# ECE 350: Digital Systems Project Checkpoint 4 Technical Report

Natalia Androsz

October 30, 2018

## Duke Community Standard

By submitting this LaTeX document, I affirm that

1. I understand that each `git` commit I create in this repository is a submission
2. I affirm that each submission complies with the Duke Community Standard and the guidelines set forth for this assignment
3. I further acknowledge that any content not included in this commit under the version control system cannot be considered as a part of my submission.
4. Finally, I understand that a submission is considered submitted when it has been pushed to the server.

## 1 Introduction

This technical report described the design of a five-stage, single-issue, 32-bit processor in structural Verilog. The processor operates with a 50MHz clock and is to be deployed on an FPGA. The design integrates previously constructed processor elements – a register file, an ALU, and multdiv. The pipeline "latches" to store data as instructions move from stage to stage. The pipeline handles hazards through bypassing and stalls.

## 2 Subcomponents

### 2.1 Regfile

Regfile is a register file with 2 read ports, 1 write port, and 32 registers which are 32-bits wide. Regfile was implemented using the following submodules:

1. register (32 d flip flops)

2. 32-bit tristate buffer

3. 5:32 decoder

32 registers are generated in the regfile module. Register 0 is generated so that it always outputs 0. Its writeEnable bit is hardwired to ground. Therefore, register 0 can be read, but cannot be written to. Each register's output is passed through two, 32 bit tristate buffers. The tristate buffers are controlled by the decoded read register values.

## 2.2 ALU

The ALU module takes in the required inputs and performs the following operations: addition, subtraction, AND, OR, shift logical left, shift arithmetic right.

The ALU performs every operation in parallel, and decides what operation result to output by decoding the ALU opcode. One bit operation flags are used to enable tristate buffers that determine what value is written to the data_result output. The ALU instantiates the following submodules:

1. opcode decoder

2. module that checks if operands are zero

3. 32-bit AND

4. 32-bit OR

5. 32-bit XOR (one for computing the propogate bits sent to the adder ($A \oplus B$), and one to negate data_operandB when subtracting)

6. 32-bit adder/subtractor

7. barrel left logical shifter

8. barrel right arithemtic shifter

The most important submodules are explained in detail below. Additionally, implementation of overflow logic and output flags is discussed.

### 2.2.1 adder/subtractor

Both addition and subtraction are performed using the same circuitry. For the rest of this report, the module will be referred to as an "adder". The adder is a carry look-ahead adder that contains the following submodules:

1. (4) 8-bit carry look ahead adders

2. (1) 32 bit summation

The propagate and generate logic is brought in from the ALU's 32 bit AND and XOR gates. The addition operation was made more efficient by generating carry signals because the propagate bit xor'ed with the nth bit's carry is the sum of two bits. At the 32 bit adder level, the carries for the 8th, 15th, 23rd, and 31st bit are generated so that the 8 bit carry look ahead submodules can calculate the sums of their bits in parallel instead of having to wait for a carry bit to ripple down the 32 bit line. The adder module is able to perform subtraction when the subtract bit flag is fed in to the 32 bit adder's carry in and the B operand is inverted. The B operand is inverted by XORing data_operandB with the subtract flag. And the subtract flag itself is fed into the adder as an initial carry in.

### 2.2.2 Barrel shifters

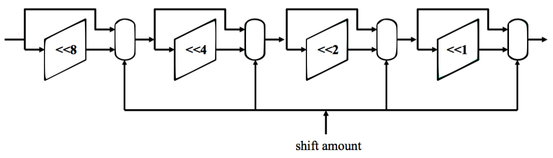


Figure 1: Schematic of left logical barrel shifter

**Left logical shifter:** A barrel shifter that can shift a value 16, 8, 4, 2, and 1 time(s), or any combination of those. When the input is shifted left, a 0 is put in the least significant bit's place, since it is a logical shift.

**Right arithmetic shift:** A barrel shifter that can shift a value 16, 8, 4, 2, and 1 time(s), or any combination of those. When the input is shifted right, a 0 or 1 is put in the most significant bit's place, based on what the most significant bit of the original data_operandA is, since it is an arithmetic shift.

### 2.2.3 ALU Status Flags

**isNotEqual:** Two operands are not equal if their difference is nonzero. If the OR output of every bit in the adder's output is 1, then we know the 32 bit quantity is nonzero. This was done in the isNotZero module.

**isLessThan:** Operand A is strictly less than operand B if their difference produces a negative value. Therefore by checking the most significant bit of the adder module output, one can determine if A¡B. One special case occurs when subtraction produces overflow and the most significant bit becomes 0. This was accounted for by an OR gate containing overflow and the most significant bit of the adder's output.

**overflow:** Overflow can be determined by the xor of the most significant bit's carry in and carry out. This was generated in the 32 bit adder and passed to the ALU.

#### 2.2.4 Challenges Faced when Implementing ALU

When submitted, the ALU was too slow for a handful of testbench cases. The adder was the primary cause of this. The two tests that failed in time trials were the longest latency commands that can occur: subtracting 0 from 0 yielded incomplete results, and isNotEqual was incorrectly set for 0-0. For subtracting 0 minus 0, all of the B operand bits must be converted to 1, XOred with the A input, added together (with a carry in every case). To set the isNotEqual flag, an addition operation must complete and then each of the adder's output bits must be or'ed to determine if the answer is zero. In this test case, 0 is also subtracted from zero.
This bug was solved before using the adder in multdiv. It was solved by adding a submodule that checks whether both of the ALU input operands are zero, and automatically sends zero to the ALU output and sets isNotEqual to 0. This allowed for the bypassing of the slow adder circuitry.

## 2.3 Multdiv

The multdiv module is a signed 32-bit integer multiplier and divider unit. The module handles signed integers in twos complement, and asserts a data_exception on overflow or division by 0. When a multiplication or division operation is complete, the data_resultRDY signal is produced. A multiplication or division operation begins when the ctrl_MULT and ctrl_DIV go high. Since these signals are only high for one clock cycle, they are stored in 1 bit dflipflops to determine whether the division module or multiplication module output should be output from the multdiv unit. The reassertion of a ctrl_MULT or ctrl_DIV, or the assert of data_resultRDY, resets the the multiplication and division modules. More specific details about the implementation of each operation and overflow checking can be found below.

#### 2.3.1 Multiplication (see Figure 2)

Multiplication was implemented using modified Booth's algorithm. It takes 16 clock cycles to complete.
When ctrl_MULT is asserted, the multiplier input is fed to a control module which outputs 3, 16-bit vectors that specify what clock cycles require a shifted multiplicand, a subtraction operation, or nothing to be done. A vector bit is read based on which bit of the 16 bit counter is high.

In parallel to the Booth control module, the input multiplicand and the input multiplicand shifted left by 1 are fed into a mux. The shift flag from the booth control determines which of these two inputs is fed into a mini_ALU. The mini_ALU can perform three operations: add, subtract, or pass an input to the output. The opcode for this ALU is determined by the subtract and nothing bits produced in the control module earlier. It is unnecessary to generate -M and -2M required for modified Booth's beforehand because if subtraction is asserted in the mini_ALU, the generation of -M and -2M will occur internally. This design decision was made with the knowledge that the subtraction operation is guaranteed to occur in less than one clock cycle, and there is no other high latency step in mult.
The output of the mini_ALU is then fed into the top 32 bits of a 64 bit register which shifts any input fed to it right by 2. The result of the multiplication is ready when the 16th bit of the counter is high.

**Multiplication Overflow:** Overflow is checked for in every addition operation in multiplication. The final result of the multiplication is also checked for sign overflow. A particular edge case that was considered was multiplying the largest positive number by itself.

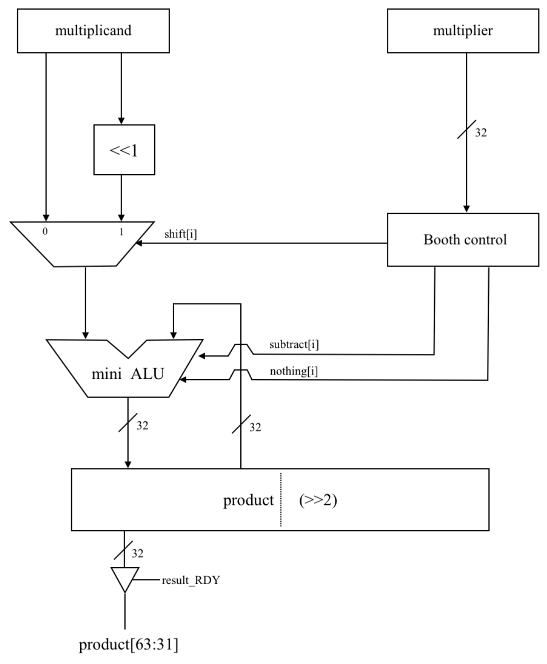


Figure 2: Overview of Multiplication Circuit

### 2.3.2 Division (see Figure 3)

Division was implemented using a closed loop in hardware. A division operation takes 34 cycles. First, the dividend is fed into the lower 32 bits of a 64 bit register that shifts inputs left by 1. On every subsequent cycle, the divisor is subtracted from the top 32 bits of the 64 bit shifting register. If the remainder is greater than or equal to 0, the top 32 bits of the 64 bit register are written and a 1 is written to the least significant bit. Otherwise, the previous remainder is written back to the register.
An important design decision that was made was how to deal with negative numbers. Only positive operands are fed to the add/subtract module. On every assertion of ctrl_DIV, the 2s complement of both operands is found, and used as the divider/dividend if the input is originally negative. If only one of the operands is negative, the output is inverted at the end. Unfortunately, this inverting adds two clock cycles to my division and is very hardware intensive: finding the 2s complement of the input operands and the output requires instantiating three additional adders to add 1 to inverted operands; an addition operation is guaranteed to complete in one clock cycle, therefore an extra clock cycle is required at the beginning and end of the division operation.

**Division Overflow:** Overflow only possible when dividing by zero. Therefore, every bit of the divisor is or-ed with each other to determine if it is nonzero.

### 2.3.3 Challenges Faced when Implementing Multdiv

Multiplication and division were designed as closed loops in hardware. The most challenging aspect of creating this loop was timing. It was difficult to create and coordinate counters that fed correct control bits at the correct cycle.

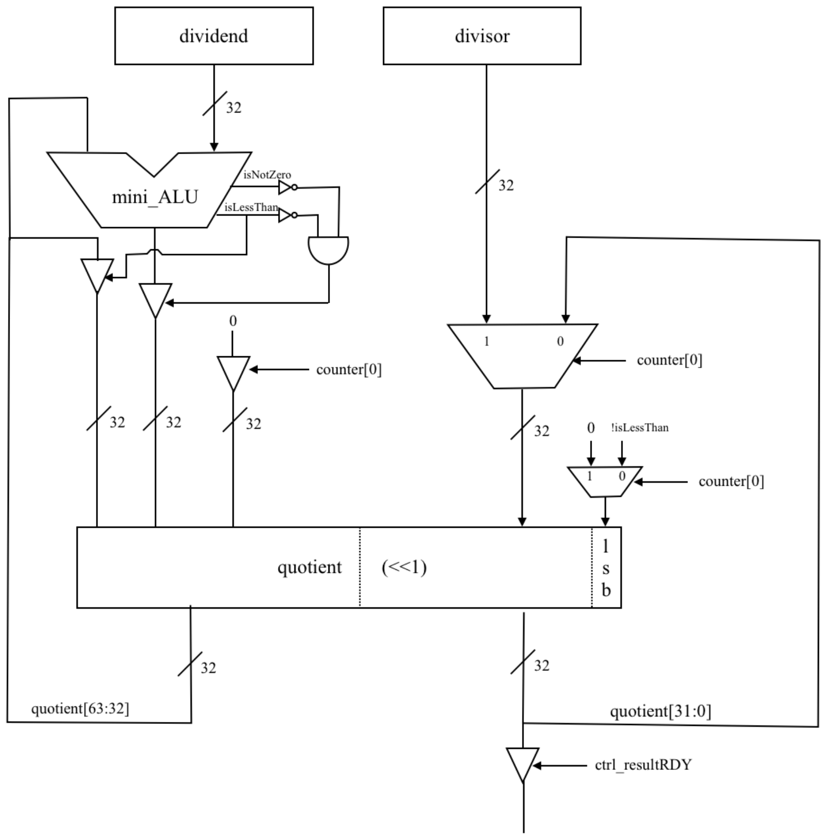


Figure 3: Overview of Division Circuit

# 3 Full Processor

The implementation of a 5 stage pipelined processor closely followed the implementation of the pipelined processor discussed in class. Any significant deviations from the pipeline design shown in lecture are justified in the appropriate stage. An important design feature of the pipeline is that the multdiv module runs outside of the 5 stage pipeline, please see the "Multdiv in the pipeline" subsection.

## 3.1 Memory Elements

### 3.1.1 Register file

Instance of regfile module described earlier. Important register conventions of this pipeline include:

1. $r0 always reads 0
2. $r30 is the status register
3. $r31 is the return address register

Both $r30 and $r31 can be set and overwritten by any instruction. However, precedence is given to writing exception codes to $r30.

### 3.1.2 imem

Single port ROM generated from the Altera IP catalog.
**Inputs:** 12 bit address, clock
**Output:** 32 bit instruction

### 3.1.3 dmem

Single port RAM generated from the Altera IP catalog.
**Inputs:** 12 bit address, clock, write enable control
**Output:** 32 bit instruction

## 3.2 F stage (see Figure 4)

Overview of actions take in this stage:

1. storage and computation of next PC (deciding whether to write PC+1 or a branch's jump address to the PC register)
2. fetch an instruction from ROM memory based on current PC

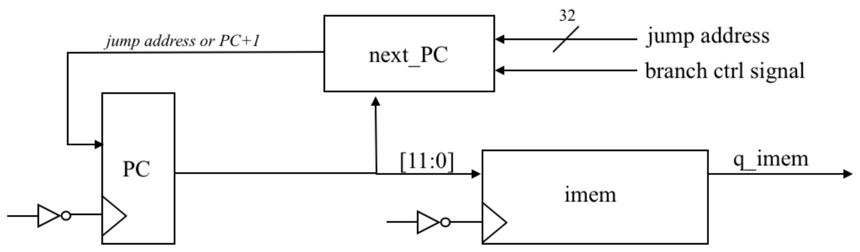


Figure 4: Overview of Fetch Stage

## 3.3 D stage (see Figure 5)

Actions take in this stage:

1. decoding of instruction type
2. retrieving data from register file
3. sign extending immediate
4. inserting nop in case of stall
5. if multiplication, load operands to multdiv unit
6. if stall, nop written to DX register

**Justification of 1:** In future pipeline stages, control logic can be implemented effectively using just the type (R, I, JI, JII) of an instruction. Therefore, the instruction type was decoded and then encoded as a two bit value that was propagated through the pipeline registers. This decreased the latency of certain logic generation.

**Justification of action 3:** The execute stage is the most hardware intense stage. In order to decrease the latency of generating the inputs required for the ALU in the execute stage, the immediate is sign extended in the F stage and that value is saved in the FD register to potentially be used in the execute stage.

**Justification of action 5:** The multdiv module requires that its operands are stable for the entire multi-cycle process. Therefore, the operands must be written to registers before ctrl_MULT and ctrl_DIV are asserted in the execute stage, since multdiv operations begin immediately on their assertion. Therefore, the values of multdiv operands are loaded in the Fetch stage. This design decision requires additional bypassing to be implemented specifically for multdiv instructions.

**Justification of action 6:** Follows the MIPS processor delayed branch model.

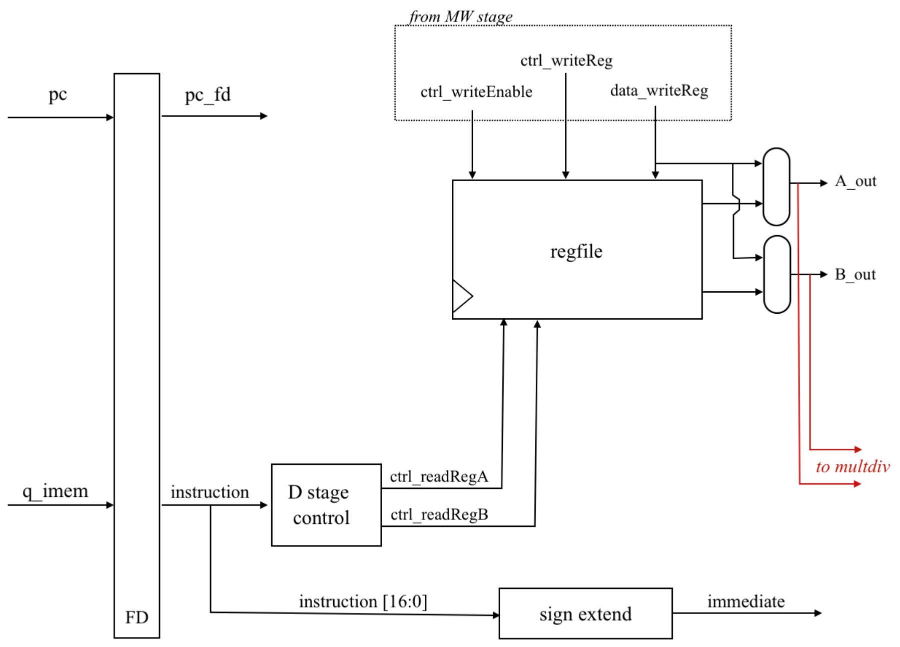


Figure 5: Overview of Decode Stage

### 3.4 X stage (see Figure 6)

Actions take in this stage:

1. computations using ALU
2. branch target address computed
3. branch taken/not taken resolution
4. redirection of values in the case of special instructions

**Justification of 5:** For certain instructions, the value written to a register is not the ALU output, nor the sign extended immediate, nor the value stored in the second register read. Such instructions, such as jal, setx, and any overflow, must have PC+1, the target, or an exception status, written to a register.

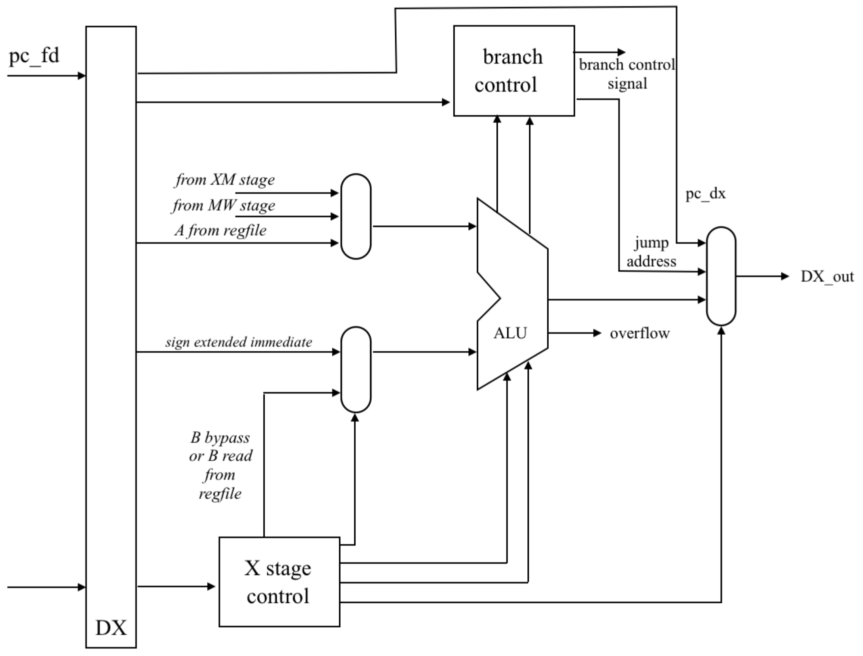


Figure 6: Overview of Execute Stage

### 3.5 M stage (see Figure 7)

Actions take in this stage:

1. reading from RAM memory during load word
2. writing to RAM memory during store word

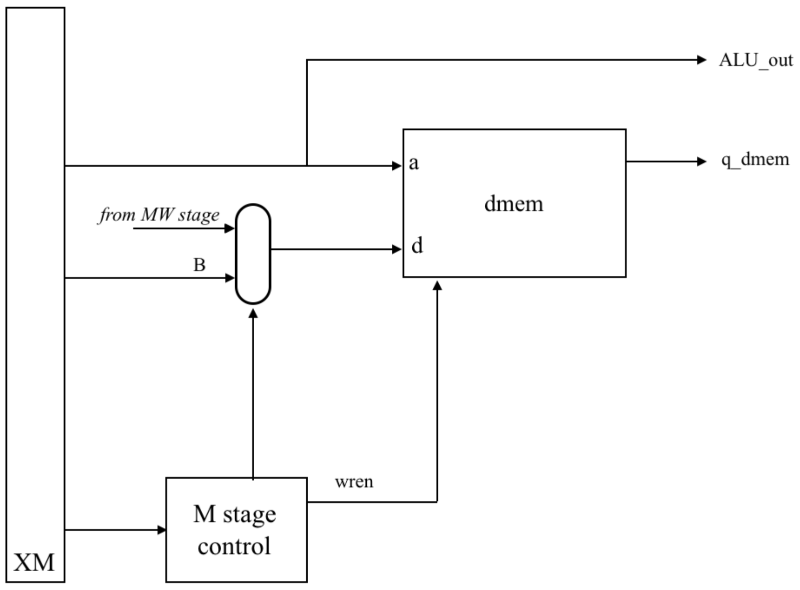


Figure 7: Overview of Memory Stage

## 3.6 W stage (see Figure 8)

Actions take in this stage:

1. selecting value to write back to regfile (if any)
2. selecting register to write (if any)
3. codifying exceptions by writing to $rstatus

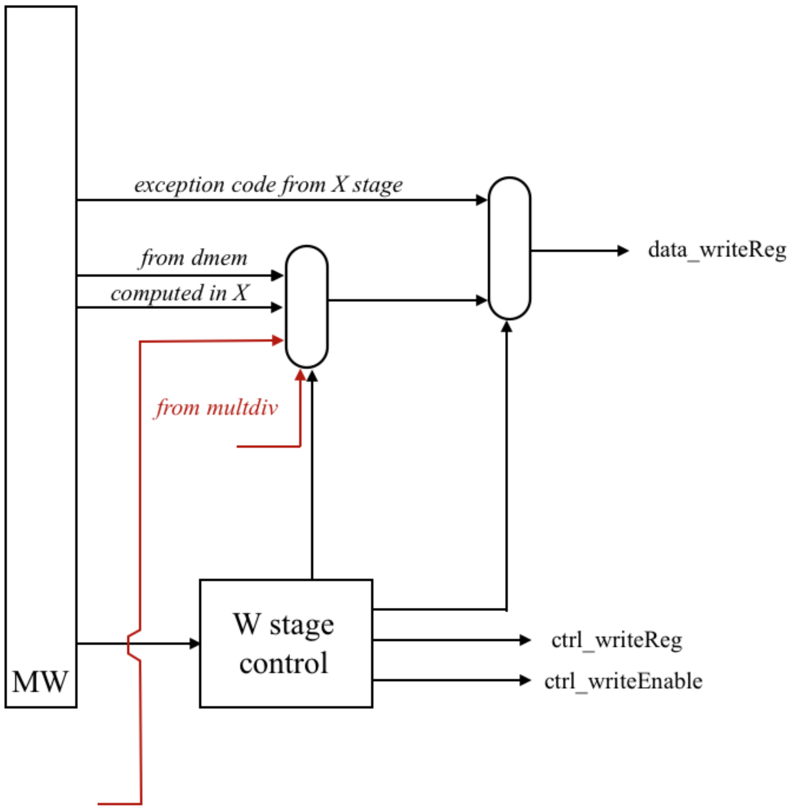


Figure 8: Overview of Write Stage

## 3.7 Multdiv "in" the pipeline

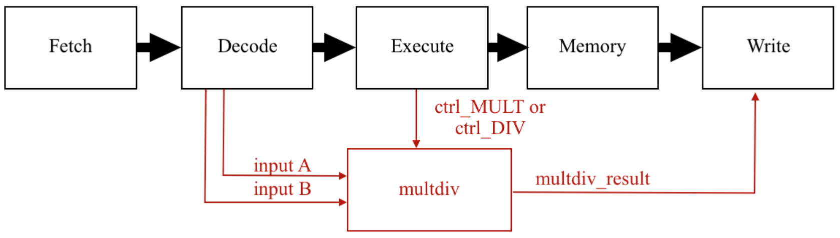


Figure 9: Data flow of multiplication and division operation

Multdiv is a time consuming, multicycle process. Therefore, it has been placed outside of 5 stage pipeline. Its required operands are routed to the module in the Decode stage, and the multdiv result is written back in the Write stage. Therefore, multdiv operation completely bypass the execute and memory stage. Unfortunately, due to this, multdiv operands do not have access to the A and B input bypasses in the execute stage. Due to time constraints, it was not possible to construct the bypasses required for multdiv operands. Therefore **A known bug in the pipeline** is that multdiv will fail if its sources are registers that will be written by instructions in the execute or memory.

## 3.8 Pipeline Control

The avoids data and control hazards through stalls and bypasses.

### 3.8.1 Stalls

The pipeline stalls in two cases:

1. data being read from memory is a source in the following instruction
2. multdiv is in progress
3. branch is taken

**Justification of 2:** Ideally, multdiv would run in parallel to the pipeline and only stall if the source register of a following instruction required the multdiv result as its source. However, due to timing constraints, this was not implemented. Instead, the entire pipeline stalls for the 16 cycle multiplication or 32 cycle division operation.

### 3.8.2 Bypassing

There are 5 bypasses in the processor. Some general logic applied to bypassing is found here:

1. If the destination register of an instruction in the the memory or write stage is the source of an instruction in the execute stage, bypass. EXCEPTION: do not bypass if the destination register of an instruction in the the memory or write stage was actually a source (eg bne, blt).
2. Do not bypass when there is a nop.
3. Bypass when reading a register being written on the same cycle.

**ALU input A & B bypass (see figure 10):** This bypass occurs when the instruction in the M or W stage is writing to a source register of the instruction in the X stage.

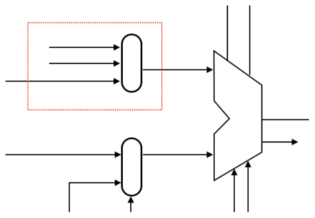


Figure 10: ALU input A bypass in X stage

**dmem data bypass (see figure 11):** This bypass occurs when the instruction in the W stage is writing to the register being written to memory in the M stage during a store word operation.

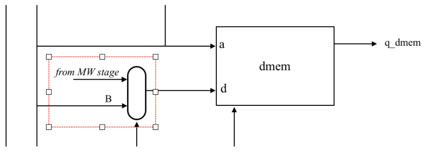


Figure 11: Dmem data bypass in M stage

**regfile read A & B bypass (see figure 12):** Regfile writes on the positive edge of the clock. Therefore, if the instruction in the W stage is writing to a register when an instruction in F stage is reading from that register, the unwritten value will be read.

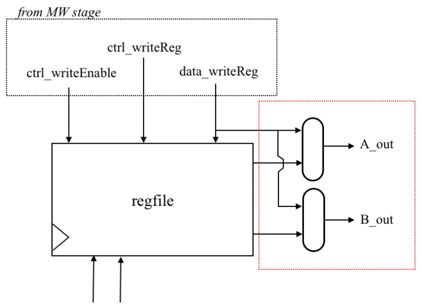


Figure 12: Read register bypass in F stage

#### 3.8.3 Branching

An important set of instructions in the pipeline API involves branching. Branches change the value written to the PC register. There are three different ways a branch instruction may alter the PC:

1. PC = extended 26 bit target
2. PC = PC + 1 + N
3. PC = value in a register

There is a mini_ALU module in the branch control that computes type 2 jump addresses. The execute stage ALU cannot be used because in some cases type 2 branch resolution is determined by the relation of two register values to each other (eg bne, blt) which requires computation in the ALU.
All jump addresses and branch taken/not taken logic takes place in the execute stage. To avoid control hazards, when a branch is taken, the FD and DX pipeline registers are flushed and a nop is inserted into the DX register.

### 3.9 Clocking

The pipeline operates correctly with a 50 MHz clock. The clocking schema of clocked subcomponents in the processor are described below.

1. **PC register:** write on negative edge
2. **imem:** read on negative edge
3. **pipeline registers:** write on positive edge
4. **regfile:** read/write on positive edge
5. **multdiv:** starts operation on positive edge
6. **dmem:** read/write on negative edge

### 3.10 Efficiency details

The efficiency of the processor was considered in two ways: chip area efficiency and timing efficiency. Area was optimized for by trying to minimize the size of modules. For example, a modified Booth multiplier was built instead of a Wallace Tree multiplier. Additionally, a pared down ALU is used in parts of the processor that do not require all the functionality of the full ALU. The trade-off for this optimization is an increased amount of time spent computing multiplication and division operations. Timing was optimized for by adding bypass logic to avoid stalling. This latter point is where more optimization could be added. For example, the construction of a branch predictor would increase the time efficiency of the processor by decreasing the number of times one needs to stall and flush after branches. Regardless, the processor clocks at 50MHz during local testing, thus meeting the specified design requirements.

## 4 Instruction Implementation

The processor ISA contains 18 instructions. Below is a description of how every instruction behaves in the pipeline.

### 4.1 add $rd, $rs, $rt

Operation: $rd = $rs + $rt ($rstatus = 1 if overflow)

1. **Decode:** $rs & $rt read from regfile or taken from write bypass
2. **Execute:** ALU opcode decoded-¿values of $rs & $rt fed to ALU-¿sum is output from X stage; if exception occurs, pass 001 as exception
3. **Memory:** no action
4. **Write:** destination register decoded-¿sum written back to regfile; if exception, select $rstatus as destination register and write exception code

### 4.2 addi $rd, $rs, N

Operation: $rd = $rs + N $rstatus = 2 if overflow

1. **Decode:** $rs read from regfile or taken from write bypass, 17 bit N immediate sign extended to 32 bits
2. **Execute:** ALU opcode set to add operation, sign extended immediate is selected as ALU input B-¿ALU computes sum-¿sum output from DX stage; if exception occurs, pass 002 as exception
3. **Memory:** no action
4. **Write:** destination register decoded-¿sum written back to regfile; if exception, select $rstatus as destination register and write exception code

## 4.3 sub $rd, $rs, $rt

Operation: $rd = $rs - $rt ($rstatus = 3 if overflow)

1. **Decode:** $rs & $rt read from regfile or taken from write bypass
2. **Execute:** ALU opcode decoded-¿values of $rs & $rt fed to ALU-¿difference is output from X stage; if exception occurs, pass 003 as exception
3. **Memory:** no action
4. **Write:** destination register decoded-¿sum written back to regfile; if exception, select $rstatus as destination register and write exception code

## 4.4 and/or $rd, $rs, $rt

Operation: $rd = $rs & $rt / $rd = $rs — $rt $rstatus = 2 if overflow

1. **Decode:** $rs & $rt read from regfile or taken from write bypass
2. **Execute:** ALU opcode decoded-¿values of $rs & $rt fed to ALU-¿ALU output is output of X stage
3. **Memory:** no action
4. **Write:** destination register decoded-¿ALU output written back to regfile

## 4.5 sra/sll $rd, $rs, shamt

Operation: $rd = $rs ¡¡ shamt / $rd = $rs ¿¿¿ shamt

1. **Decode:** $rs read from regfile or taken from write bypass
2. **Execute:** ALU opcode & shift amount decoded from instruction -¿values of $rs & $rt fed to ALU-¿shift output is output of X stage
3. **Memory:** no action
4. **Write:** destination register decoded-¿ALU output written back to regfile

## 4.6 mul/div $rd, $rs, $rt

Please see "Multdiv in the pipeline" for a more detailed description of the implementation of multiplication and division in the pipeline. Operation: $rd = $rs * $rt ($rstatus = 4 if overflow) / $rd = $rs / $rt ($rstatus = 5 if divide by 0)

1. **Decode:** $rs & $rt read from regfile or taken from write bypass
2. **Execute:** ctrl_MULT/ctrl_DIV decoded; $rd decoded and sent to multdiv control module, STALL PIPELINE UNTIL RESULT READY  flush DX register

3. **Memory:** no action

4. **Write:** when result is ready, written back to regfile using saved destination register, pipeline resumes; if exception, select $rstatus as destination register and write exception code

### 4.7 sw $rd, N($rs)

Operation: MEM[$rs + N] = $rd

1. **Decode:** $rs and $rd read from regfile or taken from write bypass ($rd is routed to ctrl_readRegB) & 17 bit N sign extended to 32 bits (saved in DX register)

2. **Execute:** ALU opcode set to add operation & sign extended immediate is selected as ALU input B-¿ALU computes sum-¿sum output from DX stage (ALU output is the address where we want to store something in memory)

3. **Memory:** assert dmem write enabled, use lower 12 bits of ALU output as dmem address, data written to dmem is value stored in $rd

4. **Write:** no action

### 4.8 lw $rd, N($rs)

Operation: $rd = MEM[$rs + N]

1. **Decode:** $rs read from regfile or taken from write bypass & 17 bit N sign extended to 32 bits (saved in DX register)

2. **Execute:** ALU opcode set to add operation & sign extended immediate is selected as ALU input B-¿ALU computes sum-¿sum output from DX stage (ALU output is the address where we want to get something from memory)

3. **Memory:** use lower 12 bits of ALU output as dmem address, save dmem out to MW pipeline register

4. **Write:** destination register decoded-¿memory output written back to regfile

### 4.9 j T

Operation: PC = T $rstatus = 2 if overflow

1. **Decode:** no action

2. **Execute:** target is extended to 32 bits-¿set jump address to extended target-¿assert take_branch (will write this address as new PC)-¿signal to insert nop at DX pipeline register  flush FD and DX register

3. **Memory:** no action
4. **Write:** no action

### 4.10 bne $rd, $rs, N

Operation: if ($rd != $rs) PC = PC + 1 + N

1. **Decode:** $rs and $rd read from regfile or taken from write bypass ($rd is routed to ctrl_readRegB) & 17 bit N sign extended to 32 bits (saved in DX register)
2. **Execute (in branch control):** sign extended immediate added to pc stored in DX register (already PC+1), this is the jump address; write this jump address to PC register ONLY IF isNotEqual from ALU is high-¿IF TAKEN signal to insert nop at DX pipeline register & flush FD and DX register
3. **Execute:** ALU opcode set to subtraction operation-¿ALU computes difference-¿isNotEqual ALU status flag sent to branch control
4. **Memory:** no action
5. **Write:** no action

### 4.11 jal T

Operation: $r31 = PC + 1, PC = T

1. **Decode:** no action
2. **Execute (in branch control):** target is extended to 32 bits-¿set jump address to extended target-¿assert take_branch (will write this address as new PC)
3. **Execute:** X stage output mux selects the pc stored in the DX register on this cycle to be saved in XM register
4. **Memory:** no action
5. **Write:** $r31 set as destination register-¿pc written to $r31

### 4.12 jr $rd

Operation: PC = $rd

1. **Decode:** $rd read from regfile or taken from write bypass ($rd is routed to ctrl_readRegB)
2. **Execute (in branch control):** rd value is set as jump address -¿ take_branch asserted (will write rd value as new PC)

3. **Execute:** no action
4. **Memory:** no action
5. **Write:** no action

## 4.13 blt $rd, $rs, N

Operation: if ($rd ¡ $rs) PC = PC + 1 + N

1. **Decode:** $rs and $rd read from regfile or taken from write bypass ($rd is routed to ctrl_readRegB) & 17 bit N sign extended to 32 bits (saved in DX register)
2. **Execute (in branch control):** sign extended immediate added to pc stored in DX register (already PC+1), this is the jump address; write this jump address to PC register ONLY IF isLessThan (since $rd is input B to ALU) from ALU is high-¿IF TAKEN signal to insert nop at DX pipeline register & flush FD and DX register
3. **Execute:** ALU opcode set to subtraction operation-¿ALU computes difference-¿isLessThan ALU status flag sent to branch control
4. **Memory:** no action
5. **Write:** no action

## 4.14 bex T

Operation: IF ($rstatus != 0) PC = T

1. **Decode:** ctrl_readRegB set to $r30
2. **Execute (in branch control):** target is extended to 32 bits-¿set jump address to extended target value read from $r30 is zero by or-ing every bit in value-¿if nonzero, assert take_branch (will write extended target as new PC)
3. **Memory:** no action
4. **Write:** no action

## 4.15 setx T

Operation: $rstatus = T

1. **Decode:** no action
2. **Execute (in branch control):** target is extended to 32 bits

3. **Execute:** extended target from branch control is selected as the output of the DX stage instead of ALU or PC
4. **Memory:** no action
5. **Write:** destination register is set to $r30 and extended target is written back to regfile

## 5 Process

### 5.1 General Implementation Overview

For previous project checkpoints, I incrementally tested every module I built before putting together the final product. I did not take this approach when building my processor. Instead, I immediately strung together all the processor components, and then went back to debug before implementing bypassing, branching, and multdiv. I found this approach better because I knew how the pipeline was supposed to behave from end to end, so debugging was a matter of narrowing down where an inter-connectivity issue was coming from. I implemented components of my processor in the following order:

1. creating 5 stage pipeline with latches
2. fiddle with clocking schemas for imem, dmem, pc register, and regfile until it worked as expected
3. add in bypassing  stalling based on lecture slides (ALU inputs and destination)
4. realize clocking schema requires another bypass in decode stage
5. implement branching
6. integrate multdiv

### 5.2 Testing Methodology

When the processor was constructed, I began writing simple MIPS programs to test it on. I wrote a program that tested one specific instruction of feature of the pipeline, and then wrote out where the programmer expects data to be and when in comments. If my processor did not behave as expected, I was able to more clearly trace where things were going wrong, because I knew what values were supposed to exist and when.

### 5.3 Learning Points

1. It's 10% coding, 90% debugging.
2. Drawing it out really does help.

3. It's more complicated than the way we did it in 250.
4. It's not going to be perfect the first time around, and iterating on a design is a good thing.

### 5.4 Challenges

1. Time. In two ways. One, getting my designs to work on a 50MHz clock was always the most difficult part. Twice I had to completely rewrite and implementation because it worked fine unlocked, and completely failed when clocked. Second, time in the real world to complete the assignments.
2. Mismatch of lecture material covered and project checkpoint requirements. I found it challenging to start early on project checkpoints when material about the checkpoint hadn't been covered in class. For example, for the ALU checkpoint, I started early and implemented a ripple carry adder, only to learn in lecture two days later that that was not fast enough.

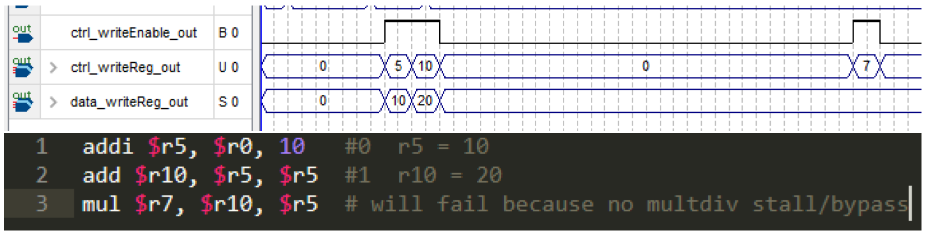
## 6 Conclusion

To conclude, I have constructed every part of a five-stage, single-issue, 32-bit processor in structural Verilog. There are still bugs (see Appendix A) and many inefficiencies with the design, however, it's functional.

## 7 Appendix A: Known Bugs

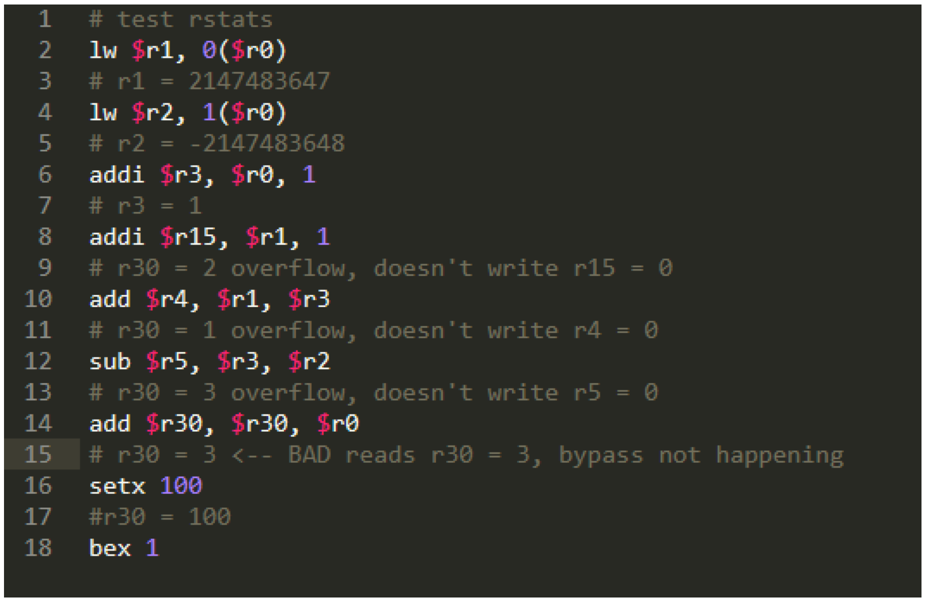
1. If multdiv operands needs to be bypassed from the execute or memory stage, multdiv output will be wrong. This is because there is not stall of bypass logic to account for this situation in multdiv. This has been explained previously in ”Multdiv in the pipeline.” See code example and waveform output in figure .

2. Bypassing an $rstatus exception value doesn’t work. This is because the bypass logic does not account for an $rd that cannot be decoded from an instruction, or isn’t a jal. See code example and waveform output in figure .



```
addi $r5, $r0, 10   #0  r5 = 10
add $r10, $r5, $r5  #1  r10 = 20
mul $r7, $r10, $r5  # will fail because no multdiv stall/bypass
```

Figure 13: Code example and waveform output of known bug 1



```
# test rstats
lw $r1, 0($r0)
# r1 = 2147483647
lw $r2, 1($r0)
# r2 = -2147483648
addi $r3, $r0, 1
# r3 = 1
addi $r15, $r1, 1
# r30 = 2 overflow, doesn't write r15 = 0
add $r4, $r1, $r3
# r30 = 1 overflow, doesn't write r4 = 0
sub $r5, $r3, $r2
# r30 = 3 overflow, doesn't write r5 = 0
add $r30, $r30, $r0
# r30 = 3 <-- BAD reads r30 = 3, bypass not happening
setx 100
#r30 = 100
bex 1
```

Figure 14: Code example of known bug 2