ECE350: Processor Checkpoint
Technical Report
Nathaniel Choe

Introduction

A five-stage pipelined processor was built using mainly structural Verilog. Behavioral Verilog was used for: (1) the D-flip flop module, which acted as the base memory unit; and (2) the multiplier/divider (multdiv) module. The processor supports a total of 18 arithmetic and jump instructions, and is scalable for other custom instructions.

The report will describe the following: (1) an overview of the processor design and layout; (2) individual instruction implementation; (3) efficiency and hardware tradeoffs; and (4) test cases and challenges faced during the processor design. The report will also contain details of individual components, most notably the ALU and the structural multdiv module; it is assumed that the reader cares more about the overall processor layout and efficiency resulting from interactions *between* units, not necessarily interactions *within* units.

Processor Design and Layout: An Overview

Basic Layout

The processor has five main stages: Fetch, Decode, Execute, Memory and Writeback. Each stage is responsible for completing certain tasks. The stages are connected by four latches: Fetch/Decode (F/D), Decode/Execute (D/X), Execute/Memory (X/M) and Memory/Writeback (M/W). Absolutely all latches and memory elements of the processor are clocked on the rising edge, with the exception of the instruction memory module. Each stage and latch will be discussed in more detail below.

Stage: Fetch

The Fetch stage is responsible for obtaining the most current instruction from the instruction memory module. The order of the instruction fetched is dictated by the program counter (PC), which is normally incremented by one every clock cycle. When the PC increments by one, the instruction is fetched on the falling edge of the clock and passed into the F/D latch.

Latch: F/D

The F/D latch is responsible for obtaining (i.e: fetching) the instruction from the Fetch stage and passing the instruction from the Fetch stage into the Decode stage. The F/D latch holds the fetched 32-bit instruction and the associated PC. This latch will output the said instruction and PC, in addition to the necessary control signals and values to the Decode stage. Most importantly, the F/D latch will send 5-bit read ports to the Decode stage.

Stage: Decode

The Decode stage obtains, from the F/D latch, the two 5-bit read ports to be sent to the Register File (regfile). It will then obtain the 32-bit values associated with the read ports from the regfile, and send them to the D/X latch. Depending on the instruction, the values are obtained differently with the assistance of control signals sent from the F/D latch; in general, control signals are also among the sea of values passed from latches to stages. The specifics, including control signal details, will be discussed in greater detail in the Instruction Implementation section of the report. As before, the current instruction and PC are also propagated to the D/X latch.

Latch: D/X

The D/X latch is responsible for obtaining the instruction from the Decode stage and generating control signals to send to the Execute stage. In addition, the latch will pass along the read port values obtained from the Decode stage into the Execute stage, along with other values required by the Execute stage for computation. As always, the current instruction and PC are passed along.

Stage: Execute

The Execute stage is responsible for obtaining the read port values from the Decode stage, and computing values appropriately with the help of the Arithmetic Logic Unit (ALU). The proper values are routed based on control signals sent from the D/X latch. More specifically, the values include the two read port values, shift amount and the ALU opcode (to determine instruction type). For the most part, the result of the computations, along with the current instruction and PC, is sent to the X/M latch.

Latch: X/M

The X/M latch is responsible for obtaining the computed value from the Execute stage and generating control signals to be sent to the Memory stage. Based on the current instruction (passed along from the Execute stage), it will determine whether the computed value is to be used in the Memory stage. This is accomplished with the help of control signals.

Stage: Memory

The Memory stage is responsible for interacting with the dynamic memory (dmem) module. If necessary, the results from the Decode stage (which are passed through the D/X and X/M latches) are stored into the dmem module within this stage; sometimes, the results (directly from the ALU through the X/M latch) are also used as addressed to obtain values from the dmem module. Either the values from the dmem module or the values from the Execute stage are sent to the M/W latch, along with the current instruction and PC.

Latch: M/W

The M/W latch is the final latch responsible for passing along the value from the Memory stage into the Writeback stage. It, like other latches, will also generate necessary control signals to be used in the Writeback stage - most importantly the write enable and several routing controls for the writeback data port and value.

Stage: Writeback

The Writeback stage is responsible for writing the values computed/obtained from previous stages into the regfile. The Writeback stage will obtain the proper 5-bit write port from the M/W latch, and store the necessary values into the regfile at that port if required. As always, the control signals will dictate the specifics.

Individual Components

There are three main components besides the latches: ALU, Register File (regfile) and multiplier/divider (multdiv).

ALU

The ALU consists of a 32-bit 2's complement two-level lookahead adder/subtractor, composed of four 8-bit lookahead adders. Each 8-bit adder uses carry lookaheads to prevent rippling, and the four 8-bit adders also use additional carry lookaheads to prevent rippling between the 8-bit adders. The adder/subtractor can also tell whether the computation overflowed or not. In addition, the ALU consists of 32-bit left logical and right arithmetic shifters, 32-bit AND and OR gates, and a not equal and less than operators. The user can choose which operation to perform by sending in a 5-bit ALU opcode, which is specified in appropriate MIPS arithmetic instructions.

Register File

The Register File consists of 32 32-bit registers. Each 32-bit register consists of 32 D-flip flops, where each D-flip flop is written in behavioral Verilog. The Register File accepts two 5-bit read ports that can select two of the 32 registers to read. A tristate buffer system is used to select the proper register to read, as a 5-bit input multiplexer with 32-bit inputs was deemed less efficient. One write register port and associated 32-bit value is also present.

Multiplier/divider

The structural multiplier/divider (multdiv) module uses a modified booth's algorithm for the multiplier, and a simple naive algorithm for the divider. The multiplier takes 16 cycles, where the divider takes 32. The multiplier calculates overflow by checking if the signs make sense (i.e: multiplying two negative numbers should not result in a negative answer). In addition, an overflow cannot occur if one of the numbers is a 0. The divider calculates overflow by checking if the signs make sense, and also checking if the divider is 0. In addition, -2^32 divided by -1 cannot give 2^32 in 2's complement, although the former two numbers can be represented in 32 bits. This was a special case accounted separately.

Instruction Implementation

The following section will detail how the instructions were implemented in the processor. The instructions are divided into five main categories: (1) ALU-type instructions; (2) memory-type instructions; (3) jump-type instructions; (4) multdiv-type instructions; and (5) miscellaneous-type instructions. It is not so much that the instructions under (5) do not fall under the other four categories - but rather the instructions are sufficiently different to warrant placement into their own “strange” category. The implementations will be discussed categorically. In addition, other miscellaneous implementations will be discussed: (6) overflow cases.

The instructions rely heavily on control signals. Each latch computes the proper control signals to turn on based on the current instruction, as opposed to passing through only the necessary control signals after computing it once in the F/D latch. The table below shows the list of all control signals used. In addition, a table that details what control signals are set by each instruction exists. That existence is confirmed in the Appendix.

Table 1. Control signals and descriptions

Control Signal	**Description**
Rwe	Turn on the write enable of regfile
ALUinB	Use immediate (N) into the ALU, shifted appropriately to 32 bits from 17 bits
RstatusC	Set the write register in regfile to $rstatus ($r30)
JalC	Turns on when Jal is the current command. Set the write port and data to $r31 and PC+1 respectively
DMwe	Enable writing to the memory element
DMld	Enable reading from the memory element
MEMwrite	Choose the write data to be data from the memory element
RdC	Choose Rd for read port 1 as opposed to Rs
PConeNC	Set PC to PC + 1 + N
PCTC	Set PC to T
PCRdC	Set PC to Rd
BNEC	Turn on only when BNE is the current command
BLTC	Turn on only when BLT is the current command
BEXC	Turn on only when BEX is the current command
ALUopAddC	Set the ALUop to "add"
SETXC	Turn on only when SETX is the current command

RsC	Set read port 2 to Rs as opposed to Rt
SwC	Turn on only when SW is the current command

ALU-type instructions

The ALU-type instructions are as follows: **add**, **sub**, **and**, **or**, **sll** and **sra**. These six instructions can solely be done in the ALU. As a result, all the values routed to and from the regfile, into and out of the Execute stage, and through the Memory and Writeback stages are done the same way. The control signals are latched to: (1) route Rs and Rt into the two read ports of the regfile; (2) route the two values associated with the read ports from the regfile directly into the inputs of the ALU; (3) route the result of the ALU through the Memory stage without altering the value; and (4) pass the result into the write data of the regfile, with the write port set to Rd. In terms of the control signals, only *Rwe* is turned on.

Memory-type instructions

The memory-type instructions are as follows: **sw** and **lw**. These two instructions are both I-type instructions, meaning they require a slightly different format from the ALU-type instructions (which are all R-type).

For **lw**, the Rs and Rt read ports are not modified, but the Intermediate value is routed in replacement of the Rt read value during the Execute stage (controlled by *ALUinB*). The two values are summed together with a hard-wired ALU opcode corresponding to add (which is appropriately routed to the ALU opcode by the control signal *ALUopAddC*), and sent to the address port of the dmem module in the Memory stage. The value read from the dmem corresponding to the address is sent to the regfile, with the write port set to Rd, and the regfile enabled (controlled by *Rwe*).

For **sw**, the first read port of the regfile is kept as Rs, but the second read port is set to Rd (controlled by *SwC*). The read value corresponding to Rd is then sent through the latches into the data of the dmem module, since the value at Rd must be written to the address. The address is computed the same way as described for **lw**. The dmem module write enable is turned on (controlled by *DMwe*), and the data value is written into the address. The regfile is not enabled, since **sw** does not write anything to the regfile.

Jump-type instructions

The jump-type instructions are as follows: **j**, **bne**, **jal**, **jr**, **blt**, and **bex**. The jump-type instructions can be further divided into two categories: (1) pure jump instructions, which include **j**, **jal** and **jr**; and (2) conditional jump instructions, which include **bne**, **blt**, and **bex**.

Pure jump instructions are carried out in the Execute stage. At the stage, control signals are checked to see if the current instruction is a pure jump instruction. They act as control signals for a series of multiplexers that choose between adding 1 to the overall PC or adding the necessary value as dictated by the MIPS instructions sheet: **j** and **jal** set the PC to sign extended Target (controlled by *PCTCR*) and **jr** sets the PC to Rd (controlled by *JalC*). Since the overall PC is being updated, the first two latches are flushed of their respective instructions. This is as simple as using a flush signal as the control signal for multiplexers that choose between the Fetch/Decode instructions and nops. The flush signal activates if any of the jump instructions are visited. The jump instruction is allowed to propagate through the Memory and Writeback stages, since some jump instructions (i.e: only **jal**) also require writing to the regfile. For instance, **jal** writes the current PC, added with one, to $r31 (this is controlled by *JalC*, which routes the proper write port and data to the regfile in the Writeback stage).

Conditional jump instructions are also carried out in the Execute stage. They are implemented the exact same way as pure jump instructions, but only carry out if the ALU deems necessary. Since conditional jump instructions rely purely on BNE and BLT results from the ALU (which will be comparing two read values set up the same way as memory-type instructions), the BNE and BLT results are effectively put through AND gates with the control signals corresponding to **bne**, **blt**, and **bex** (the control signals are *BNEC*, *BLTC* and *BEXC* respectively). On a smaller note, since **bex** compares $rstatus with 0 (i.e: $r0), the two read ports to the regfile are set to $rstatus and $r0 respectively in the Decode stage, controlled by *BEXC*.

Multdiv-type instructions

The multdiv-type instructions are as follows: **mult** and **div** (not surprisingly). The two instructions work the same way as ALU-type instructions, but are done separately in a multdiv behavioral module. All the control signals are set up the same way as ALU-type instructions. In the Execute stage, when the Opcode corresponds to either a **mult** or **div** instruction, the control signals telling the multdiv module to start either **mult** or **div** are turned on for one cycle. The F/D and D/X latches are disabled during the multdiv progress, along with the overall PC. The multdiv progress is turned on from the start of the control mult or div signals, until the multdiv asserts a ready signal. Once the multdiv asserts a ready signal, the F/D, D/X and PC are re-enabled, and the instructions continue. The instructions that come after the mult/div instructions freely propagate through Memory and Writeback without causing hazards, since bypassing is implemented fully (which will be discussed later).

Miscellaneous-type instructions

The miscellaneous-type instructions are as follows: **addi** and **setx**. The instruction **addi** is implemented the same way as memory-type instructions; Rs is passed into the first read port of

the regfile, and “Rt” is passed through the second read port, which is overwritten by the sign-extended Intermediate in the Execute stage (“Rt” is technically not Rt, since it is not defined in the MIPS instruction set for addi instructions; it does not matter what is passed in as the second read port). The **setx** instruction writes sign extended Target into $rstatus; in the Execute stage, the *SETXC* control signal routes the sign extended Target in replacement of the ALU output, and the *RstatusC* control routes $rstatus into the write port of the regfile in the Writeback stage.

Overflow cases

Another miscellaneous instruction is handling overflow cases. If an overflow occurs during any ALU operation, the instruction in the X/M latch is overridden so Rd is set to $rstatus, and the value to write is set to the proper $rstatus value as defined by the MIPS instruction sheet. This overriding will also take care of bypass cases where subsequent instructions use $rstatus.

Efficiency and Hardware Trade-offs

This section of the report will detail the measures taken to increase efficiency of the processor, along with a detailed discussion of why certain hardware implementations were made.

Bypassing

Most critically, bypass logic was implemented to avoid data hazards. It is considered more efficient compared to stalling in between instructions, since, if implemented correctly, most instructions can proceed immediately after each other. This is because the Writeback stage occurs two cycles after the Execute stage; without bypassing, instructions that rely on values computed in less than two preceding instructions will not use the correct values. The bypass logic is detailed below in two sections: (1) X/M bypassing; and (2) M/W bypassing.

X/M bypassing

The X/M bypass system routes values settled in the Memory stage (which were computed in the Execute stage one cycle ago) into the Execute stage. The system checks to see if the first or second read port of the Execute stage is the same as the Rd port of the Memory stage. In ECE250-layman's terms, this checks if the data port written to in the Memory stage is ever used in either of the two read ports to the regfile. There are four cases: (1) the above statement is never true - this means there is no hazard, so therefore no bypassing is necessary; (2) the first read port of the Execute stage is the same as the Rd port of the Memory stage - this would result in the address value in the Memory stage (i.e: previous ALU output) to be bypassed into the first argument of the ALU; (3) the second read port of the Execute stage is the same as the Rd port of the Memory stage - this would result in the same address value in the Memory stage to be bypassed into the second argument of the ALU *prior* to the multiplexer choosing between this value as the sign-extended Intermediate value to send into the second argument of the ALU; (4) both the first and second read ports are the same as the Rd port - in this case, (2) and (3) would happen at the same time.

M/W bypassing

The M/W bypass system routes values settled in the Writeback stage (which were computed in the Execute stage two cycles ago) into the Execute stage. Similar to X/M bypassing, there are four cases, and they are handled the same way, except the Rd port of the Writeback stage is compared to the two read ports of the Execute stage.

If X/M and M/W bypasses were to both occur in the same cycle at the same port, X/M bypassing takes precedence, since it is the most recent instruction and thus has the most up-to-date values (updated from the previous X/M bypassing). Even with bypassing, certain instructions require

additional stall logic - for instance, it is impossible to properly bypass **lw**'s Rd value from the Memory stage into the read ports of any instruction of the Execute stage (other than **sw**'s first second port, the Rd value to be written to dmem), since computation would already be completed. As a result, if this were to occur, the F/D latch and PC are stalled (i.e: disabled), and a nop is flushed through the D/X latch (this is equivalent to spacing out the **lw** and any other instruction other than particular types of **sw** while they are in the Execute and Decode stages respectively). In addition, certain instructions do not write to the destination ports in the regfile (i.e: **sw**, **bne**, **bex**, **blt**). If there were any of these instructions, along with any attempting to write to $r0, bypassing was disabled.

Hardware Trade-offs

Computation of control signals

One potentially controversial hardware implementation was having the control signals computer in every latch of the pipeline. One may argue that it is more efficient hardware-wise to compute the control signals once in the Fetch stage and pass through the same control signals throughout the pipes. This, however, was not done for two reasons: (1) the inputs of the latches would be significantly more extensive; and (2) the control signals computer only consisted of two 5-32 decoders and a total of additional 7 AND/OR gates. In addition, having a control signals computer in every stage allows scalability that would most definitely justify the additional hardware cost.

Fast branch

Fast branching was not implemented for several reasons: (1) conditional jump statements require 32-bit comparators, which are hardware-wise much more expensive than decoders, although less expensive compared to an ALU; (2) pure jump statements cannot be implemented separately from conditional jump statements due to potential jump-related hazards, so (1) will always apply. Although one should assume jump statements are taken frequently due to frequent use of loops, the hardware cost does not justify reducing a two-cycle cost of taking jumps into a one-cycle cost. In addition, if there is a two-cycle cost of taking jumps, there would be no issues with faulty bypass logic as long as bypass does not work when the Rd port equals 0. However, these issues may arise when the cost is one cycle.

Behavioral vs. Structural multdiv

In short, the structural multdiv did not work with the processor; it caused instructions preceding the mult/div instructions to not follow through (true reasons not yet known). This was most likely due to hold-time violations, since the structural multdiv was not fully functional. In addition, the hardware cost for the structural multdiv was immense (uses approximately 100

more logical elements). The behavioral multdiv was much more elegant, especially since in overflow divide cases, it asserted the correct ready signals in two cycles.

Test Cases and Challenges

Test Cases

The processor was tested with simple arithmetic instructions, complex instructions with potential hazards and dependencies, overflow cases, overflow cases with dependencies involving $rstatus, jump statements, instructions involving memory elements along with potential hazards (i.e: **lw** dependencies), and instruction sets combining all of these cases together. In addition, cases were created in a way to ensure that an instruction in the Decode stage can appropriately obtain values written by an instruction in the Writeback stage in the same cycle to prevent hazards that should not exist.

For instance, see the following test case:

```
# r1 = 5, r3 = 10, r4 = 10, r5 = 20, r6 = 30, r7 = 20
addi $r1, $r1, 5
add $r3, $r1, $r1 # checks read ports 1&2 XM bypass
add $r4, $r1, $r1 # checks read ports 1&2 MW bypass
add $r5, $r4, $r3 # checks read port 1 XM bypass, port 2 MW bypass
add $r6, $r4, $r5 # checks read port 1 MW bypass, port 2 XM bypass
add $r7, $r3, $r4 # checks no bypass instruction
```

Figure 1. Simple bypass test case

This simple test case checks to see if the X/M and M/W bypass cases work for the regfile read ports. Cases such as this was made for the instructions listed above; a complete list of test cases used is shown in the Appendix. The test cases were checked through functional and timing simulation waveforms.

Challenges

Several challenges arose during the designing of the processor. This was mainly due to not writing test cases beforehand (i.e: not thinking about the potential problems one might encounter during said design).

One of the biggest challenges was implementing bypass logic. In bypassing, X/M bypassing should take precedence over M/W bypassing - yet this was not thought of during the construction of the bypass system. As a result, hours were spent “debugging” after a few simple test cases failed. In addition, there are certain instructions that do not write to the regfile; as a result, these

instructions should not bypass. This includes instructions that attempt to write to $r0. Coming up with this list of instructions was easy, but was not thought of beforehand during the designing of the bypass system.

Another main issue was partly due to Verilog's warning system. When one attempts to assign a 5-bit wire to a 1-bit wire (which shouldn't work), Verilog automatically assigns the least significant bit(s) of the 5-bit wire to the 1-bit wire. This caused "warnings" but not "errors." Unless the user manually compiled and checked the flags thrown while compiling a test bench, the program was allowed to continue. As a result, these mistakes in the code were not picked up immediately, but only after hours of debugging. Of course, this is the fault of the person who wrote the code. But Verilog should be more transparent.

It was fairly difficult to get the multdiv module working with the rest of the processor. More specifically, the control signals to start the multdiv computation were hard to assert only for one cycle - two D-flip flops had to be used. In addition, it was difficult to reset the memory elements when the ready signal was asserted from the multdiv module. The stall logic that worked well with the multdiv module also proved to be difficult to implement, but with careful testing and Logisim was fine. In addition, at least 3 hours were spent attempting to "debug" faulty overflow logic involving the divider, when in reality the divider's control ready signal was not working as intended.

In conclusion, although there were challenges, the overall implementation of the processor felt streamlined. The processor was designed in Logisim prior to being written in Verilog (images in Appendix), which really helped the process(or).

Appendix

Test Cases

full.s

```
## Full Testing
simple_arithmetic:
     # r1 = 7, r2 = 17, r3 = 22, r4 = 44, r5 = 66, r6 = -49, r7
= 17
     addi $r1, $r1, 7
     addi $r2, $r1, 10
     addi $r3, $r1, 15
     add $r4, $r3, $r3
     add $r5, $r4, $r3
     sub $r6, $r2, $r5
     add $r7, $r6, $r5
     blt $r6, $r7, simple_multdiv
     j simple_halt

simple_multdiv:
     # r8 = 49, r9 = 2401, r10 = 49, r11 = 0, r12 = -343, r13 =
-15, r14 = 5145
     mul $r8, $r1, $r1
     mul $r9, $r8, $r8
     div $r10, $r9, $r8
     div $r11, $r2, $r4
     mul $r12, $r6, $r1
     div $r13, $r12, $r3
     mul $r14, $r13, $r12
     bne $r8, $r10, simple_halt
     bne $r14, $r13, 0
     j simple_overflow

simple_overflow:
     # r15 = 131070, r17 = 4, r18 = 5, r19 = 011...1111000..000,
r20 = 0111...11, r22 = 1, r24 = 2, r25 = 4, r26 = 6
     # mult overflow testing
     addi $r15, $r15, 65535 # $r1 = 0000...001111...111
     addi $r15, $r15, 65535 #r1 = 131070
```

```
     mul $r16, $r15, $r15 # mult overflow
     addi $r17, $rstatus, $r0 # $r3 = 4 = $rstatus

     # div overflow testing
     div $r7, $r7, $r0 # div overflow
     add $r18, $rstatus, $r0 # $r5 = 5 = $rstatus

     # add overflow testing
     addi $r15, $r0, 65535
     sll $r19, $r15, 15 # $r8 = 0111....111000... 000
     addi $r20, $r19, 32767 # $r9 = 0111...1111
     add $r21, $r20, $r20 # add overflow
     add $r22, $rstatus, $r0 # $r11 = 1 = $rstatus
     addi $r23, $r20, 1 # addi overflow
     add $r24, $rstatus, $r0 # $r13 = 2 = $rstatus
     mul $r25, $rstatus, $rstatus
     add $r26, $r25, $rstatus
     addi $r29, $r29, 2047
     j simple_dmem

simple_dmem:
     sw $r24, 0($r17)
     lw $r27, 0($r25)
     bne $r24, $r27, simple_dmem
     j simple_halt
     j simple_dmem

simple_halt:
     bne $r0, $r8, -1
     addi $r8, $r8, 1
     bne $r0, $r8, -1
```

jump.s

```
# r1 = 1, r2 = 4, r3 = 5, rstatus = 13, r4 = 69, r5 = 70
addi $r1, $r1, 1
addi $r2, $r1, 3
jal to_jump
j halt
```

```
to_jump:
     addi $r3, $r2, $r1
     bex halt
     setx 13
     bex to_start

to_start:
     addi $r5, $r5, 70
     jr $r31

halt:
     addi $r4, $r4, 69
     bne $r0, $r1, -1
```

overflow.s

```
# overflow testing

# mult overflow testing
addi $r1, $r1, 65535 # $r1 = 0000...001111...111
addi $r1, $r1, 65535 #r1 = 131070
mul $r2, $r1, $r1 # mult overflow
addi $r3, $rstatus, $r0 # $r3 = 4 = $rstatus

# div overflow testing
div $r4, $r1, $r1 # $r4 = 1
div $r4, $r1, $r0 # div overflow
add $r5, $rstatus, $r0 # $r5 = 5 = $rstatus
add $r6, $r5, $r5 # $r6 = 10

# add overflow testing
addi $r7, $r7, 65535 # $r7 = 000000..01111...1111
sll $r8, $r7, 15 # $r8 = 0111....111000... 000
addi $r9, $r8, 32767 # $r9 = 0111...1111
add $r10, $r9, $r9 # add overflow
add $r11, $rstatus, $r0 # $r11 = 1 = $rstatus
addi $r12, $r9, 1 # addi overflow
add $r13, $rstatus, $r0 # $r13 = 2 = $rstatus
```

misc.s

```
# miscellaneous tests
# r1 = 100, r3 = 0, r4 = 0, r5 = 100, r6 = 200, r7 = 200

# adding to $r0
addi $r1, $r1, 100
add $r0, $r1, $r2
add $r3, $r0, $r0
add $r4, $r0, $r0

# lw stall
sw $r1, 0($r1)
lw $r5, 0($r1)
add $r6, $r5, $r1
add $r7, $r6, $r0

# memory bypass
addi $r1, $r1, 10
sw $r1, 0($r0)
lw $r2, 0($r0)
```

Control Signals

This table illustrates what control signals are activated per instruction. A description of each control signal is shown above in the report.

	Rwe	ALUinB	RstatusC	JalC	DMwe	DMld	MEMwrite	Rdc	PConeNC	PCTC	PCRdC	BNEC	BLTC	BEXC	RsC	ALUopAddC	SwC	SETXC
add	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
addi	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
sub	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
and	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
or	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sll	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sra	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mul	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
div	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

sw	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0
lw	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0
j	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
bne	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0
jal	1	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
jr	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0
blt	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0
bex	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
setx	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
custom_r	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Model of Entire Processor

This link is an exact Logisim replica of the Verilog-implementation of the processor described in this report:
https://imgur.com/a/jEvv1oS