CSC3050 Project 3

Overview

In Project 3, a 5-stage pipelined CPU was implemented using Verilog language to support the MIPS ISA. Some changes are performed to the pipeline structure from the structure in the project description:

- The field alignment and sign extension originally performed in ID are advanced to IF.
- One more pipeline register is added at the beginning of the IF stage to store the predicted value of the PC, which is more convenient for handling the hazards.
- The huge control unit is divided into multiple small combinational blocks, which significantly reduces the memory requirements of the pipeline register.

Note: all hazards are solved.

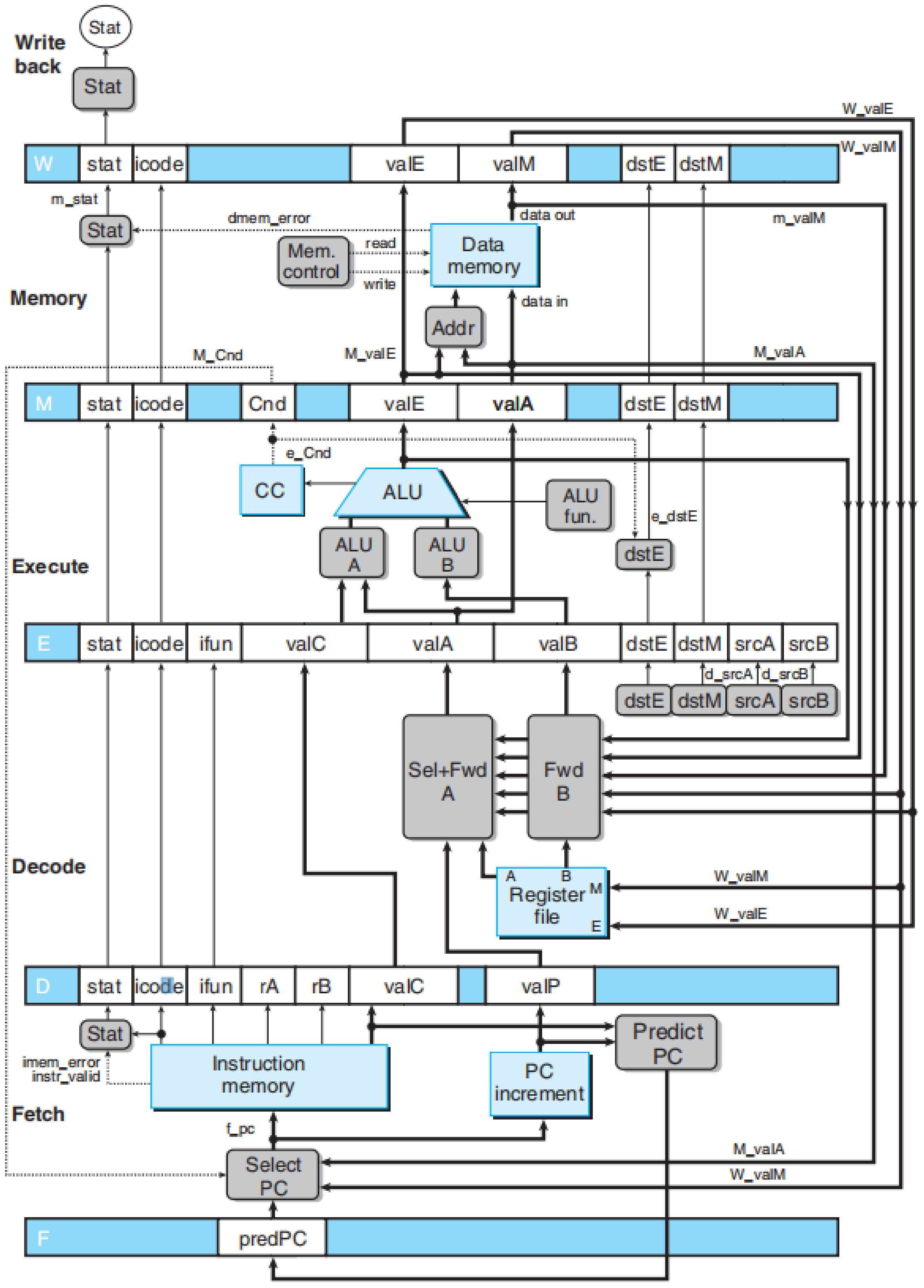
Usage

Here is an example for testing the CPU using test case 1.

```
yyyx@HP-DESKTOP:~/proj3/src/cpu$ make compile
iverilog -o CPU test_cpu.v
yyyx@HP-DESKTOP:~/proj3/src/cpu$ make test1
cp /home/yyyx/proj3/testcase/cpu_test/machine_code1.txt instructions.bin
vvp CPU
WARNING: ./InstructionRAM.v:38: $readmemb(instructions.bin): Not enough words in the file for the requested range [0:511].
VCD info: dumpfile test.vcd opened for output.
The execution take  56 clock cycles
After execution, the register file:
** VVP Stop(0) **
** Flushing output streams.
** Current simulation time is 118200000 ticks.
> finish
** Continue **
cmp data.bin /home/yyyx/proj3/testcase/cpu_test/DATA_RAM1.txt
```

Data flow

The architecture is analogous to the architecture introduced in CSAPP Chapter 4, which is shown below.



Implementation highlights

PC selection and prediction

The PC selection block chooses the right PC value among all sources, including misprediction handling, and jump targets of `jr`, `j`, and `jal`. For PC prediction, the *always-taken* strategy is used for `beq` and `bne`, while the three jump instructions are always regarded as *not taken*.

```
assign f_pc =
        // misprediction
        (M_opcode == OP_BEQ | M_opcode == OP_BNE) & ~M_Cnd ? M_valP :
        // jr
        (E_opcode == OP_R & E_func == FUNC_JR) ? E_valA :
        // j, jal
        (M_opcode == OP_J | M_opcode == OP_JAL) ? M_valE :
        F_pred_pc;

assign f_pred_pc =
        f_opcode == OP_BEQ | f_opcode == OP_BNE ?
        f_valP + (f_sign_extended_imm << 2) :
        f_valP;
```

Register

There are multiple types of registers (such as `PipelinedReg` and `FlagsReg`) to serve different functionalities. All these registers are derived from the basic register called `BasicReg`, which is *clocked register with enable signal and synchronous reset*.

```
// Clocked register with enable signal and synchronous reset
module BasicReg(out, in, enable, reset, resetval, clock);
    parameter width = 8;
    output [width-1:0] out;
    reg [width-1:0] out;
    input [width-1:0] in;
    input enable;
    input reset;
    input [width-1:0] resetval;
    input clock;

    always
        @(posedge clock)
        begin
            if (reset)
                out <= resetval;
            else if (enable)
                out <= in;
        end
endmodule
```

The register file uses `BasicReg` to store values of general-purpose registers. One trick to synchronous the reading and writing for the register file and the memory is to perform read or write **on the falling edge of the clock signal**. The reason is that all signals are updated at the rising edge of the clock signal so that there exists uncertainty due to the propagation delay of the combinational blocks. The register file has two read ports and one write port, i.e., only one of `valM` and `valE` will be written into the register file. To determine whether the read or write is desired, `REG_NONE = 5'b00000` is used to indicate the operation is not enabled, since `$zero` cannot be written and always read as zero.

```
module RegisterFile(clock, reset, srcA, srcB, dstE, valE, dstM, valM,
                    valA, valB, zero, at, v0, v1, a0, a1, a2, a3, t0,
                    t1, t2, t3, t4, t5, t6, t7, s0, s1, s2, s3, s4,
                    s5, s6, s7, t8, t9, k0, k1, gp, sp, fp, ra);
    input clock, reset;
    input [4:0] srcA, srcB;
    input [4:0] dstE, dstM;
    input [31:0] valE, valM;
    output [31:0] valA, valB;

    // Make every registers visible for debugging
    output [31:0] zero, at, v0, v1, a0, a1, a2, a3, t0, t1, t2, t3, t4,
                  t5, t6, t7, s0, s1, s2, s3, s4, s5, s6, s7, t8, t9, k0,
                  k1, gp, sp, fp, ra;
    ...
endmodule
```

ALU

The ALU receives two ALU operands and one 4-bit vector indicating the function that ALU needs to perform. Then ALU will output the result and the 3-bit flag updated by the calculation.

```
module ALU(aluA, aluB, aluFunc, result, flags);
    input signed [31:0] aluA;
    input signed [31:0] aluB;  // two operands
    input [3:0] aluFunc;  // operation type
    output [31:0] result;  // calculation result
    output [2:0] flags; // carry, zero, negative, overflow
    ...
endmodule
```

Hazard control

data hazard (forwarding)

All data hazards except the load/use hazard can be handled by **forwarding**, which is located in the decode stage. In combinational logic for selecting `d_valA` and `d_valB`, any `_valE` or `_valM` in each stage will take the place of the value read from the register file if their destination is the decoding source. The section logic for `d_valA` is shown below, and the logic is exactly the same for `d_valB`:

```
assign d_valA =
        d_srcA == REG_NONE ? 0 :
        d_srcA == E_dstE ? e_valE :
        d_srcA == M_dstM ? m_valM :
        d_srcA == M_dstE ? M_valE :
        d_srcA == W_dstM ? W_valM :
        d_srcA == W_dstE ? W_valE :
        d_rvalA;
```

special hazard (stall and bubble)

There are in total three hazards that need to be specially considered:

- The load/use hazard: stall in fetch and decode stage pipelined registers.
- Misprediction: bubble in decode and execute stage pipelined registers.
- Jump instructions: stall in fetch stage pipelined registers and bubble in decode stage pipelined registers. The jump target will be obtained in execute stage and selected by the PC section in the next cycle.

Each stage has `_stall` and `_bubble` signals that control the behavior of the pipelined registers. These signals are considered as global control signals since they need the status from the other stages.

Stop detection

Each stage will maintain a `_stat` signal to indicate the status of the processor. When the stop instruction `32'hFFFFFFFF` is fetched in the fetch stage, `f_stat` will become `STAT_STOP` and propagate through the pipeline to the write-back stage. After that, since only the memory stage and the write-back stage will change the status of the processer, the pipelined registers in the memory stage will be injected bubble consistently and the pipelined registers in the write-back stage will be stalled.

Performance evaluation

The following table shows the number of clock cycles for executing each test case.

Test No.	The number of clock cycles	Test No.	The number of clock cycles
1	56	5	161
2	15	6	57
3	18	7	45
4	17	8	29