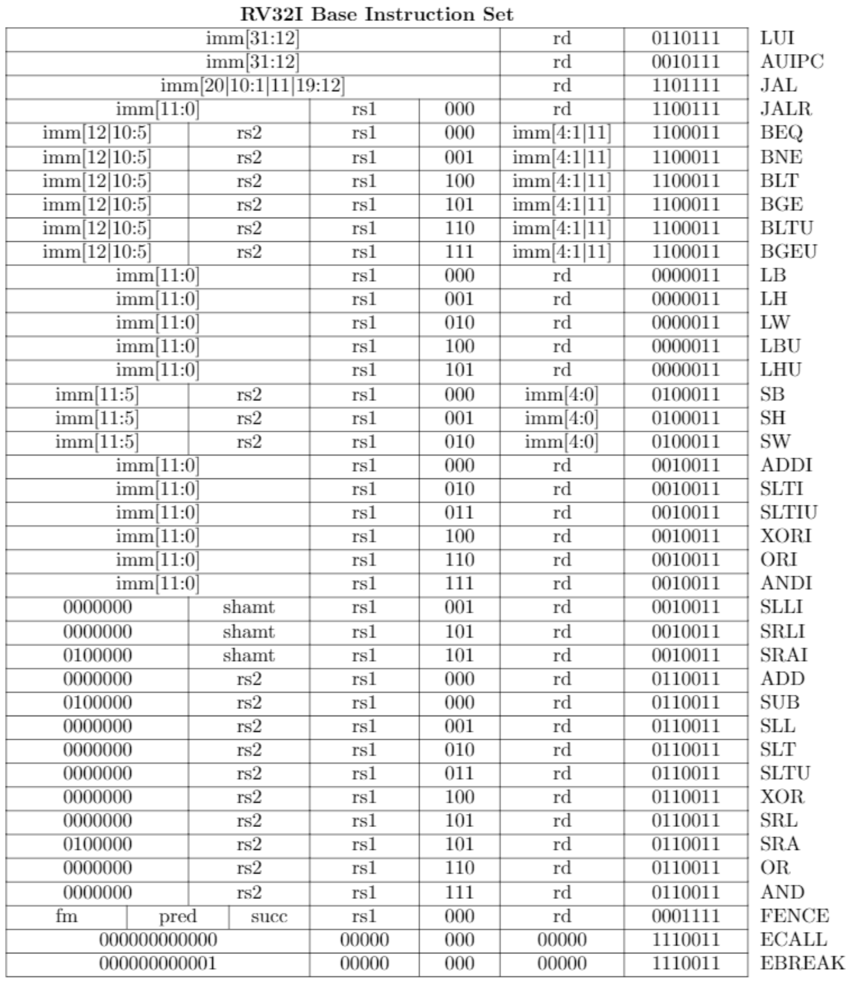
EECS 151/251A Homework 8

Due Monday, April 8th, 2024

Introduction

This homework is meant to test your understanding of the RISC-V architecture as well as the design of memory blocks. There are 8 questions. Please check Ed first if you have any questions.

Below is a list of RISC-V instructions that you can use for problems 1 and 2.



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm pred	succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 1: RISV Instruction set

Problem 1: Enabling Instructions

Ignore the FENCE, ECALL, and EBREAK instructions. For the other instructions, which instructions would not be possible to execute on the datapath presented in class for the single-cycle processor we showed in class (Lecture 15 slide 15). For each instruction that cannot be run, please explain what signals/processing units are missing, and how you can modify the datapath to enable them. To clarify any instructions you are unfamiliar with, you can use the RISC-V Instruction Manual (https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf). You can assume the memory is byte-addressable.

Solution:

You are unable to do the LB, LH, LBU, or LHU instructions as you must have a signal to only grab a piece of a full word and then extend it. In order to fix this, you need to append a data splicer and sign extender after the memory unit that additional takes in an **UN** signal in order to zero-extend for LBU and LHU.

The current datapath cannot currently do a LUI or AUIPC instruction either as the ImmGen sign extends from the left. In these cases, we just need an additional signal to zero extend from the right (or left shift by 12 bits).

Problem 2: Encoding Instructions

Let's assume you wanted to allow a more compact encoding of each instruction (and didn't care about all instructions being the same size, 32-bits), but needed to retain the same size immediates for each instruction that uses one. List each instruction that can be minimized from 32-bits, and explain how the instruction is encoded.

Solution:

1. R type ALU instructions. Let us set the OpCode to be 1-bit (0) and so then the total instruction is 1-bit opcode + 3-bit ALU op + 3*5 register addresses = 19 bits.
2. I type ALU instructions. Opcode is still 1-bit, so that saves us 6 bits, giving us 26 bits. The Shift instructions don't use the full immediate space, so they will use 20 bits.
3. Let Stores and Loads use the Opcode 10 (note that cannot do 01, since opcode of 0 alone is used for ALU instructions). This saves us 5 bits, giving us 27 bits.
4. Let branches have an opcode of 110 (again can't use 0 or 10). The type of branch is still indicated by the 3-digit ALU op, so we can save 4 bits, giving us 28 bits.
5. J, JAL, JALR, LUI, and AUIPIC have the opcodes ranging from 111000 - 111111. This saves us 1 bit giving us 31 bits. For JALR, we no longer need the ALU op, so that becomes 29 bits.

Problem 3: Implementing Multiply Add

How could you add an FMA (fused multiply add) instruction to the ISA. And FMA takes 3 registers, *r1, r2, and rd* and does $rd = r1 * r2 + rd$. Please explain the instruction encoding for this operation, as well as what signals and units (if any) you would add to the single-cycle datapath. Give 2 potential solutions to this problem and their trade-offs relative to each other.

Solution:

There are several solutions.

1. Add a multiply unit that takes in r1 and r2 and write back to a temp register. Then on the next cycle add the temp register to rd and write back to rd.
2. Add a multiply unit and an accumulator. First, write r1 * r2 to the accumulator and on the next cycle add rd and write back to rd. Need to have a signal on the accumulator unit that specifies whether you accumulate or overwrite the current contents of the accumulator.
3. Add a 3rd read port to the register with your multiply unit. You can grab r1, r2, and rd and perform the entire operation in one cycle.
4. Break up the multiply into several additions and shifts. By adding just an accumulator unit, you can perform both the multiplication and the last addition.

In terms of tradeoffs, options 1 and 2 are very similar. Option 1 requires no accumulator unit, but does require you to keep a register slot available just for temporary use. Option 3 requires by far the most complex design. In addition, this may lengthen your critical path, and make all of register reads slower, but the latency of this instruction could be the fastest. Option 4 requires only a single accumulator unit (which you can also have as a temporary register if you did not want to change anything about the data path). It has the longest latency for this instruction, but does not affect your critical path or clock frequency.

Problem 4: Improving CPI

Assume you have the 3 stage pipeline (I, X, M). You can assume that data is available the cycle after M. Additionally you can assume that a load spends 3 cycles in the M stage, and a store takes 2 cycles. Assume that the memory block has 2 read ports and 1 write port, but is not pipelined. This means that at most 2 loads and 1 store can be in the M stage at once. Compute the total number of cycles that are needed to complete the program in each of the given scenarios for the following code block. 1. Assume that memory accesses do not go to the same location unless it is obvious in the code (you can explicitly calculate the memory address or it is the same offset to the same register value). The M stage, as we discussed in class, does Memory and Writeback. Instructions must be written back **in order** and only one value can be written back at once. The last cycle of M for both loads and stores finishes the memory access and does write back. Therefore, you cannot use the respective port until the instruction is over.

```
addi x5, x0, 2
loop:
    sll x4, x4, x5
    lw x1, 0(x4)
    lw x2, 4(x4)
    lw x6, 12(x4)
    add x4, x1, x2
    add x6, x4, x6
    sw x6, 12(x4)
    sw x4, 8(x4)
    subi x5, x5, 1
    beq x5, x0, done
    j loop
done:
    add x4,x4,x6
```

1. Scenario 1: You have implemented no forwarding. If data is unavailable, the only option to deal with the hazard is to stall. Additionally, you always predict Not Taken on the branch. If a branch is taken, the pipeline must be flushed as soon as it is resolved and the correct instruction is loaded on the next cycle.

2. Scenario 2: You implement data forwarding. This means that data is available the cycle after it is executed.

3. Scenario 3: You keep the forwarding from part 2 and additionally add a perfect branch predictor, which will never take the wrong branch.

4. Scenario 4: You have the forwarding and a perfect branch predictor, and implement a pipelined memory access. This means that loads do not have to stall if both ports are in use. Additionally, stores also do not have to wait for each other. Note that loads that come after a store to the same address must still wait (data hazard).

5. Scenario 5: You decide that the pipelined memory access is not worth it. Instead, you implement a cache that holds the most recent 2 addresses accessed (via loads and stores). If the address you try to load or store is in the cache, then the operation can be done in a single cycle. At the beginning of the code block, the cache is empty.

Solution:

Cycle	1	2	3	4	5	6	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
addi	I	X	M																										
sll		I	X	M																									
lw			I	I	I	X	M	M	M																				
lw						I	X	M	M	M																			
lw							I	X	X	M	M	M																	
add								I	I	I	I	X	M																
add												I	I	I	X	M													
sw															I	I	I	X	M	M									
sw																		I	X	X	M	M							
subi																			I	X	X	X	M						
beq																				I	I	I	I	I	X	M			
j																									I	X	M		
sll																											I	X	M

1. We show the table for the first loop. The rest of the program runs for another 22 cycles to get to the X stage of the beq. On this beq, the j instruction is flushed, costing 1 cycle, and the add is started on cycle 53 and finishes on cycle 55.

Cycle	1	2	3	4	5	6	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
addi	I	X	M																			
sll		I	X	M																		
lw			I	X	M	M	M															
lw				I	X	M	M	M														
lw					I	X	X	M	M	M												
add						I	I	I	X	X	M											
add									I	I	X	M										
sw											I	X	M	M								
sw												I	X	X	M	M						
subi													I	I	X	X	M	M				
beq															I	I	X	X	M			
j																	I	I	X	M		
sll																				I	X	M

2. Data forwarding saves 7 cycles per loop, which brings us to 41 cycles.

3. We flush one time. This saves us 1 cycle bringing our total instructions to 40.

4. This saves us 1 cycle on the 3rd load and 1 cycle of the 2nd store, which saves us 2 cycles per loop, for a total of 4 cycles. This brings our total cycles down to 36.

5. Unfortunately, we have no temporal locality in this program, as addresses are not accessed more than once. A more effective cache would have large cache lines to take advantage of spatial locality. So the total instructions is still 40.

Problem 5: Memory Block Design

We want to design a SRAM memory block with 1024 32-bit words. We know that in the 6T cell the word line connects to the gate of both access transistors and each bit line connects to the s/d node of one access transistor. We assume that the capacitance of an access transistor gate is equal to the s/d capacitance. In our design we would like to balance the word-line/bit-line delay. How many address bits do we need for the row decoder and how many for the column decoder?

Solution:

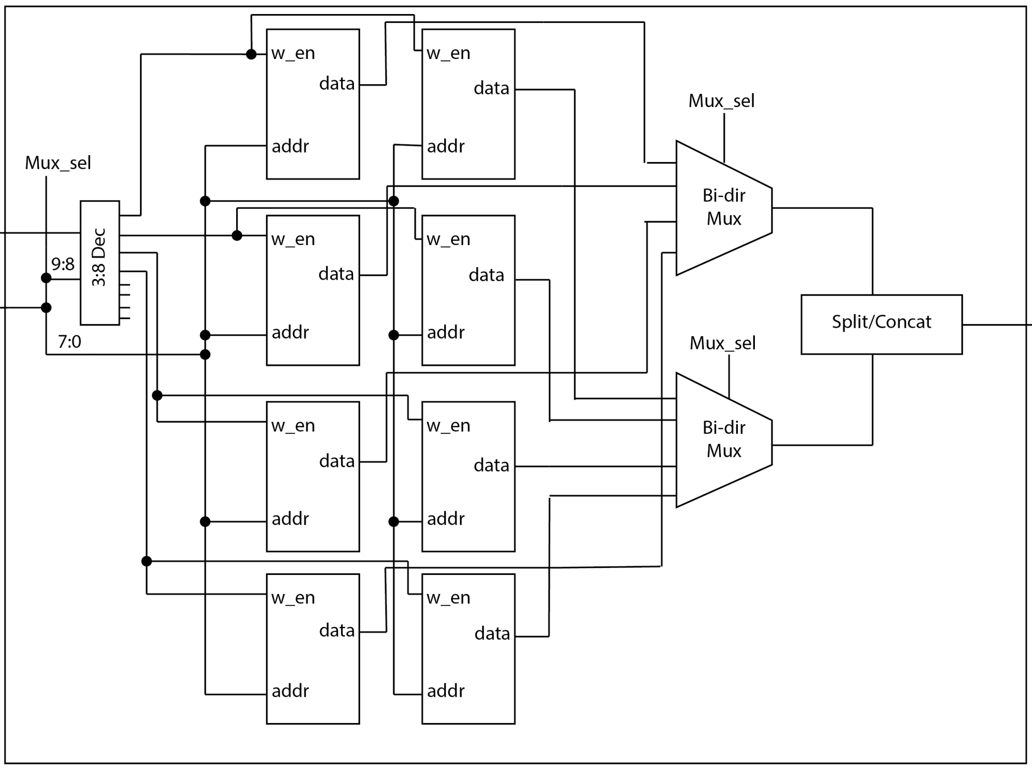
There are 2^{15} bits in the SRAM cell array. Because there are 2 bit lines for every 1 word line, each row must drive 2 access transistors while every bit line drives 1. Therefore, we must have half as many columns as rows, so we have 2^8 rows and 2^7 columns. Because words are 32 bits, we have 4 words in a line and need 2 bits to index them.

Therefore, you have 8 bits for row decoding, and 2 bits for column decoding.

Problem 6: Expanding Memory

Using as many instances of a memory block that is 256 x 16-bits, draw a circuit for a memory block that is 1024 x 32-bits.

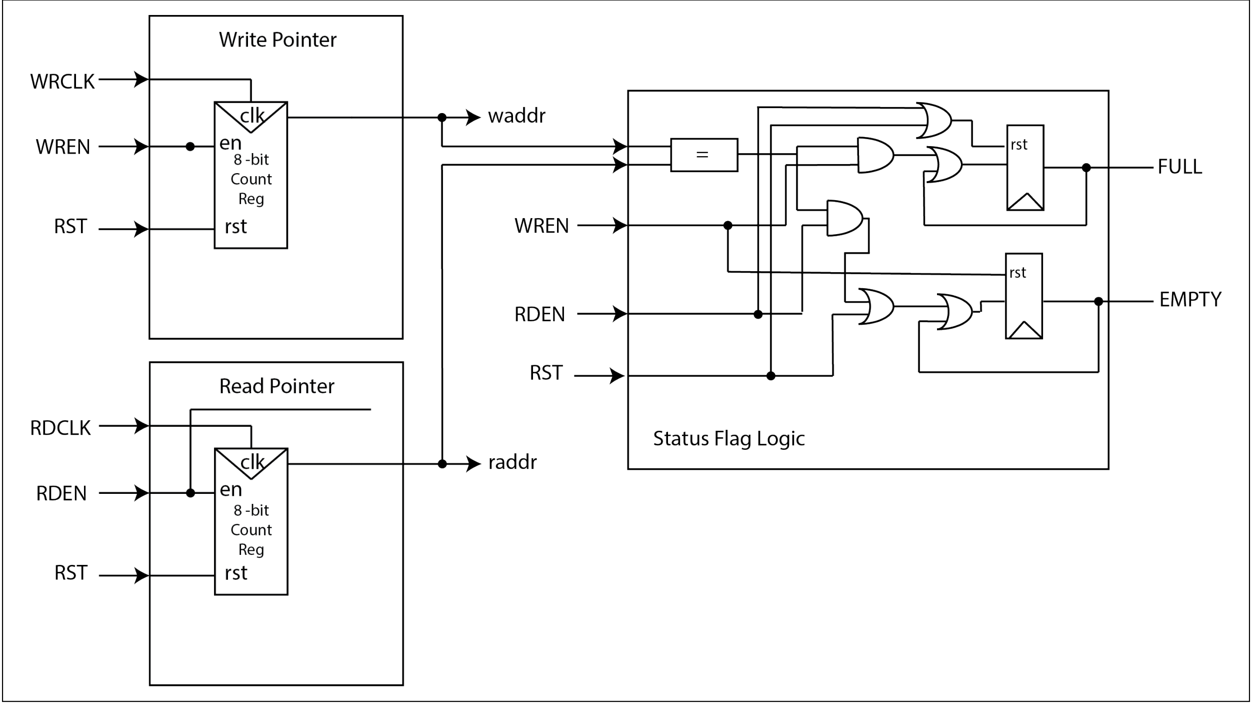
Solution:



Problem 7: Implementing a FIFO

Using the pointer based FIFO design presented in class (Lecture 17). Design the circuitry for the the FULL and EMPTY signals in the Status Flag Logic block as well as the *raddr* and *waddr* from the read and write pointer blocks. Assume that the FIFO that is 256 x 8-bits and using a simple dual ported memory block.

Solution:

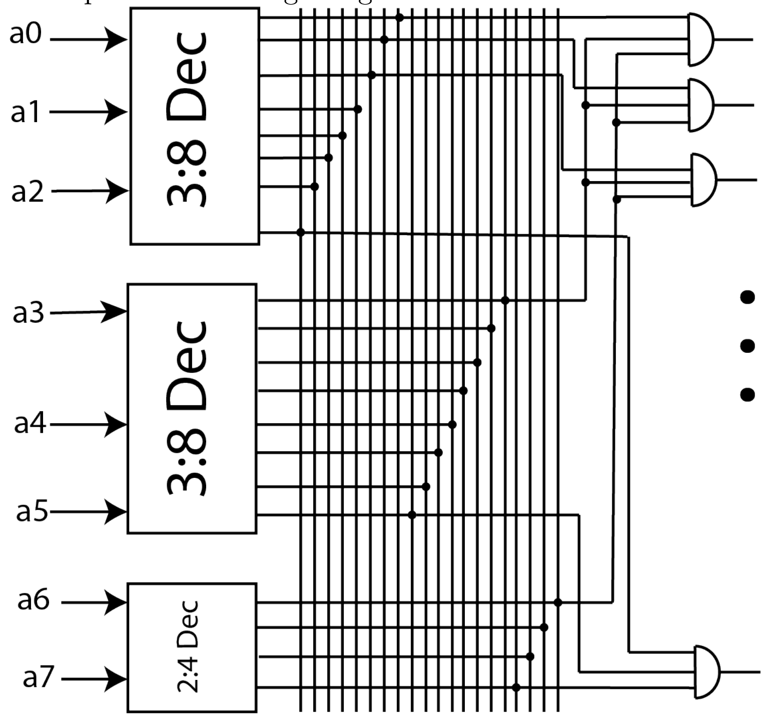


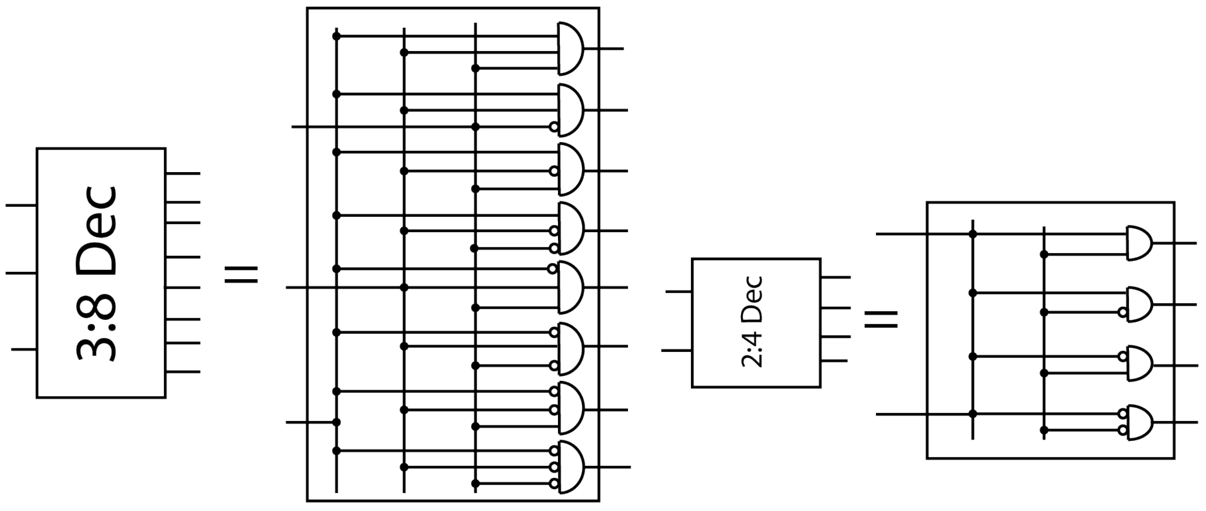
Problem 8: Pre-Decoder

Design the row decoder with 256 outputs using the pre-decoder technique discussed in lecture. You may only use 1,2, and 3-input logic gates.

Solution:

We show the pattern for using the gates below.


a0
a1
a2
3:8 Dec
a3
a4
a5
3:8 Dec
a6
a7
2:4 Dec


3:8 Dec
=
2:4 Dec
=