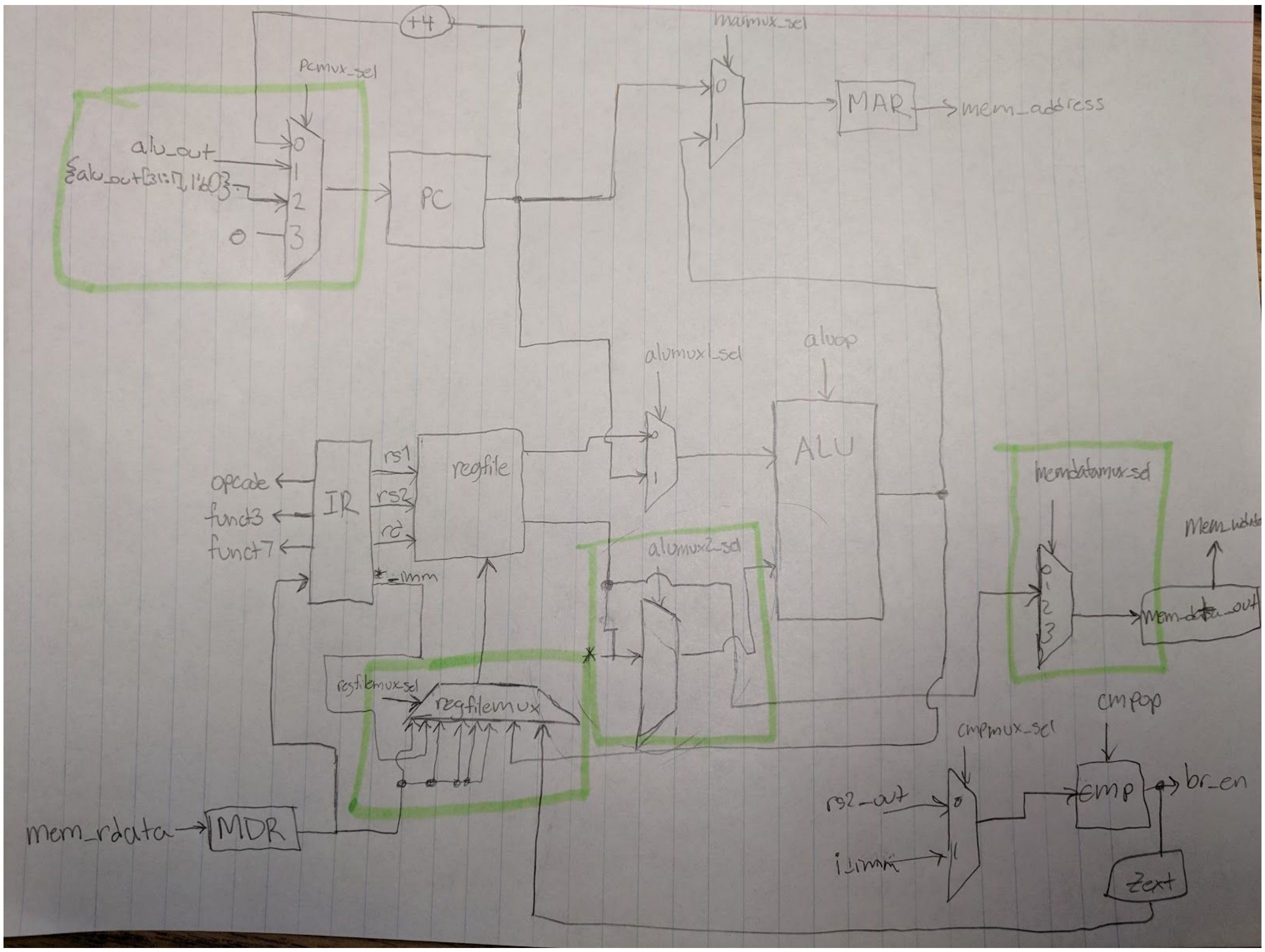
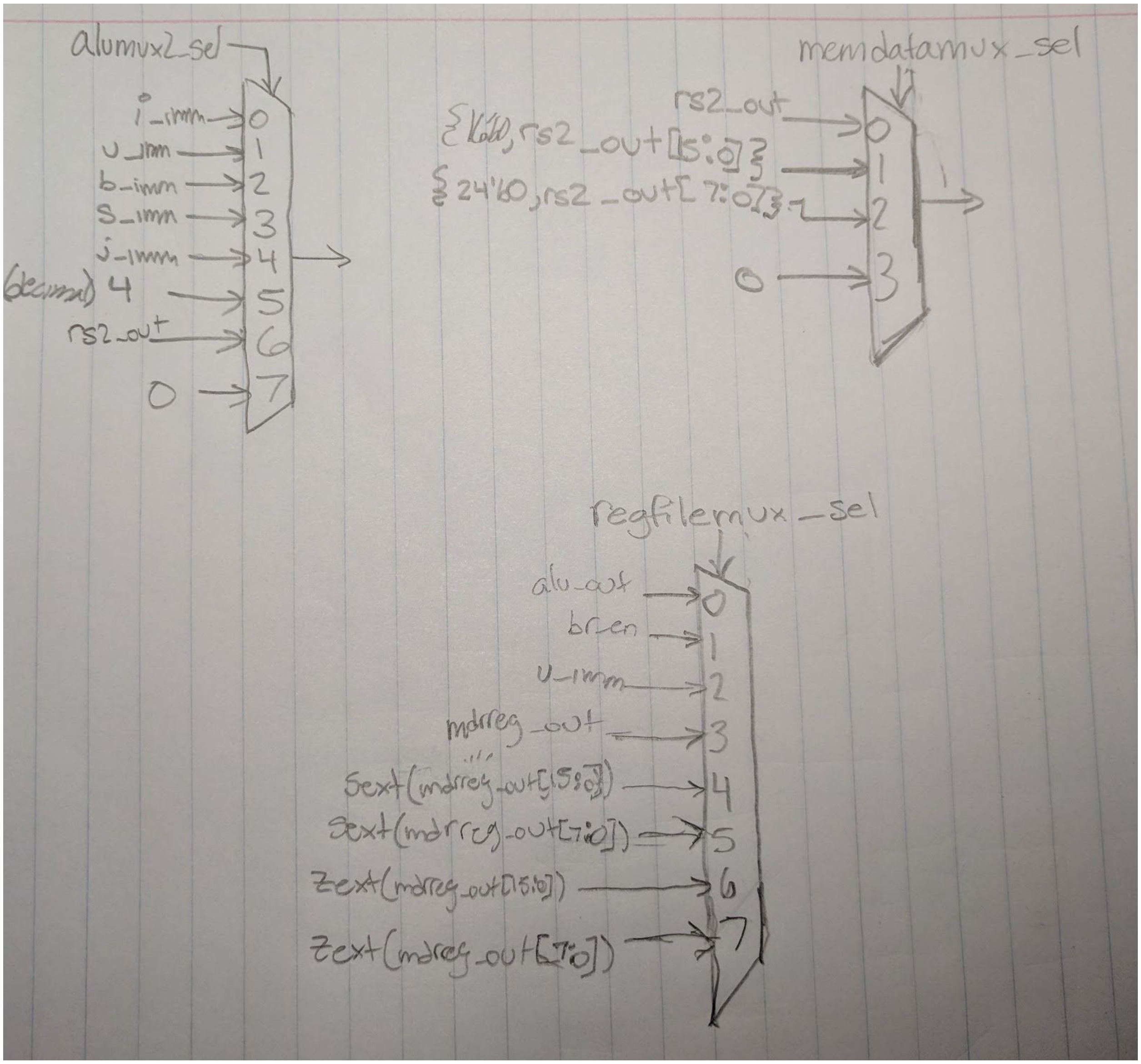
Matthew Grossfeld
ECE 411
Professor Kumar
9/16/18

MP1 Handin

Datapath Diagram:





Description of Highlighted Blocks That Were Added:

- Mem_data_mux
  - Mem_data_mux is a 4:1 mux that selects the appropriate data from rs2_out. The output is then sent to mem_data_out, where the data is later written into memory. The mux is used when instructions SW, SH, and SB are used.

Testing Design:

In order to test the design, I broke up the instructions that needed to be implemented into different parts. First, I implemented the jump instructions (JAL and JALR). Then, I implemented all of the arithmetic register-register operations (including instructions that require a bit30 check). Then after that I added support for the store and load operations. In order to test this, I used my own code that I have put below this section.

To implement JAL and JALR, I knew that I would need to create new states in the control logic, and I would also need to set the least significant bit of the address to 0 for JALR. To do this, I knew that I'd need to modify the PC mux because it would output the address. Once the control logic was made, I knew that it would simply need to perform unconditional jumps to specified labels. So, I simply made a label called "jal_loop" and put a JAL instruction under it that jumps back to the label, creating an infinite loop. For JALR, I did a similar test. I created a label and then loaded register X6 with the value of the address. After that, I use JALR to jump back to the label. If JALR didn't work, it would load a value into X6, indicating that JALR has failed. It should be noted that I never tested JAL and JALR at the same time because they create infinite loops in my test code. So when I was testing, I would comment one of those tests out (i.e. I would comment the JAL test when I wanted to see if JALR worked).

For the arithmetic operations, I knew I needed to add a new state in the control logic called s_reg and then create a lot of case statements within that state. To test my design, I simply loaded arbitrary values into registers X1, X2, and X3, and then performed all of the required operations that we had to implement. I stored all of the results of these operations into various registers so that I can easily check to see if the operation was performed correctly. In order to properly perform register-register operations, I had to add rs2_out as an input to the alumux2.

For the load and store operations, I simply tested this by loading arbitrary values of different sizes (words, bytes, halfs) into different registers. I tested every single load instruction with both positive and negative values to ensure that sign extension and zero extension was performed properly for the appropriate instructions. To test store, I loaded arbitrary values into registers again. Then, I would store those values into memory addresses labeled at the bottom of my test code. Then, I loaded the values from those address labels into different registers and compared the values. This way proved to be the simplest method of testing stores. Another way I could have tested it would have been by seeing what mem_address was and what was being written into the address, but it became very difficult to follow wave forms doing that.

Test Code:

```
riscv_mp0test.s:
.align 4
.section .text
.globl _start
        # Refer to the RISC-V ISA Spec for the functionality of
        # the instructions in this test program.
_start:
        # Note that the comments in this file should not be taken as
        # an example of good commenting style!!  They are merely provided
        # in an effort to help you understand the assembly style.


#Making sure all load instructions work
    lw x1, LVAL1     # x1 = 0x00ECE411 GOOD    POSITIVE WORD
    lw x2, LVAL2     # x2 = 0xECE41100 GOOD  POSITIVE WORD
    lb x3, LVAL4     # x3 = 0x0000001A GOOD  POSITIVE BYTE
```

```
    lb x4, LVAL5     # x4 = 0xFFFFFFFF GOOD   NEGATIVE BYTE
    lbu x5, LVAL4    # x5 = 0x0000001A GOOD  UNSIGNED POSITIVE BYTE
    lbu x6, LVAL5    # x6 = 0x000000FF GOOD  UNSIGNED NEGATIVE BYTE
    lh x7, LVAL9     # x7 = 0xFFFFABCD GOOD   SIGNED NEGATIVE HALF
    lh x8, LVAL10    # x8 = 0x00001ABC GOOD  SIGNED POSITIVE HALF
    lhu x9, LVAL9    # x9 = 0x0000ABCD GOOD  UNSIGNED NEGATIVE HALF
    lhu x10, LVAL10  # x10 = 0x00001ABC GOOD UNSIGNED POSITIVE HALF
    lw x11, LVAL11   # x11 = 1
    lw x12, LVAL12   # x12 = 0

#Making sure all store instructions work
    sw x1, SVAL1, x12    #SVAL1 should hold 0x00ECE411 GOOD
    sw x2, SVAL2, x12    #SVAL2 should hold 0xECE41100 GOOD
    sb x3, SVAL3, x12    #SVAL3 should hold 0x0000001A GOOD
    sb x6, SVAL4, x12    #SVAL4 should hold 0x000000FF GOOD
    sh x8, SVAL5, x12    #SVAL5 should hold 0x00001ABC GOOD
    sh x9, SVAL6, x12    #SVAL6 should hold 0x0000ABCD GOOD
    sb x9, SVAL7, x12    #SVAL7 should hold 0x000000CD BAD
    sh x2, SVAL8, x12    #SVAL8 should hold 0x00001100 BAD

    lw x11, SVAL1    # x11 should now be 0x00ECE411
    lw x12, SVAL2    # x12 = 0xECE41100
    lw x13, SVAL3    # x13 = 0x0000001A
    lw x14, SVAL4    # x14 = 0x000000FF
    lw x15, SVAL5    # x15 = 0x00001ABC
    lw x16, SVAL6    # x16 = 0x0000ABCD
    lw x17, SVAL7    # x17 = 0x000000CD
    lw x18, SVAL8    # x18 = 0x00001100

# Arithmetic operations
    lw x1, ZERO    # x1 = 0
    lw x2, ONE     # x2 = 1
    lw x3, FIVE    # x3 = 5

#Loop until x1 = 5
add_loop:
    add x1, x1, x2        # x1 = x1 + x2
    bne x1, x3, add_loop

    add x4, x1, x0        # x4 = x1 = 5

#Loop until x4 = 0
sub_loop:
    sub x4, x4, x2        #x4 = x4 - x2
    bne x4, x0, sub_loop




#Testing shift functions
    lw x1, ONE    # x1 = 1
    lw x2, GO_RIGHT    # x2 = x80000000
    lw x3, FOUR    # x3 = 4

    sll x2, x4, x2
    srl x1, x4, x1
    sra x1, x4, x1
```

```
#Testing Logic
    lw x1, LOGIC1
    lw x2, LOGIC2

    xor x3, x1, x2
    and x4, x1, x2
    or x5, x1, x2

#Testing comparisons
    lw x1, COMP1
    lw x2, COMP2
    lw x3, COMP3

    slt x4, x1, x2
    slt x5, x2, x3
    sltu x6, x1, x2
    sltu x7, x2, x3


there:
       lw   x6, jalr_test    #X6 <= GOOD = 0x600d600d
       jalr x0, x7, 0        # Loop to THERE
       lw   x6, fail         # If still here, X6 <= 0xd


jal_loop:
       jal  x0, jal_loop

.section .rodata

LVAL1: .word 0x00ece411
LVAL2: .word 0xece41100
LVAL3: .word 0x01234567
LVAL4: .word 0x0000001A
LVAL5: .word 0x000000FF
LVAL6: .word 0x00000078
LVAL7: .word 0x00000012
LVAL8: .word 0x000000F2
LVAL9: .word 0x0000ABCD
LVAL10:      .word 0x00001ABC
LVAL11:      .word 0x00000001
LVAL12:      .word 0x00000000

ZERO:     .word 0x00000000    # For initializing
ONE:      .word 0x00000001    # For shifting left and incrementing/decrementing.
FIVE:     .word 0x00000005    # For loops
FOUR:     .word 0x00000004    # For shifting loops
GO_RIGHT: .word 0x80000000 # For shifting right
LOGIC1:   .word 0x0F0F0F0F    #For AND, OR, XOR
LOGIC2:   .word 0x000FF0FF    #For AND, OR, XOR
COMP1:    .word 0x000000009    #For comparisons
COMP2:    .word 0x000000008    #For comparisons
COMP3:    .word 0x0000000F0    #For comparisons

jalr_test: .word 0x600d600d
Fail:      .word 0xFEDCBA98
```

```
SVAL1: .word    0x00000000
SVAL2: .word    0x00000000
SVAL3: .word    0x00000000
SVAL4: .word    0x00000000
SVAL5: .word    0x00000000
SVAL6: .word    0x00000000
SVAL7: .word    0x00000000
SVAL8: .word    0x00000000
```