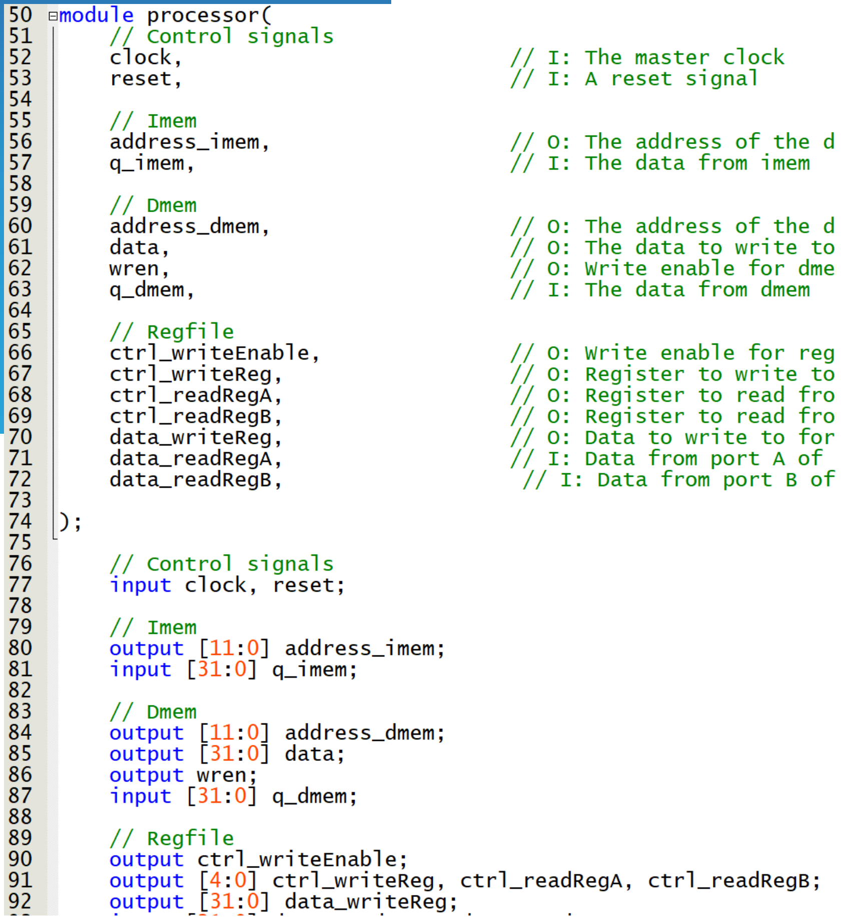


```
module processor(
    // Control signals
    clock,                          // I: The master clock
    reset,                          // I: A reset signal

    // Imem
    address_imem,                   // O: The address of the d
    q_imem,                         // I: The data from imem

    // Dmem
    address_dmem,                   // O: The address of the d
    data,                           // O: The data to write to
    wren,                           // O: Write enable for dme
    q_dmem,                         // I: The data from dmem

    // Regfile
    ctrl_writeEnable,               // O: Write enable for reg
    ctrl_writeReg,                  // O: Register to write to
    ctrl_readRegA,                  // O: Register to read fro
    ctrl_readRegB,                  // O: Register to read fro
    data_writeReg,                  // O: Data to write to for
    data_readRegA,                  // I: Data from port A of
    data_readRegB,                   // I: Data from port B of

);

    // Control signals
    input clock, reset;

    // Imem
    output [11:0] address_imem;
    input [31:0] q_imem;

    // Dmem
    output [11:0] address_dmem;
    output [31:0] data;
    output wren;
    input [31:0] q_dmem;

    // Regfile
    output ctrl_writeEnable;
    output [4:0] ctrl_writeReg, ctrl_readRegA, ctrl_readRegB;
    output [31:0] data_writeReg;
```

ECE 350 Pipelined Processor

Luke Truitt

11/5/2019

lot

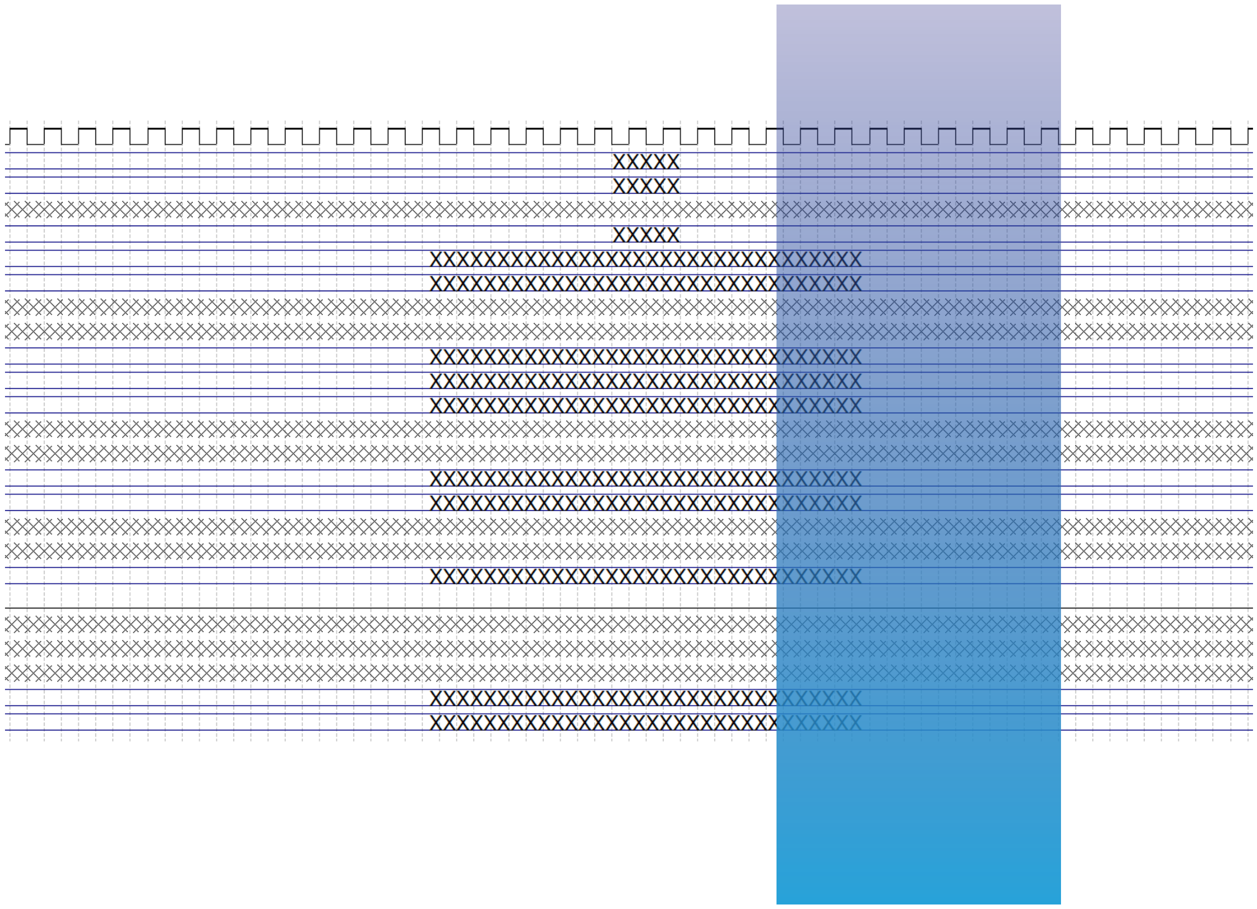
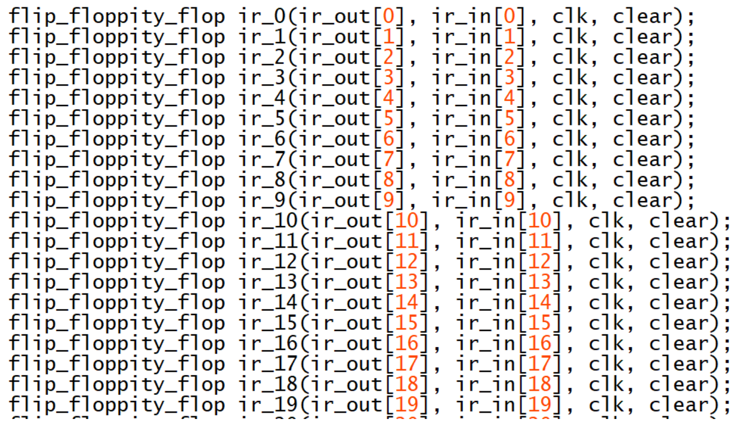


Table of Contents



```
flip_floppity_flop ir_0(ir_out[0], ir_in[0], clk, clear);
flip_floppity_flop ir_1(ir_out[1], ir_in[1], clk, clear);
flip_floppity_flop ir_2(ir_out[2], ir_in[2], clk, clear);
flip_floppity_flop ir_3(ir_out[3], ir_in[3], clk, clear);
flip_floppity_flop ir_4(ir_out[4], ir_in[4], clk, clear);
flip_floppity_flop ir_5(ir_out[5], ir_in[5], clk, clear);
flip_floppity_flop ir_6(ir_out[6], ir_in[6], clk, clear);
flip_floppity_flop ir_7(ir_out[7], ir_in[7], clk, clear);
flip_floppity_flop ir_8(ir_out[8], ir_in[8], clk, clear);
flip_floppity_flop ir_9(ir_out[9], ir_in[9], clk, clear);
flip_floppity_flop ir_10(ir_out[10], ir_in[10], clk, clear);
flip_floppity_flop ir_11(ir_out[11], ir_in[11], clk, clear);
flip_floppity_flop ir_12(ir_out[12], ir_in[12], clk, clear);
flip_floppity_flop ir_13(ir_out[13], ir_in[13], clk, clear);
flip_floppity_flop ir_14(ir_out[14], ir_in[14], clk, clear);
flip_floppity_flop ir_15(ir_out[15], ir_in[15], clk, clear);
flip_floppity_flop ir_16(ir_out[16], ir_in[16], clk, clear);
flip_floppity_flop ir_17(ir_out[17], ir_in[17], clk, clear);
flip_floppity_flop ir_18(ir_out[18], ir_in[18], clk, clear);
flip_floppity_flop ir_19(ir_out[19], ir_in[19], clk, clear);
```

Processor Description

The following is an outline of how I implemented the 5-Stage Pipelined processor with bypassing.

1. ALU

My ALU has the standard input/outputs, with the Adder implemented using the Carry Look-Ahead Method. It is comprised of four, 8-bit CLAs which are connected in a CLA fashion. There is a series of gates to check for overflow, this is potentially an area that can be simplified if I were to spend more time improving the ALU.

2. RegFile

The Register File is a set of 32 Registers each implemented using 32 DFFs. The control signals into the Register File are decoded using the module “decode532” which stands for 5 to 32 bit decode. Each of the register read ports are filled by the outputs of 32 tristate buffers in parallel. Finally, the write data is piped into every register, but the clock for each is combined with the control write signal to prevent from writing to multiple registers at a time.

3. MultDiv

Currently, the processor implements the behavioral component, but that is only because of a one off error in the way the data was being released. Below I describe the fully functional multiplier that would have been used given more time.

The Multiplier was implemented using Booth’s Algorithm. The algorithm is relatively straightforward in implementation. There is a register that hold the multiplicand, a register to hold the multiplier, and then two 32-bit registers to hold the top and bottom halves of the product. Booth’s Algorithm works by either adding or subtracting a power of two of the multiplicand to the top 32 bits of the product based on a shifting version of the multipliers last two bits.

The Divider was implemented using the simple 64-bit divisor, 64-bit dividend, 64-bit remainder circuit, where each clock cycle, a multiple of two of the divisor is subtracted from the dividend, if the result is positive, a one is added to the quotient reg, otherwise, the subtraction is undone and the next multiple of two is tried.

4. Controls

There are three main control blocks to discuss: the hazards and bypassing control block, the exception control block, and the traditional processor controls.

The hazards/bypassing control block is comprised of three bypass blocks (mx_bypass, wx_bypass, and wm_bypass) which control the signals related to bypassing.

The exception control block is a single module that takes in an instruction, the ALU not equal signal and the MultDiv data exception signal and returns the instruction the results from exception handling. If no exception has occurred, the original instruction passes through, otherwise some form of the setx command with the appropriate values.

The traditional processor controls are spread out throughout but are responsible for ensuring the correct registers are read/written to, the correct pc is taken, the write enable signal for both DMEM and the Register File are correctly set, that the correct registers are used through computation, and that the correct register is written to data memory. This occurs over 9 modules: pc_ctrl_sel, reg__write_ctrl, b_ctrl, wren_ctrl, alu_data_ctrl, alu_opcode_ctrl, itypectrl, branch_ctrl, and reg_read_ctrl.

5. Pipeline Registers

The processor implements a 5-stage data pipeline with the five stages PC, Fetch/Decode, Decode/Execute, Execute/Memory, and Memory/Write. These five stages are placed between each of the major components of the processor and allow for at most 5 instructions to be in operation at anytime. I created separate registers for each stage, named “stage” then _reg. The logic for what data needs to be written into each register is accounted for by the traditional processor controls.

In the instances where control hazards occur, the pipeline will need to be flushed, and this is shown with a muxed input to the “_ir” component of each relevant register.

Data hazards are almost entirely taken care of by bypassing. This results in no extra steps needing to be taken with regard to the pipeline registers.

ALU

ADD –

Implementation: The ADD instruction is passed in, the appropriate two register are read from, those values are passed into the ALU with the opcode for ADD generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: This was the easiest implementation with respect to the other ALU operations. It allowed me to reuse almost all of the same control signals other than the ALU opcode.

Tests: The tests for this were primarily around doing many of the operations and trying to write to as many different registers as possible. I would add many different registers to one another then take those newly written values and add them again.

Challenges: The only real challenges here were in bypassing, the initial implementation was straight forward, but it got a bit more complicated as I tried to bypass both mx and wx.

ADDI –

Implementation: The ADDI instruction is passed in, the appropriate register is read from, the register value is passed into the ALU with the B input to the ALU the sign-extended immediate and the opcode for ADD generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: This implementation allowed me to reuse some of the same hardware as the ALU operations, just needed an extra mux and control logic for the sign-extended immediate and the ALU opcode.

Tests: Same as ADD.

Challenges: Bypassing was a challenge here as well, but it was also initially difficult getting the add opcode into the ALU. It required extra control logic that I had forgotten to include.

SUB –

Implementation: The SUB instruction is passed in, the appropriate registers are read from, the register values are passed into the ALU with the opcode for SUB generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: Same as ADD.

Tests: Same as ADD. Also used ADD and ADDI in series with SUB to try maneuvering the register values correctly.

Challenges: Verifying the subtraction worked. This really wasn't too challenging, more of a delay, but I had more trouble verifying negative numbers were what I expected them to be than positive numbers.

AND –

Implementation: The AND instruction is passed in, the appropriate registers are read from, the register values are passed into the ALU with the opcode for AND generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: Same as ADD.

Tests: Piped a bunch of AND instructions through and verified their outputs were correct.

Challenges: NONE.

OR –

Implementation: The OR instruction is passed in, the appropriate registers are read from, the register values are passed into the ALU with the opcode for OR generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: Same as ADD.

Tests: Piped a bunch of OR instructions through and verified their outputs were correct.

Challenges: NONE.

SLL –

Implementation: The SLL instruction is passed in, the appropriate registers are read from, the register values are passed into the ALU with the opcode for SLL generated by the alu_opcode_ctrl along with the SHFT_AMT, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: Same as ADD.

Tests: Piped a bunch of SLL and SRA instructions through and verified watched the registers bounce back and forth between values.

Challenges: NONE.

SRA –

Implementation: The SRA instruction is passed in, the appropriate register is read from, the register values are passed into the ALU with the opcode for SUB generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File.

Why this Implementation: Same as ADD.

Tests: Same as SLL.

Challenges: NONE.

MULTDIV

MULT –

Implementation: The MULT instruction is passed in, the appropriate registers are read from, the register values are passed into the MultDiv with ctrl_MULT generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File. While the Multiplier is working (32 clock cycles), the instruction is stored in a register and the pipeline freezes behind the MULT. NOPs are inserted into the pipeline after the MULT until it has finished, then it resumes where it left off.

Why this Implementation: The MULT will take 32-cycles to successfully complete, and because of this we need to basically freeze the entire pipeline while the MultDiv chugs along. This implementation was the most straightforward method to doing this.

Tests: Called MULTS in different orders before and after other instructions to see how it'd handle the delay, tried overflowing it as well.

Challenges: The delayed IR, using my own MultDiv I was always one clock cycle off and it resulted in an exception being thrown even on basic calculations. This resulted in using the behavioral component for MultDiv.

DIV –

The DIV instruction is passed in, the appropriate registers are read from, the register values are passed into the MultDiv with ctrl_DIV generated by the alu_opcode_ctrl, the output is sent to the memory stage before being written back to Register File. While the Multiplier is working (32 clock cycles), the instruction is stored in a register and the pipeline freezes behind the DIV. NOPs are inserted into the pipeline after the DIV until it has finished, then it resumes where it left off.

Why this Implementation: The DIV will take 32-cycles to successfully complete, and because of this we need to basically freeze the entire pipeline while the MultDiv chugs along. This implementation was the most straightforward method to doing this.

Tests: Called DIVS in different orders before and after other instructions to see how it'd handle the delay, tried dividing by zero as well.

Challenges: The delayed IR, using my own MultDiv I was always one clock cycle off and it resulted in an exception being thrown even on basic calculations. This resulted in using the behavioral component for MultDiv.

MEMORY

SW –

Implementation: The SW instruction is passed in, the appropriate registers are read from, the register value from the second register is passed into the ALU with the B input to the ALU the sign-extended immediate and the opcode for ADD generated by the alu_opcode_ctrl, the output is sent to the memory stage with Data_WriteEnabled on and writing the first register argument to DMEM before being going back to Register File with reg_writeEnable being off.

Why this Implementation: Doing this allowed for me to use the same control blocks as my ALU commands, with the additional control blocks needed for storing the data and reordering the registers being written in.

Tests: Loading and storing data on initialized DMEMs as well as blank ones. Trying stores right after loads that rely on WM bypassing.

Challenges: Switching around the registers that were called. Normally, $rd was used for writing, so there was logic necessary to get the value out of $rd and then pipe it into data.

LW –

Implementation: The LW instruction is passed in, the appropriate register is read from, the register value from the second register is passed into the ALU with the B input to the ALU the sign-extended immediate and the opcode for ADD generated by the alu_opcode_ctrl, the output is sent to the memory stage with Data_WriteEnabled off and reading the ALU output address from DMEM and writing it back to $rd in the Register File.

Why this Implementation: After doing SW, this required only two extra control signals, which made it very simple. Most of the ALU logic could be reused, the only signal needed was for enabling writing to the Register File.

Tests: Loading and storing data on initialized DMEMs as well as blank ones. Trying stores right after loads that rely on WM bypassing.

Challenges: The major challenge here was WM bypassing, otherwise, this was done after SW and ALU so it used mostly existing hardware.

JUMP

J –

Implementation: The J instruction is passed in, the it goes through the Decode stage without doing anything, the instruction goes through a jump check and the last 27 bits are prepended with zeros, then sent to the PC Register along with a signal to select it and flush the pipeline.

Why this Implementation: I had one central branch controller, it took in the instruction in Execution every time and when necessary, assigned a new PC and branch bit. This implementation worked well for jump, as the PC could easily be reassigned.

Tests: All the Jump instructions were tested together in more robust test, but in simple tests, the goal was to jump over instructions modifying specific registers and checking to make sure those registers were unmodified.

Challenges: Correctly reassigning the PC. There was a bug with which it would either jump one too far or one not far enough, but this was fixed by passing in PC+1 instead of PC.

BNE –

Implementation: The BNE instruction is passed in, the it goes through the Decode stage and grabs the appropriate two registers, the instruction goes through a branch check with the output of ALU not equal and if necessary, the PC + 1 + the branch is sent to the PC Register along with a signal to select it and flush the pipeline.

Why this Implementation: I had one central branch controller, it took in the instruction in Execution every time and when necessary, assigned a new PC and branch bit. This implementation worked well for BNE if I passed in the ALU not equal, as the PC could easily be reassigned.

Tests: Create a few scenarios where two registers may or may not be equal and run them through the BNE to see if it correctly jumps and then flushes the pipeline.

Challenges: Assigning the ALU opcode to subtraction to make sure that the ALU not equal signal is correct.

JAL –

Implementation: The JAL instruction is passed in, the it goes through the Decode stage without doing anything, the instruction goes through the same steps as J however once it gets to the Execute stage, the current value of the PC is piped into the first load data position and is eventually written back into the final register.

Why this Implementation: It reused everything from jump with one additional piece of hardware to write back the current PC.

Tests: Call JAL, test the value of $r31, than use JR to try to jump to that location.

Challenges: Rerouting the data out to point to the PC instead of the ALU or MULT out.

JR –

Implementation: The JR instruction is passed in, the it goes through the Decode stage to get the register value to jump to, the instruction goes through the same steps as J however instead of writing to the value of the end of the instruction, it pushes the register value in as a replacement.

Why this Implementation: It reused everything from jump with one additional piece of hardware to overwrite the PC with the selected register.

Tests: Call JAL, test the value of $r31, than use JR to try to jump to that location. Load up various values into registers and try jumping to different spots in the program.

Challenges: Rerouting the register out data into the PC.

BLT –

Implementation: The BLT instruction is passed in, the it goes through the Decode stage to get the two register values to make it's decision on, then the instruction goes through the same steps as BNE however it uses the ALU Less Than output instead of ALU not equal. If necessary, it performs the same modification to the PC.

Why this Implementation: It reused everything from BNE with one additional input to the branch control module and an extra check internally as to whether or not the combination of instructions and values was sufficient to warrant a PC change.

Tests: Create a few scenarios where two registers have different values and run them through the BLT to see if it correctly jumps and then flushes the pipeline.

Challenges: Assigning the ALU opcode to subtraction to make sure that the ALU not equal signal is correct.

EXCEPTIONS

BEX –

Implementation: The BEX instruction is passed in, the exception register ($r30) is read from, it is passed into the ALU with $r0 and the opcode for SUB generated by the alu_opcode_ctrl, it then goes into the branch control block similarly to the other branch instructions. If $r30 wasn't equal to zero, it jumps to the specified location.

Why this Implementation: It reused the same control block as before, the only change that was needed is hardwiring the read register values, then it could effectively be a BNE command.

Tests: Set $r30 to any value other than zero, make sure it branches when BEX is called. Set $r30 to zero and make sure BEX does not branch.

Challenges: Adding in the control signals for the register read values.

SETX –

Implementation: The SETX instruction is passed in, nothing is done in the Decode stage, in the Execute stage, it is passed through a control block that alters the data out value to equal the bottom bits of the instruction and the output is sent to the memory stage before being written back to Register File in $r30.

Why this Implementation: It allowed for easy implementation of exception handling. By checking the values after the execute stage and then making any changes, the operations could finish and then be checked before potentially being overwritten by a SETX if there was an exception.

Tests: Try to use the command and determine if BEX branched as expected. Get exceptions on various arithmetic operations and ensure $r30 is overwritten correctly.

Challenges: Building a robust control block, there were a few edge cases were became difficult to deal.

ERRORS:

Other than the previously mentioned MultDiv error that was replaced, there are no known errors based on the testing I had done.