0)            PC = RB;
F_STORE = 0;        end            CALL: begin        // call a subroutine; store PC into RD and jump
to location in REGFILE[RB]        W = PC;        PC = RB;        F_STORE = 1;        end            //
catch all : NOP!     default: begin         F_STORE = 0;     end    endcase         // STORE : store
the ALU working register into the destination register RD    if(F_STORE)     REGFILE[RD] = W;
       // move the control unit to the next state   end  end    endmodule
```