

## Project 2: Pipelining (10%)

ENEE 446: Digital Computer Design

### 1. Purpose

This project is intended to help you understand in detail how a pipelined microprocessor works. You will build a pipelined RiSC-16, complete with *data forwarding*, simple *branch prediction*, and *speculative execution*. The next project will add *precise interrupts*. For details on the RiSC-16 pipeline, see the document *The Pipelined RiSC-16* on the class website.

### 2. Pipelines

In the previous project, you built a sequential processor. In that example, the entire instruction is executed before the next clock, at which point the results of the instruction are latched in the register file or data memory. Not surprisingly, this results in a relatively long clock period.

The computer market is not fond of slow clocks, however. Increased clock speeds are possible as the amount of logic between successive latches is decreased. If execution is sliced up into smaller sub-tasks, the clock can run as fast as the longest sub-task. Theoretically, a pipeline of  $N$  stages should run with a clock that is  $N$  times faster than a sequential implementation. For many reasons, this theoretical limit is never reached, due to latch overhead, sub-tasks of unequal length, etc. Nonetheless, extremely fast clock rates are possible. Slicing up the instruction execution this way is called *pipelining*, and it is exploited to great degree in nearly every aspect of modern computer design, from the processor core to the DRAM subsystem, to the overlapping of transactions on memory and I/O buses, etc.

The RiSC-16 pipeline is shown in Fig. 1 on the next page. It is similar to the 5-stage DLX/MIPS pipeline that is described in both *Hennessy & Patterson* and *Patterson & Hennessy*, and it fixes a few minor oversights, such as lack of forwarding to store data, lack of forwarding to comparison logic in decode implementing the 1-instruction delay slot, etc. This pipeline adds in forwarding for store data and eliminates branch delay slots. As in the DLX/MIPS, branches are predicted not taken, though implementations of more sophisticated branch prediction are certainly possible.

In the figure, shaded boxes represent clocked registers; thick lines represent 16-bit buses; thin lines represent smaller data paths; and dotted lines represent control paths. The figure illustrates how pipelining is achieved: the sub-tasks into which instruction execution has been divided are instruction fetch, instruction decode, instruction execute, memory access, and register-file write-back. Each of these sub-tasks, which is executed by dedicated hardware called a *pipeline stage*, produces intermediate results that must be stored before an instruction may move on to the next stage. By breaking up execution into smaller sub-tasks, it is possible to overlap the different sub-tasks of several different instructions simultaneously. If the intermediate results of the various sub-tasks are not stored, they would be lost, as during the next cycle another instruction would be using the same hardware for its own sub-task. For instance, after an instruction is fetched, it is necessary to store the fetched instruction somewhere, because the output of the instruction memory will be different on the following cycle—the fetch stage will be fetching a completely different instruction.

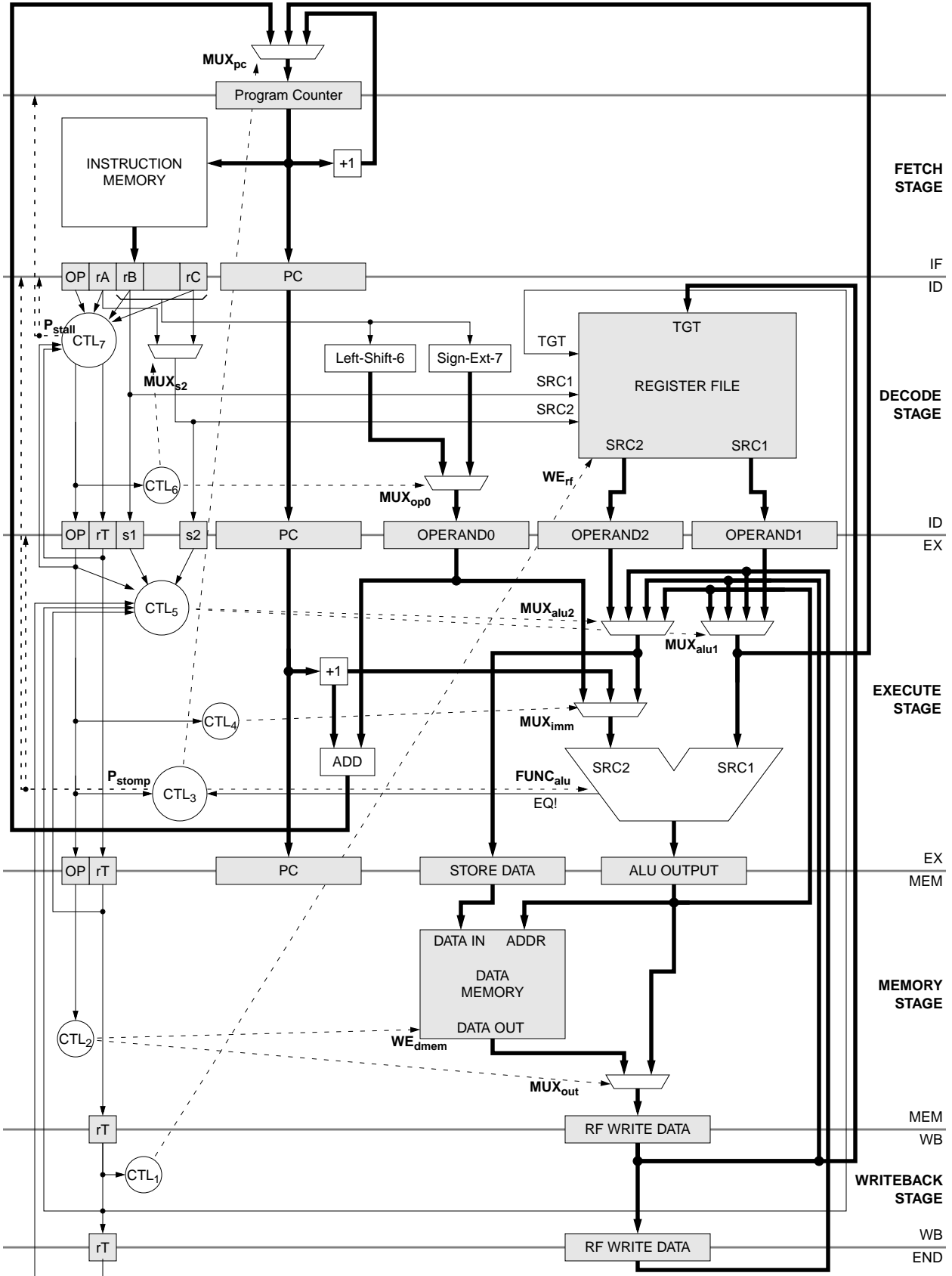


Fig. 1: RISC-16 5-stage pipeline

The storage locations for the intermediate results are called *pipeline registers*, and the figure illustrates their contents. It is common to label a pipeline register with the two stages that it divides. For example, the pipeline register that divides the instruction fetch (IF) and instruction decode (ID) stages is called the *IF/ID register*; the register that divides the instruction execute (EX) and memory-access (MEM) stages is called the *EX/MEM register*; and the register at the end of the register-file writeback (WB) stage could be called the *WB/END register*.

Note that neither the WB/END register nor the data-forwarding path it supports is present in the DLX/MIPS architecture described by Hennessy & Patterson. The DLX/MIPS assumes a half-cycle register-file access, so that the writeback stage completes in the first half of the cycle and the register-file read component of the decode stage happens in the second half of the cycle. This allows data to be forwarded from the writeback stage to the decode stage directly. Otherwise such forwarding is impossible, unless the register file has a pass-through design that connects data-in to data-out whenever reading and writing the same register. If the register file does not do such forwarding, then the data written to the register file is only available on the following cycle. Thus, there must be a path to forward data to the instruction in the decode stage at the same time as the instruction writing to the register file in the writeback stage. This is the function of the WB/END register and the forwarding stage it represents.

### 3. RiSC-16 Pipeline Registers

<b>Program Counter</b>	The address of the instruction currently being fetched.
<b>IF/ID Register:</b>	
<b>INSTR</b>	The instruction to execute, with opcode, rA, rB, rC, and immediate fields.
<b>PC</b>	Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.
<b>ID/EX Register:</b>	
<b>OP</b>	Contains the instruction opcode.
<b>rT, s1, s2</b>	Contains the instruction's 3-bit register specifiers rA, rB, and rC—s1 is equal to rB. For SW and BEQ instructions, s2 is taken from the instruction's rA field, otherwise it is taken from the instruction's rC field. The rT field contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).
<b>PC</b>	Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.
<b>OPERAND0</b>	Contains the instruction's immediate operand. If the instruction uses a shifted or sign-extended immediate value (ADDI, LUI, LW, SW, BEQ), that value is available immediately and is stored here.
<b>OPERAND1</b>	Contains the instruction's first register operand; this is the contents of the register <i>register-file[rB]</i> .
<b>OPERAND2</b>	Contains the instruction's second register operand. For ADD and NAND instructions, it is the contents of <i>register-file[rC]</i> . For BEQ and SW instructions, it is the contents of <i>register-file[rA]</i> .
<b>EX/MEM Register:</b>	
<b>OP</b>	Contains the instruction opcode.
<b>rT</b>	Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).
<b>PC</b>	Contains the address of the instruction whose state is contained in this pipeline register. This is used by BEQ and JALR instructions and in handling pipeline interrupts.
<b>STORE DATA</b>	Contains the data to store to DATA MEMORY. Note that if the instruction is not a SW, this information is not used.
<b>ALU OUTPUT</b>	Contains the most recent output of the ALU.
<b>MEM/WB Register:</b>	
<b>rT</b>	Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).
<b>RF WRITE</b>	Contains the data that will be written to the register file on the following cycle (provided the rT register has a non-zero value).
<b>WB/END Register:</b>	
<b>rT</b>	Contains the instruction's 3-bit target-register identifier, or the binary value 000 if the instruction has no target (i.e. SW and BEQ instructions).
<b>RF WRITE</b>	Contains the data that was written to the register file on the previous cycle (provided the rT register has a non-zero value).

## 4. Verilog Implementation

You have been given a skeleton Verilog file that looks like this:

```
// RiSC-16 ... project 2 skeleton
`define ADD          3'd0
`define ADDI        3'd1
`define NAND        3'd2
`define LUI         3'd3
`define SW          3'd4
`define LW          3'd5
`define BEQ         3'd6
`define JALR        3'd7
// JALR and EXTEND are synonyms
`define EXTEND      3'd7

// extended sub-classes:
`define EXTEND_NONE 3'd0
`define EXTEND_EXC  3'd7

// exception types:
`define EXC_NONE    4'd0
`define EXC_HALT    4'd1

`define INSTRUCTION_OP 15:13 // opcode
`define INSTRUCTION_RA 12:10 // rA
`define INSTRUCTION_RB 9:7   // rB
`define INSTRUCTION_RC 2:0   // rC
`define INSTRUCTION_IM 6:0   // immediate (7-bit)
`define INSTRUCTION_LI 9:0   // large immediate (10-bit, 0-extended)
`define INSTRUCTION_SB 6     // immediate's sign bit
`define INSTRUCTION_E1 6:4   // the extended sub-class (NONE, EXC, etc.)
`define INSTRUCTION_E2 3:0   // the extended sub-class identifier value

`define HALTINSTRUCTION { `EXTEND, 3'd0, 3'd0, 3'd7, 4'd1 }
`define ZERO             16'd0

module RiSC (clk);
    input      clk;

    // global data structures
    reg [15:0] m[0:65535];
    reg [15:0] rf[0:7];
    reg [15:0] pc;

    // pipeline registers
    reg [15:0] IFID_instr;
    reg [15:0] IFID_pc;

    reg [2:0]  IDEX_op;
    reg [2:0]  IDEX_rT;
    reg [2:0]  IDEX_sl;
    reg [2:0]  IDEX_s2;
    reg [15:0] IDEX_pc;
    reg [15:0] IDEX_arg0;
    reg [15:0] IDEX_arg1;
    reg [15:0] IDEX_arg2;

    reg [2:0]  EXMEM_op;
    reg [2:0]  EXMEM_rT;
    reg [15:0] EXMEM_pc;
    reg [15:0] EXMEM_stdata;
    reg [15:0] EXMEM_aluout;

    reg [2:0]  MEMWB_rT;
    reg [15:0] MEMWB_rfdata;

    reg [2:0]  WBEND_rT;
    reg [15:0] WBEND_rfdata;

    // aliases helpful for decoding the fetched instruction
    wire [2:0] IFID_op  = IFID_instr[ `INSTRUCTION_OP];
    wire [2:0] IFID_rA  = IFID_instr[ `INSTRUCTION_RA];
    wire [2:0] IFID_rB  = IFID_instr[ `INSTRUCTION_RB];
    wire [2:0] IFID_rC  = IFID_instr[ `INSTRUCTION_RC];
    wire       IFID_sb  = IFID_instr[ `INSTRUCTION_SB];
    wire [15:0] IFID_imm7 = { {9 { IFID_sb }}, IFID_instr[ `INSTRUCTION_IM]};
    wire [15:0] IFID_imm10 = {IFID_instr[ `INSTRUCTION_LI], 6'd0};

    // muxes and control lines ... roughly top-down from figure 1
    wire      Pstall;
    wire      Pstomp;
    wire [15:0] MUXpc_out;
    wire      WErf;
    wire [2:0] MUXs2_out;
    wire [15:0] MUXop0_out;
    wire [15:0] MUXalu1_out;
    wire [15:0] MUXalu2_out;
    wire [15:0] MUXimm_out;
    wire      WEdmem;
    wire [15:0] MUXout_out;
    wire [3:0] FUNCalu = ((IDEX_op == `ADDI) || (IDEX_op == `SW) || (IDEX_op == `LW))
        ? `ADD
        : IDEX_op;

```

```

integer j;
initial begin
    pc = 0;

    IFID_instr = 0;
    IFID_pc = 0;
    IDEX_op = 0;
    IDEX_rT = 0;
    IDEX_s1 = 0;
    IDEX_s2 = 0;
    IDEX_pc = 0;
    IDEX_arg0 = 0;
    IDEX_arg1 = 0;
    IDEX_arg2 = 0;
    EXMEM_op = 0;
    EXMEM_rT = 0;
    EXMEM_pc = 0;
    EXMEM_stdata = 0;
    EXMEM_aluout = 0;
    MEMWB_rT = 0;
    MEMWB_rfdata = 0;
    WBEND_rT = 0;
    WBEND_rfdata = 0;

    rf[0] = `ZERO;
    rf[1] = `ZERO;
    rf[2] = `ZERO;
    rf[3] = `ZERO;
    rf[4] = `ZERO;
    rf[5] = `ZERO;
    rf[6] = `ZERO;
    rf[7] = `ZERO;
    for (j=0; j<65536; j=j+1)
        m[j] = 0;
end

always @(negedge clk) begin
    rf[0] <= `ZERO;
end

always @(posedge clk) begin: fetch_stage
end

always @(posedge clk) begin: decode_stage
end

always @(posedge clk) begin: execute_stage
    .
    .
    //
    // note -- i have defined FUNCalu so that the instructions
    // that need ADD operations (ADD, ADDI, LW, SW) are simply ADDs
    //
    case (FUNCalu)
        ADD:EXMEM_aluout    <= MUXimm_out + MUXalul_out;
        `NAND:EXMEM_aluout  <= ~(MUXimm_out & MUXalul_out);
        `LUI:EXMEM_aluout   <= MUXimm_out;
        `BEQ:EXMEM_aluout   <= MUXimm_out ^ MUXalul_out;
        `EXTEND:            if ((IDEX_arg0[`INSTRUCTION_E1] == `EXTEND_EXC)
                                && (IDEX_arg0[`INSTRUCTION_E2] == `EXC_HALT)) begin
                                $finish;
                            end
                            else begin
                                EXMEM_aluout <= MUXimm_out;
                            end
        default:            begin
                                EXMEM_aluout <= 16'hDEAD;
                            end
    endcase
    .
    .
end

always @(posedge clk) begin: memory_stage
end

always @(posedge clk) begin: writeback_stage
end

always @(posedge clk) begin
    $display("    pc - %h", pc);
    $display(" IFID - %h %h", IFID_instr, IFID_pc);
    $display(" IDEX - %h %h %h %h %h %h %h %h", IDEX_op, IDEX_rT, IDEX_s1, IDEX_s2, IDEX_pc,
                                                IDEX_arg0, IDEX_arg1, IDEX_arg2);
    $display("EXMEM - %h %h %h %h %h", EXMEM_op, EXMEM_rT, EXMEM_pc, EXMEM_stdata, EXMEM_aluout);
    $display("MEMWB - %h %h", MEMWB_rT, MEMWB_rfdata);
    $display("WBEND - %h %h", WBEND_rT, WBEND_rfdata);
end
endmodule

```

The skeleton file includes definitions for all data structures that you will need, both registers and busses. It contains definitions for all of the pipeline registers shown in Figure 1, and it contains definitions for many of the control lines (like  $P_{\text{stall}}$  and  $P_{\text{stomp}}$ ) and multiplexer outputs. The names are roughly equivalent to each other, which should make it relatively simple to translate from one of the other.

Note that the document *The Pipelined RiSC-16* on the class website describes the control signals and CTL modules in detail.

Your job is to connect everything together—for example, on the positive edge of the clock, the program counter  $pc$  should latch a new value, unless the  $P_{\text{stall}}$  signal is high. The value to latch comes from the output of the  $MUX_{pc}$  multiplexer. Therefore, in the fetch stage of your code (PC update is typically included with fetch) you would have the following logic:

```
always @(posedge clk) begin: fetch_stage
    if (!Pstall) pc <= MUXpc_out;
end
```

This indicates that the program counter is to latch a new value on every cycle, unless the  $P_{\text{stall}}$  signal is high. Note that there is no conflict with the  $P_{\text{stomp}}$  signal (which would override the stall signal) because the stomp signal is only set if there is a BEQ or JALR in the execute stage, whereas the stall signal is only set when a LW is in the execute stage.

**Notes:** you may not use the blocking assignment operator (use “<=” instead of “=” in procedural assignments), and you may not use delay statements (e.g. “#1” sprinkled through the code).

## 5. In Comparison to DLX/MIPS

For the most part, you can follow the discussion in the textbook for a description of how the pipeline works. For example, as in the DLX/MIPS pipeline, the RiSC-16 implements load-use interlocks: whenever there is a LW in the ID/EX register and an instruction in the IF/ID register that uses the result of the load as an operand, the pipeline stalls one cycle and inserts a NOP into the ID/EX register following the LW.

The RiSC-16 pipeline differs slightly from Hennessy & Patterson in the treatment of branches and jumps: in the RiSC-16 pipeline, unlike DLX/MIPS, there is no *delay slot*. Instead, every taken branch incurs a two-cycle penalty—this is the function of the STOMP signal: it turns into NOPs the two instructions behind the BEQ/JALR in the execute stage. When a JALR or a taken BEQ is in the ID/EX register, the ID/EX register and IF/ID register should both latch NOP instructions at the end of the cycle. As in DLX/MIPS, branches are predicted not-taken, and no penalty is incurred if this prediction turns out to be correct.