EECS151/251A Fall 2022 Final

Name of the person on left (or aisle)	Your Name	Name of the person on right (or aisle)
	Your SID	

Question	1	2	3	4	5	6	7	**Total**
Clobber	MT	MT	-	-	-	-	MT	
Max. points	11	12	14	10	8	9	11	**75**

Exam Notes:

1. You have 170 minutes to work, starting at 8:10 AM and ending at 11 AM Pacific Time.
2. Before 8:10 AM Pacific Time, you may write down your name, SID, names of the persons next to you, etc., on the first page, but you may NOT begin working.
3. Please write your name and SID on every page.
4. Problems are NOT organized in order of increasing difficulty. If you find that it takes too long to come up with a solution, consider moving on to the next one.
5. Both versions of the RISC-V green card are provided at the end of the document.
6. Q1,2, and 7 can be used for midterm clobber. If you get higher scores on these questions than your midterm scores, the new scores will be used for your midterm scores.

Question 1: Animation State Machine [11 points]

We are going to design an FSM to control Kirby, a video game character. Kirby has a special ability to inhale air to gain power so that it can then exhale to attack. As a result, we decide to support four states in Kirby's FSM: **Idle** (00), **Walk** (01), **Inhale** (10), and **Exhale** (11). Every cycle, Kirby can transition to a new state or stay at its current state based on its input. The player can control Kirby with two buttons as input, **walk** and **breath**. 1 indicates that a button is pressed and 0 indicates that it is not.

We make the following assumptions about this FSM:

- The input is encoded in input = {walk, breath}, e.g., an input of 2'b10 means that the walk button is pressed and the breath button is not.
- The breath button has higher priority than the walk button. i.e. when pressing both the walk and the breath buttons, Kirby will inhale/exhale but not walk.
- Kirby cannot walk while inhaling/exhaling
- Kirby can inhale infinitely long as long as the breath button was pressed and will only exhale for one cycle after the breath button is released (i.e. a 1 to 0 transition).
- When no button is pressed, the FSM has an input of {0,0} and is **NOT** stalled.

(a) Idle (b) Walk (c) Inhale (d) Exhale

Figure 1: The animation you have in your gallery, thanks to Nintendo

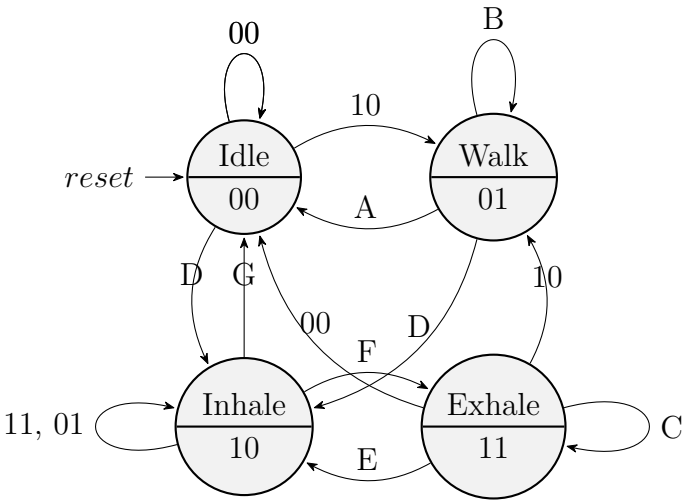


Figure 2: Kirby's FSM

(a) **Complete the FSM.** [7 points] We decided to design a Moore FSM for Kirby, and an incomplete FSM diagram is shown in Figure 2. Answer the following questions to complete the FSM.

i) What input triggers the state transition **A**?

- ◯ 00
- ◯ 01
- ◯ 11
- ◯ None

ii) What input triggers the state transition **B**?

- ◯ 00
- ◯ 01
- ◯ 10
- ◯ None

iii) What input triggers the state transition **C**?

- ◯ 00
- ◯ 01
- ◯ 10
- ◯ None

iv) What input triggers the state transition **D**?

- ◯ 11
- ◯ 01, 11
- ◯ 01
- ◯ None

v) What input triggers the state transition **E**?

- ◯ 11
- ◯ 01, 11
- ◯ 01
- ◯ None

vi) What input triggers the state transition **F**?

- ◯ 10, 00
- ◯ 01, 11
- ◯ 10, 11
- ◯ None

vii) What input triggers the state transition **G**?

- ◯ 10
- ◯ 01, 11
- ◯ 11
- ◯ None

(b) **Boolean Algebra.** [4 points] Next we will derive the `next_state` boolean expression based on the state machine. Input and state encoding: Input[0] refers to the breath button, Input[1] refers to the walk button. State encoding is shown in Figure 2. Take the **Inhale** state as an example, where State[0] = 0, and State[1] = 1.

i) What is the boolean algebra equation for next_state[1]?

- ◯ `Input[0] + State[1] * State[0]`
- ◯ `Input[0] + State[1] * !State[0]`
- ◯ `Input[1] + State[0] * !State[1]`
- ◯ `!Input[0] + State[0] * !State[1]`

ii) What is the boolean algebra equation for next_state[0]?

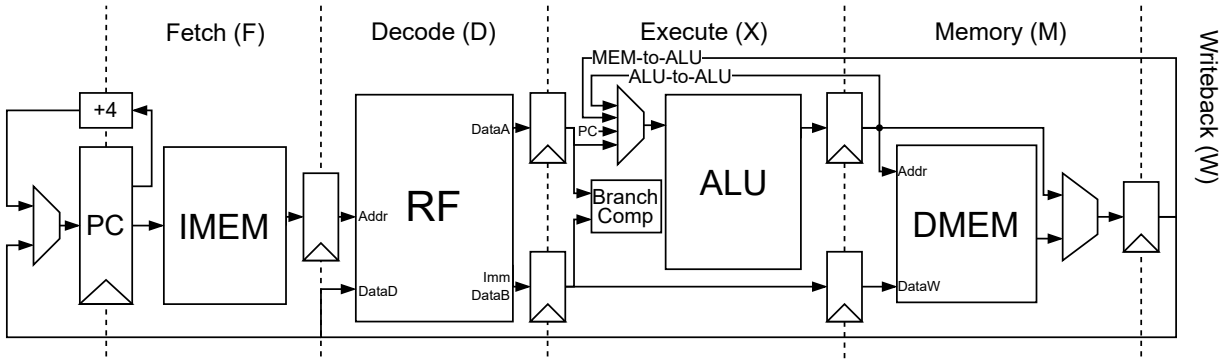
- ◯ `Input[1] * !Input[0] + State[1] * !State[0] * Input[0]`
- ◯ `Input[1] * !Input[0] + State[1] * !State[0] * !Input[0]`
- ◯ `!Input[1] * Input[0] + State[1] * !State[0] * Input[1]`
- ◯ `Input[1] * !Input[0] + !State[1] * State[0] * Input[0]`

Question 2: RISC-V and Pipelining [12 points]

This problem explores two possible optimizations for the 5-stage RISC-V pipeline design we covered in the lecture: a) an **alternative stage** design, and b) **multithreading**. Here are the basic assumptions that you can make for both of the subproblems.

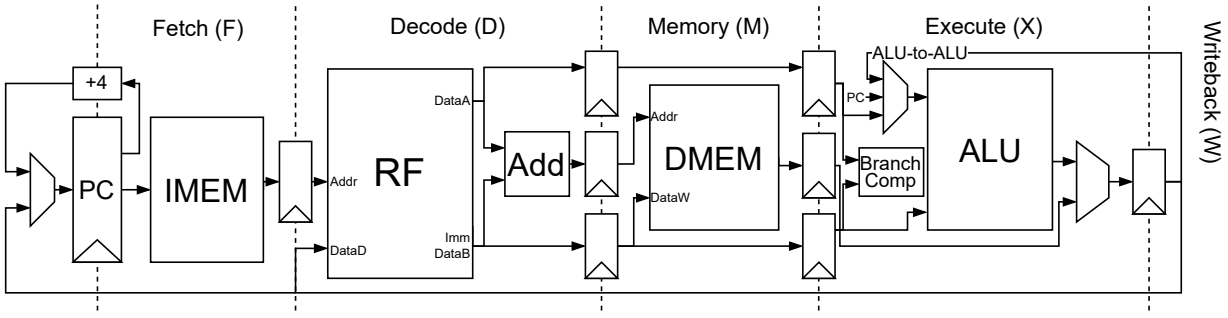
- Similar to the assumptions we have in lecture, register file, IMEM, and DMEM are all synchronous-write and asynchronous-read, where read and write can happen in one cycle.
- While some components such as immediate generator and load extension logics are omitted in the diagrams, the designs have all components needed to support RV32I.

For reference, here is the diagram of a **baseline** 5-stage pipeline design as we have learned from the lecture.



a) (6 points) Below, we have an **alternative** 5-stage pipeline design where the memory and execute stages are swapped, compared to the baseline.

Note the additional Add block in the Decode stage, since we need to calculate address for the DMEM. Branches are predicted always-not-taken.



For each of the 3 RISC-V programs given below, select whether it will take **more, the same or fewer cycles** in the new design, compared to the baseline F-D-X-M-W design shown previously.

Briefly explain the reason why. You do not have to give the exact number of cycles.

i) (2 points) The program on the right takes:

☐ More cycles

☐ The same cycles

☐ Fewer cycles

```
lw   x2, 4(x1)
add  x3, x2, x1
```

Because:

---

ii) (2 points) The program on the right takes:

☐ More cycles

☐ The same cycles

☐ Fewer cycles

```
    beqz x1, L0 // x1 was 0
    xor  x2, x2, x1
L0:
    and  x2, x2, x1
```

Because:

---

iii) (2 points) The program on the right takes:

☐ More cycles

☐ The same cycles

☐ Fewer cycles

```
add  x2, x2, x1
sw   x3, 4(x2)
```

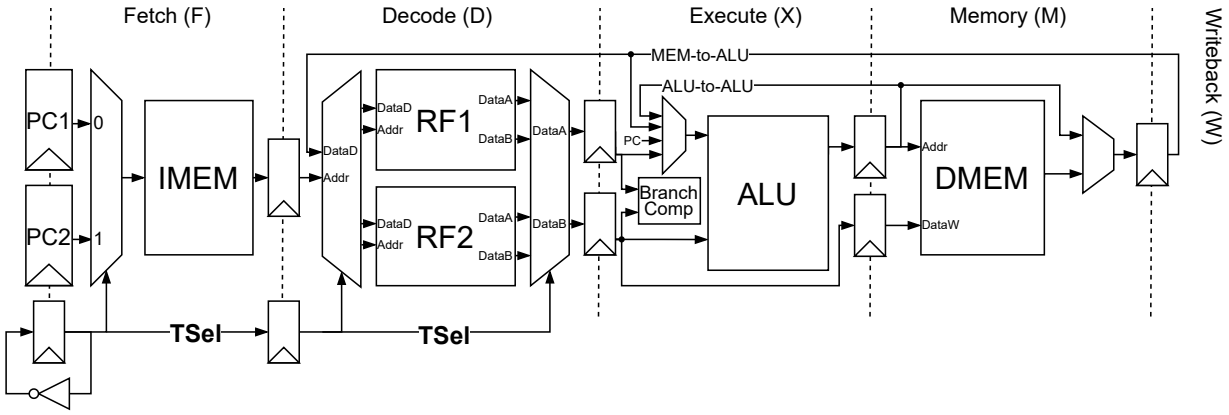
Because:

---

b) (6 points) Now, let's think about a **multithreaded** pipeline design. Multithreading works by duplicating the PC and register file for multiple threads inside the pipeline, and using a MUX selection logic to switch between them. Because PC and register file contains all the states needed to restore execution of a thread, this enables us to concurrently execute multiple threads inside a single pipeline.

In our design depicted below, the thread selection signal **TSel** changes between 0 and 1 at every cycle, so the execution alternates between the two threads at every cycle.

Note that both ALU-ALU and MEM-ALU forwarding path are implemented. The forwarding logic distinguishes between registers of different threads. For example, it does not allow forwarding from thread 1's *rd* to thread 2's *rs1*, even if *rs1* and *rd* match.



i) (2 points) First of all, let's remind ourselves of how a single-threaded design performs. Below we have a simple RV32I program running on the **baseline** design shown earlier. Fill in the pipeline table.

```
lw   x2, 4(x1)
lh   x3, 0(x2)
add  x3, x3, x1
sw   x1, 4(x3)
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw`	F	D	X	M	W											
`lh`																
`add`																
`sw`																

ii) (4 points) Now, let's run the same program on the **multithreaded** design. The two threads are both running the same program given below, starting from the same cycle. Fill in the pipeline table.
For convenience, we format instructions in different threads using *inst.thread*, e.g. `lw.1` and `add.2`.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`lw.1`	F	D	X	M	W											
`lw.2`																
`lh.1`																
`lh.2`																
`add.1`																
`add.2`																
`sw.1`																
`sw.2`																

Question 3: Logical Effort and Path Delay [14 points]

For parts (a) and (b), suppose that you are working in a process node where for a minimum size transistor ($W = 1$),

- NMOS on resistance $R_{on,n} = R$
- PMOS on resistance $R_{on,p} = 2R$
- Gate capacitance $C_g = C$ (same for both NMOS and PMOS)

Assume $\gamma = 1$.

a) Inverter design

i) How should a minimum inverter be sized in this technology? Specify the NMOS width W_n and PMOS width W_p. [1 point]

$W_n =$ _____

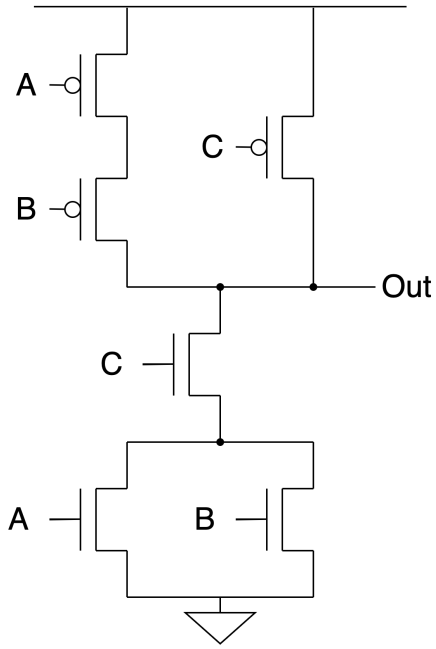
$W_p =$ _____

ii) What is the R_{eq} and C_{in} of the minimum size inverter? Express in terms of R and C, which are defined above. [1 point]

$R_{eq} =$ _____

$C_{in} =$ _____

b) You are in charge of designing the following CMOS gate:



i) Write the boolean logic function that is implemented by this gate. [1 point]

ii) How should each transistor be sized to have the same output current as the minimum size inverter? [2 points]

$W_{A,NMOS} =$ _____ $W_{A,PMOS} =$ _____

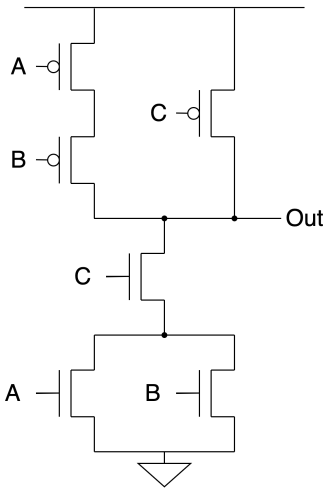
$W_{B,NMOS} =$ _____ $W_{B,PMOS} =$ _____

$W_{C,NMOS} =$ _____ $W_{C,PMOS} =$ _____

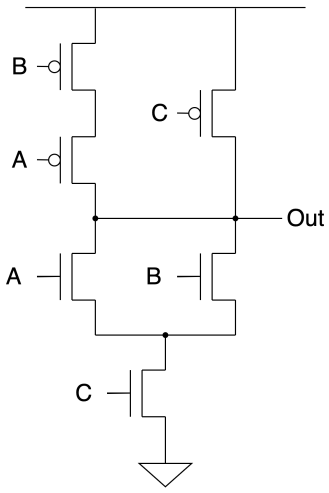
iii) Compute the logical effort (LE) and normalized parasitic delay (P) of the gate. [2 points]

$LE_A =$ _____ $LE_B =$ _____ $LE_C =$ _____ $P =$ _____

iv) Your friend proposes an alternate gate topology (as shown in Figure (b)), where the stacked transistors are flipped.



(a) Original

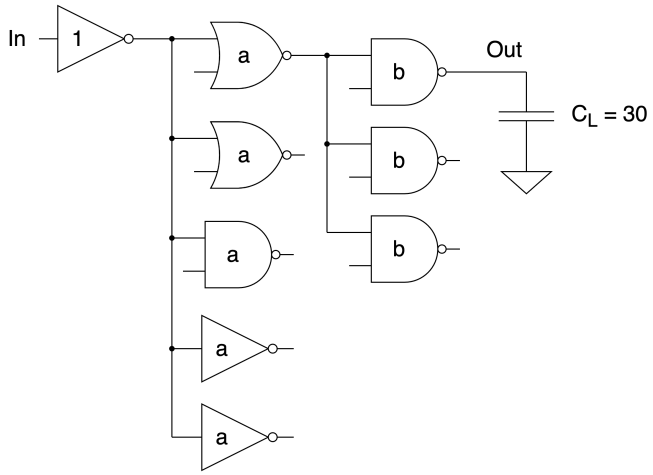


(b) Flipped transistor stacks

Which one would you pick? Briefly explain your answer. [1 point]

c) Suppose that you are now working in a **different** technology and want to optimize the following gate network. For this technology node, you're given the following information about the gates and assume $\gamma = 1$:

- $LE_{NAND} = 5/3$, $P_{NAND} = 2$
- $LE_{NOR} = 4/3$, $P_{NOR} = 2$



i) Compute gate sizes a and b that would result in the minimum delay from In to Out. [4 points]

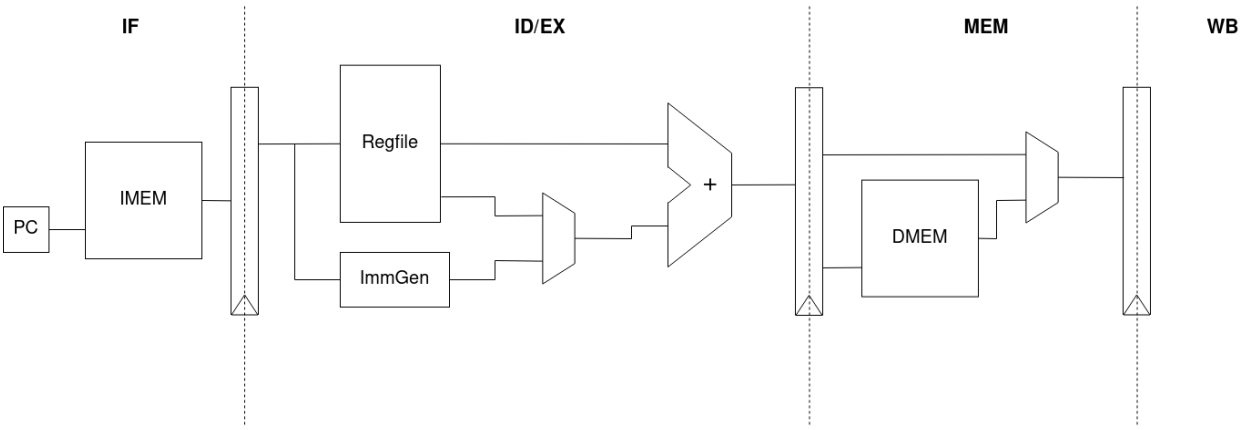
ii) What is the total delay from In to Out? Express in terms of τ_{inv}. [2 points]

Question 4: Flipflop Timing [10 points]

The EECS151 Staff team is trying to make a 4-stage, pipelined CPU, and they need your help to answer some critical timing problems.

a) Simon is in charge of designing the pipeline diagram. Next page shows a simplified diagram of his design. [2 points]

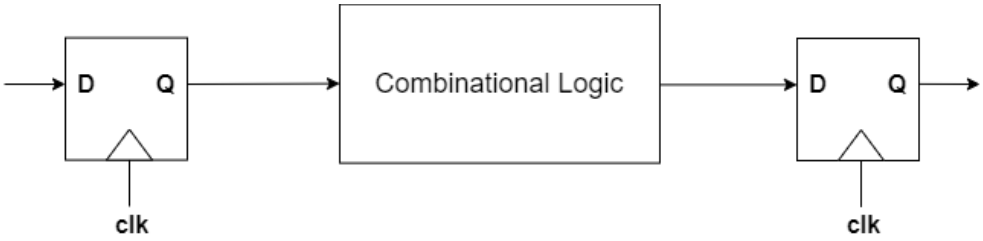
- The WB stage is implicitly implemented in the ID/EX stage, and you do not have to consider it
- You should consider the register file as a combinational logic block
- The delay of each block is as follows
  - Mux: 50 ps
  - Regfile: 200 ps
  - ALU: 400 ps
  - Mem: 600 ps
  - ImmGen: 150 ps
- Register parameters
  - Clk-Q delay: 20 ps
  - Setup time: 50 ps
  - Hold time: 40 ps
  - Clock jitter: 10 ps
  - There is no clock skew in **part a)**



i) What is the minimum clock period of this design?

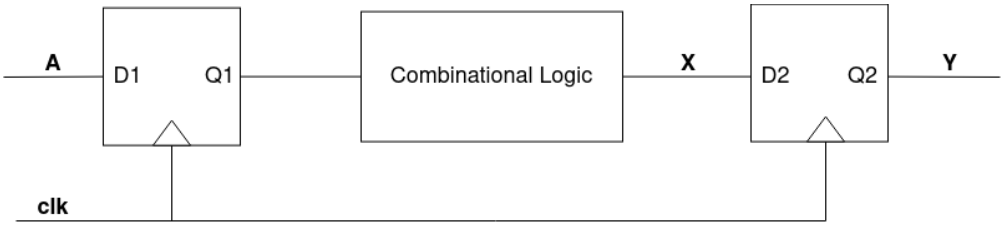
A. 700 ps
B. 710 ps
C. 720 ps
D. 730 ps
E. 740 ps

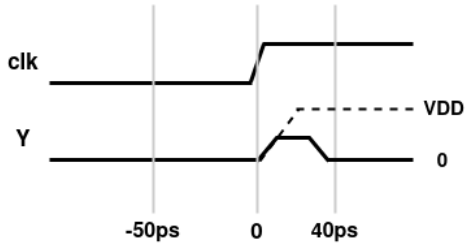
ii) What is the minimum delay of the combinational logic (shown below) to avoid hold time violations? *Hint:* The function of combinational logic block does not matter.



A. -10 ps
B. 0 ps
C. 10 ps
D. 20 ps
E. 30 ps

b) The team was able to tape out the chip and get it back from the manufacturers. However, Ella and Jennifer discovered some timing issues while testing the chip **at the minimum clock period.** The output Y should be VDD after the positive clock edge, but is 0 instead. (The dashed line shows the correct output.) [8 points]





i) What are the possible waveforms of X to result in the incorrect output waveform? Circle all that apply.

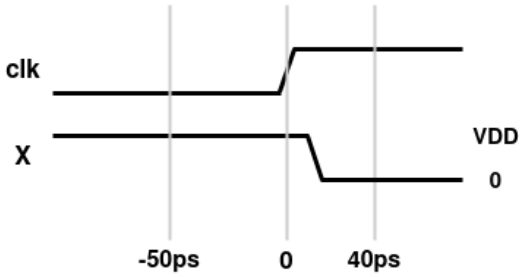


Figure 4: A

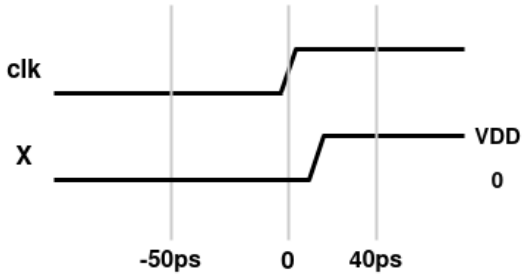


Figure 5: B

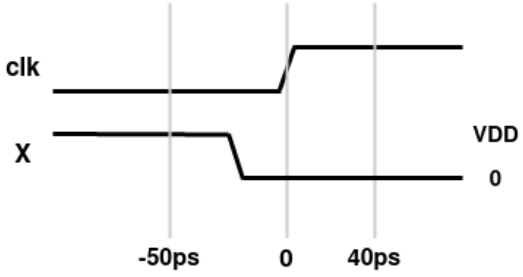


Figure 6: C

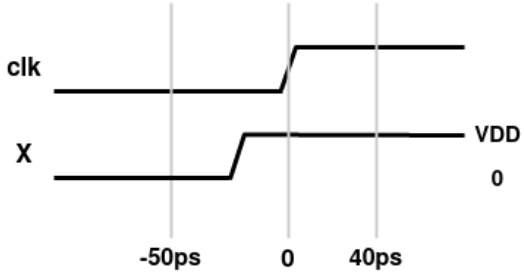


Figure 7: D

ii) Ella thinks it's a setup time violation. What following actions should she try in order to fix the chip? Select all that apply.

A. Raise clock frequency alone.

B. Lower clock frequency alone.

C. Raise VDD alone.

D. Lower VDD alone.

E. There is nothing she can do about it.

iii) Jennifer thinks it's a hold time violation. What following actions should she try in order to fix the chip? Select all that apply.

A. Raise clock frequency alone.

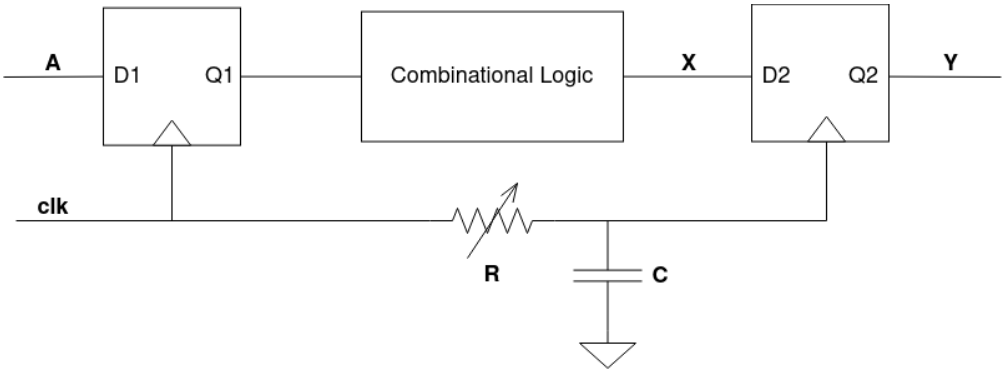
B. Lower clock frequency alone.

C. Raise VDD alone.

D. Lower VDD alone.

E. There is nothing she can do about it.

iv) Fortunately, Professor Shao had the foresight to implement a clock skew controller with a variable resistor in the clock distribution network, shown in the diagram below. Which of the following statements are true? Select all that apply.



A. If it is a setup time violation, she should increase the resistance.

B. If it is a setup time violation, she should decrease the resistance.

C. If it is a hold time violation, she should increase the resistance.

D. If it is a hold time violation, she should decrease the resistance.

E. If she increases the resistance, she might be able to run the chip at a higher frequency.

Question 5: Arithmetics [8 points]

a) (3 points) **True or False**

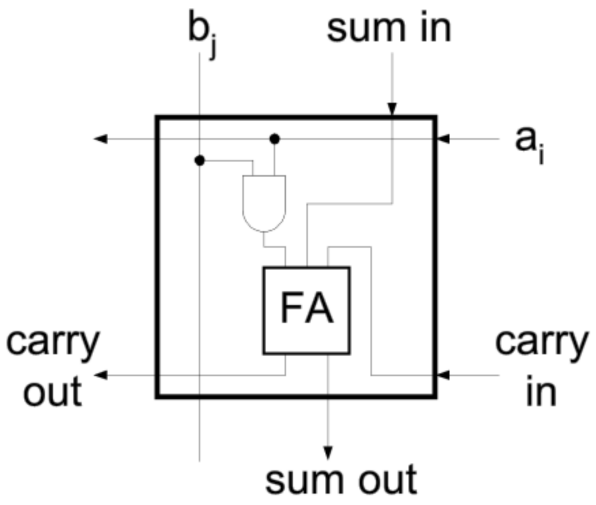
(i) ___ Recall in HW8, compared to Radix-2, a Radix-4 Parallel-Prefix Carry-lookahead Adder (CLA) reduces the depth of the adder tree by a factor of 2 (excluding the input and output stages). Theoretically, if we use a Radix-16 CLA, we can reduce the depth of the adder tree by another factor of 2, compared to a Radix-4 CLA.

(ii) ___ In a Parallel-Prefix Carry-lookahead Adder with a carry-in bit, $G_{i:0}$ is completely equivalent to $C_{out,i}$, where $G_{i:0}$ is the "group generate" from bit 0 to bit i, and $C_{out,i}$ is the carry-out bit of bit i.

(iii) ___ A shift-and-add multiplier that reuses the adder circuit for all partial products can be implemented without using registers (purely combinational).

b) (5 points) **Parallel Array Multiplier**

Consider the following unit cell for a single-cycle parallel array multiplier:



(i) Write down the Boolean expression for the Sum and Carry-out bit. You may use 2-input AND, OR and XOR only.

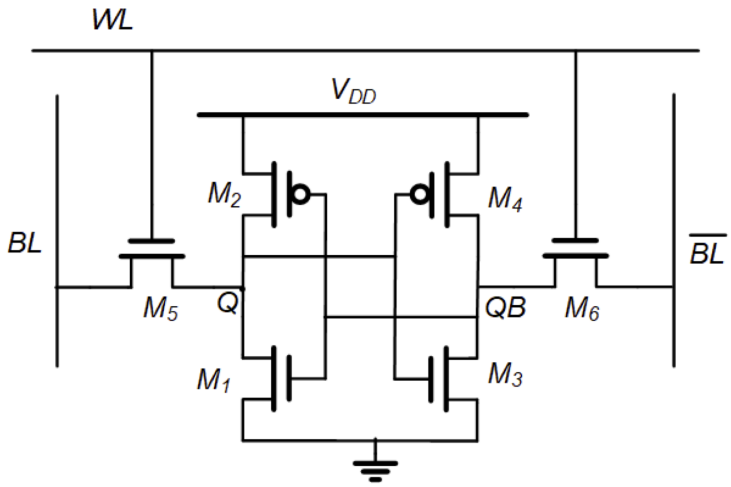
Sum_{out} = _____

C_{out} = _____

(ii) Recall the principle of multipliers: Computing the partial products, and adding the partial product terms together at the correct bit offsets. Implement a 2-bit by 2-bit multiplier by connecting several of such unit cells (**you can represent each cell using a box without the internal circuit details**). Clearly mark the following input/output: $b_1, b_0, a_1, a_0, p_3, p_2, p_1, p_0$ where p is the product. You may also use constants like 1s and 0s.

Question 6: SRAM [9 points]

Consider the 6-Transistor (6T) SRAM shown below and answer the following questions.



a) (5 points) **True or False**

(i) ___ Assuming all transistors have the channel same length. The size of the transistors should be ordered as $M_1 > M_5 > M_2$ for proper read-stability and writability.

(ii) ___ Decreasing the supply voltage V_{DD} for the cross-coupled inverters (while keeping WL and BL at the original nominal voltage) will hurt writability.

(iii) ___ Decreasing the width of the access transistors helps the writability.

(iv) ___ Increasing the width of M_1, M_3 helps the read stability.

(v) ___ If the SRAM is optimized for writability, then it is more difficult to write a "0" than a "1" into the cell in write mode.

b) (4 points) **Fill the blanks**

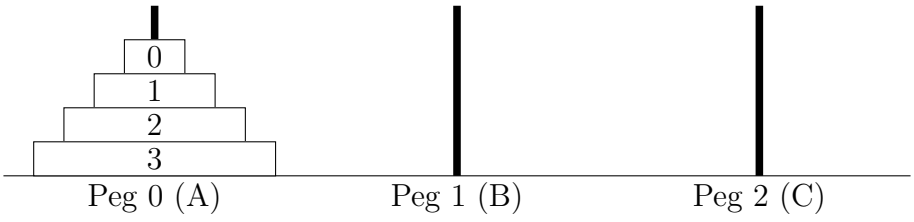
(i) An SRAM made of 6-T cells like above can atmost support ___ read port(s) and ___ write port(s).

(ii) Before a read operation, the bitlines need to be _____ before turning on the wordlines.

(iii) To decode a large-width address, it is preferable to use predecoders to break the decoding into smaller groups first, followed by the final decoders to do the rest. Suppose you have a 128-word SRAM with 64-bit words. The row-decoding logic uses 5-bit predecoders. In this case, each predecoder output will need to drive _____ final decoders.

Question 7: Veri-long [11 points]

In the classic puzzle "Tower of Hanoi", we have three pegs (rods) and N disks each sized differently. Each of the disks go on one of the pegs. Disk 0 is the **smallest** disk and $N-1$ is the **largest** disk. Initially, the disks form a pyramid on peg 0 (A).



Initial state of a Hanoi puzzle with N=4.

The goal is to move the stack to peg 2 (C) by taking the **top disk** from a stack and place it on another stack or an empty rod, **one at a time**. Importantly, **no disk may be above a smaller disk**. In this question, you will build a hardware accelerator in Verilog that outputs a move every cycle. The outputted sequence of moves should solve the puzzle with N =num_disks cycles. Assume your Hanoi solver accelerator module has the following inputs and outputs.

```
input start,
input clk,
input [5:0] num_disks,
output [1:0] src_peg,
output [1:0] dst_peg,
output done
```

a) (1 point) **Boundary conditions.** We use a 32-bit binary number to represent the disks on each peg, where bit i indicates whether disk i is on the peg (1) or not (0). We will use 3 of these 32-bit numbers to represent the 3 pegs. The code to initialize the game state with num_disks disks when start is asserted is already provided. Once a solution is found, write the logic to assert done.

```
reg [31:0] pegs[3];
wire [31:0] all_disks;

assign all_disks = (1 << num_disks) - 1;
always @(posedge clk) begin
    if (start) begin
        pegs[0] <= all_disks;
        pegs[1] <= 32'd0;
        pegs[2] <= 32'd0;
    end
end
```

```
// Check for done
assign done = pegs[0] _____ &&
              pegs[1] _____ &&
              pegs[2] _____;
```

b) (4 points) **Finding a move.** To solve the puzzle, we will repeat these 3 steps until solved:

- **Step AB:** make the legal move between pegs A and B,
- **Step AC:** make the legal move between pegs A and C,
- **Step BC:** make the legal move between pegs B and C.

For simplicity, we will assume that we always have an even number of disks.

i) (2 points) **State machine**. We will use a 2-bit number `state` to indicate which step we are on: `AB=00, AC=01, BC=10`. Fill out the expressions for peg_x and peg_y, which are **the two pegs concerned in the current step**, in no particular order.

```
reg [1:0] peg_x, peg_y, state, next_state;
assign next_state = (state + 2'd1) % 2'd3;
always @* begin
    case (state)
        2'd0: {peg_x, peg_y} = _____; // AB
        2'd1: {peg_x, peg_y} = _____; // AC
        2'd2: {peg_x, peg_y} = _____; // BC
        default: {peg_x, peg_y} = 4'd0;
    endcase
end
```

ii) (2 points) What are valid ways of updating `state`? Select all that apply.

☐
```
always @* state = start ? 2'd0 : next_state;
```

☐
```
assign state = clk ? (start ? 2'd0 : next_state) : state;
```

☐
```
always @(posedge clk) state <= start ? 2'd0 : next_state;
```

☐
```
always @(clk, start) state = start ? 2'd0 : next_state;
```

☐
```
initial state = 2'd0;
always @(posedge clk) state <= next_state;
```

c) (0 points) **Locating top disks**. Recall that any movement is made only to the top disk of a peg so we need to locate the top disk on each peg. Good news that your teammate has implemented the code for you. You don't need to worry about the implementation details but just note that `peg_x_top_index` and `peg_y_top_index` store the indexes of the top disks on `peg x` and `peg y`.

d) (4 points) **Picking the legal direction.** After we pick the two pegs, we need to choose whether we want to move from peg x to peg y or vice versa. We designate the *forward* direction as `peg x→peg y`. The forward direction is valid the if following two conditions are met:

1. `(peg_x is not empty)`
2. `(peg_y has no smaller disk than the top disk from peg_x)`

Initially, you wrote `pegs[peg_y][peg_x_top_idx-1:0] === 0` to test for **the second condition**. But turns out, this is not valid Verilog - slices cannot have a variable length! Select all valid expressions for the second condition that circumvents this restriction.

☐ `peg_x_top_idx < peg_y_top_idx`

☐ `((pegs[peg_y] & ((32'd1 << peg_x_top_idx) - 32'd1)) === 0);`

☐ `((pegs[peg_y] & ((1'd1 << peg_x_top_idx) - 1'd1)) === 0);`

☐ `((pegs[peg_y] & (32'hffffffff)[peg_x_top_idx-1:0]) === 0);`

☐ `(&(pegs[peg_y] ^ {peg_x_top_idx{1'b1}}) === 1'b0);`

e) (2 points) **Making a move.** Now that we know the two pegs and the direction of the move, we should perform the move to update our state and output the move.

```
wire [4:0] disk_index; // target disk
assign disk_index = fwd_dir_valid ? peg_x_top_idx : peg_y_top_idx;

assign src_peg = fwd_dir_valid ? peg_x : peg_y; // src peg of the move
assign dst_peg = fwd_dir_valid ? peg_y : peg_x; // dst peg of the move

always @(posedge clk) begin
  if (~start) begin

      pegs[src_peg] <= _____;

      pegs[dst_peg] <= _____;
  end
end
```

Congratulations! You have just made a Tower of Hanoi accelerator. Your accelerator dissects the iterative algorithm into a state machine, uses a combinational logic circuit to find which of the 2 moves are legal, and updates the state with that knowledge.

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

① RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

② **ARITHMETIC CORE INSTRUCTION SET**

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

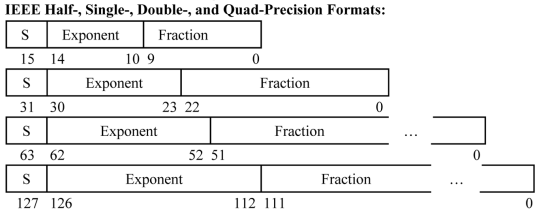
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

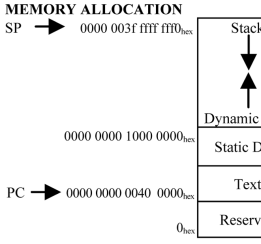
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

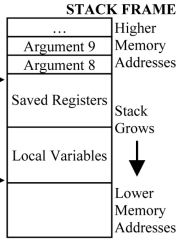
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

CS 61C Reference Card

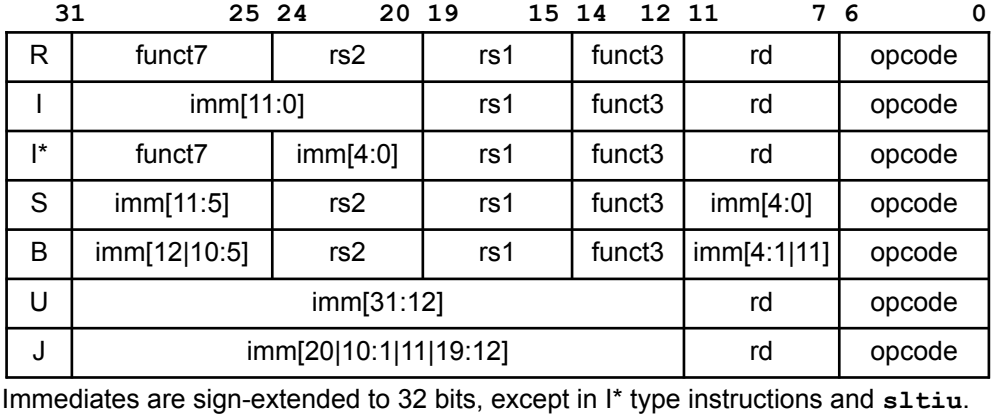
Version 1.5.0

	Instruction	Name	Description	Type	Opcode	Funct3	Funct7
Arithmetic	`add rd rs1 rs2`	ADD	`rd = rs1 + rs2`	R	`011 0011`	`000`	`000 0000`
	`sub rd rs1 rs2`	SUBtract	`rd = rs1 - rs2`	R	`011 0011`	`000`	`010 0000`
	`and rd rs1 rs2`	bitwise AND	`rd = rs1 & rs2`	R	`011 0011`	`111`	`000 0000`
	`or rd rs1 rs2`	bitwise OR	`rd = rs1 | rs2`	R	`011 0011`	`110`	`000 0000`
	`xor rd rs1 rs2`	bitwise XOR	`rd = rs1 ^ rs2`	R	`011 0011`	`100`	`000 0000`
	`sll rd rs1 rs2`	Shift Left Logical	`rd = rs1 << rs2`	R	`011 0011`	`001`	`000 0000`
	`srl rd rs1 rs2`	Shift Right Logical	`rd = rs1 >> rs2` (Zero-extend)	R	`011 0011`	`101`	`000 0000`
	`sra rd rs1 rs2`	Shift Right Arithmetic	`rd = rs1 >> rs2` (Sign-extend)	R	`011 0011`	`101`	`010 0000`
	`slt rd rs1 rs2`	Set Less Than (signed)	`rd = (rs1 < rs2) ? 1 : 0`	R	`011 0011`	`010`	`000 0000`
	`sltu rd rs1 rs2`	Set Less Than (Unsigned)		R	`011 0011`	`011`	`000 0000`
	`addi rd rs1 imm`	ADD Immediate	`rd = rs1 + imm`	I	`001 0011`	`000`	
	`andi rd rs1 imm`	bitwise AND Immediate	`rd = rs1 & imm`	I	`001 0011`	`111`	
	`ori rd rs1 imm`	bitwise OR Immediate	`rd = rs1 | imm`	I	`001 0011`	`110`	
	`xori rd rs1 imm`	bitwise XOR Immediate	`rd = rs1 ^ imm`	I	`001 0011`	`100`	
	`slli rd rs1 imm`	Shift Left Logical Immediate	`rd = rs1 << imm`	I*	`001 0011`	`001`	`000 0000`
	`srli rd rs1 imm`	Shift Right Logical Immediate	`rd = rs1 >> imm` (Zero-extend)	I*	`001 0011`	`101`	`000 0000`
	`srai rd rs1 imm`	Shift Right Arithmetic Immediate	`rd = rs1 >> imm` (Sign-extend)	I*	`001 0011`	`101`	`010 0000`
	`slti rd rs1 imm`	Set Less Than Immediate (signed)	`rd = (rs1 < imm) ? 1 : 0`	I	`001 0011`	`010`	
	`sltiu rd rs1 imm`	Set Less Than Immediate (Unsigned)		I	`001 0011`	`011`	
Memory	`lb rd imm(rs1)`	Load Byte	`rd` = 1 byte of memory at address `rs1 + imm`, sign-extended	I	`000 0011`	`000`	
	`lbu rd imm(rs1)`	Load Byte (Unsigned)	`rd` = 1 byte of memory at address `rs1 + imm`, zero-extended	I	`000 0011`	`100`	
	`lh rd imm(rs1)`	Load Half-word	`rd` = 2 bytes of memory starting at address `rs1 + imm`, sign-extended	I	`000 0011`	`001`	
	`lhu rd imm(rs1)`	Load Half-word (Unsigned)	`rd` = 2 bytes of memory starting at address `rs1 + imm`, zero-extended	I	`000 0011`	`101`	
	`lw rd imm(rs1)`	Load Word	`rd` = 4 bytes of memory starting at address `rs1 + imm`	I	`000 0011`	`010`	
	`sb rs2 imm(rs1)`	Store Byte	Stores least-significant byte of `rs2` at the address `rs1 + imm` in memory	S	`010 0011`	`000`	
	`sh rs2 imm(rs1)`	Store Half-word	Stores the 2 least-significant bytes of `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`001`	
	`sw rs2 imm(rs1)`	Store Word	Stores `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`010`	

	Instruction	Name	Description	Type	Opcode	Funct3
Control	`beq rs1 rs2 label`	Branch if EQual	`if (rs1 == rs2) PC = PC + offset`	B	`110 0011`	`000`
	`bge rs1 rs2 label`	Branch if Greater or Equal (signed)	`if (rs1 >= rs2) PC = PC + offset`	B	`110 0011`	`101`
	`bgeu rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		B	`110 0011`	`111`
	`blt rs1 rs2 label`	Branch if Less Than (signed)	`if (rs1 < rs2) PC = PC + offset`	B	`110 0011`	`100`
	`bltu rs1 rs2 label`	Branch if Less Than (Unsigned)		B	`110 0011`	`110`
	`bne rs1 rs2 label`	Branch if Not Equal	`if (rs1 != rs2) PC = PC + offset`	B	`110 0011`	`001`
	`jal rd label`	Jump And Link	`rd = PC + 4` `PC = PC + offset`	J	`110 1111`	
	`jalr rd rs1 imm`	Jump And Link Register	`rd = PC + 4` `PC = rs1 + imm`	I	`110 0111`	`000`
Other	`auipc rd imm`	Add Upper Immediate to PC	`rd = PC + (imm << 12)`	U	`001 0111`	
	`lui rd imm`	Load Upper Immediate	`rd = imm << 12`	U	`011 0111`	
	`ebreak`	Environment BREAK	Asks the debugger to do something (`imm = 0`)	I	`111 0011`	`000`
	`ecall`	Environment CALL	Asks the OS to do something (`imm = 1`)	I	`111 0011`	`000`
Ext	`mul rd rs1 rs2`	MULtiply (part of mul ISA extension)	`rd = rs1 * rs2`	(omitted)		

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	
Caller saved registers					
Callee saved registers (except `x0`, `gp`, `tp`)					

Pseudoinstruction	Name	Description	Translation
`beqz rs1 label`	Branch if EQuals Zero	`if (rs1 == 0) PC = PC + offset`	`beq rs1 x0 label`
`bnez rs1 label`	Branch if Not Equals Zero	`if (rs1 != 0) PC = PC + offset`	`bne rs1 x0 label`
`j label`	Jump	`PC = PC + offset`	`jal x0 label`
`jr rs1`	Jump Register	`PC = rs1`	`jalr x0 rs1 0`
`la rd label`	Load absolute Address	`rd = &label`	`auipc`, `addi`
`li rd imm`	Load Immediate	`rd = imm`	`lui` (if needed), `addi`
`mv rd rs1`	MoVe	`rd = rs1`	`addi rd rs1 0`
`neg rd rs1`	NEGate	`rd = -rs1`	`sub rd x0 rs1`
`nop`	No OPeration	do nothing	`addi x0 x0 0`
`not rd rs1`	bitwise NOT	`rd = ~rs1`	`xori rd rs1 -1`
`ret`	RETurn	`PC = ra`	`jalr x0 x1 0`



Immediates are sign-extended to 32 bits, except in I* type instructions and `sltiu`.

Selected ASCII values

HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR
0x20	32	SPACE	0x30	48	0	0x40	64	@	0x50	80	P	0x60	96	`	0x70	112	p
0x21	33	!	0x31	49	1	0x41	65	A	0x51	81	Q	0x61	97	a	0x71	113	q
0x22	34	"	0x32	50	2	0x42	66	B	0x52	82	R	0x62	98	b	0x72	114	r
0x23	35	#	0x33	51	3	0x43	67	C	0x53	83	S	0x63	99	c	0x73	115	s
0x24	36	$	0x34	52	4	0x44	68	D	0x54	84	T	0x64	100	d	0x74	116	t
0x25	37	%	0x35	53	5	0x45	69	E	0x55	85	U	0x65	101	e	0x75	117	u
0x26	38	&	0x36	54	6	0x46	70	F	0x56	86	V	0x66	102	f	0x76	118	v
0x27	39	'	0x37	55	7	0x47	71	G	0x57	87	W	0x67	103	g	0x77	119	w
0x28	40	(	0x38	56	8	0x48	72	H	0x58	88	X	0x68	104	h	0x78	120	x
0x29	41	)	0x39	57	9	0x49	73	I	0x59	89	Y	0x69	105	i	0x79	121	y
0x2A	42	*	0x3A	58	:	0x4A	74	J	0x5A	90	Z	0x6A	106	j	0x7A	122	z
0x2B	43	+	0x3B	59	;	0x4B	75	K	0x5B	91	[	0x6B	107	k	0x7B	123	{
0x2C	44	,	0x3C	60	<	0x4C	76	L	0x5C	92	\	0x6C	108	l	0x7C	124	|
0x2D	45	-	0x3D	61	=	0x4D	77	M	0x5D	93	]	0x6D	109	m	0x7D	125	}
0x2E	46	.	0x3E	62	>	0x4E	78	N	0x5E	94	^	0x6E	110	n	0x7E	126	~
0x2F	47	/	0x3F	63	?	0x4F	79	O	0x5F	95	_	0x6F	111	o	0x00	0	NULL

C Format String Specifiers

Specifier	Output
d or i	Signed decimal integer
u	Unsigned decimal integer
o	Unsigned octal
x	Unsigned hexadecimal integer, lowercase
X	Unsigned hexadecimal integer, uppercase
f	Decimal floating point, lowercase
F	Decimal floating point, uppercase
e	Scientific notation (significand/exponent), lowercase
E	Scientific notation (significand/exponent), uppercase
g	Use the shortest representation: %e or %f
G	Use the shortest representation: %E or %F
a	Hexadecimal floating point, lowercase
A	Hexadecimal floating point, uppercase
c	Character
s	String of characters
p	Pointer address

IEEE 754 Floating Point Standard

	Sign	Exponent	Significand
Single Precision	1 bit	8 bits (bias = -127)	23 bits
Double Precision	1 bit	11 bits (bias = -1023)	52 bits
Quad Precision	1 bit	15 bits (bias = -16383)	112 bits

Standard exponent bias: $-(2^{E-1}-1)$ where E is the number of exponent bits

SI Prefixes

Size	Prefix	Symbol	Size	Prefix	Symbol	Size	Prefix	Symbol
10^{-3}	milli-	m	10^{3}	kilo-	k	2^{10}	kibi-	Ki
10^{-6}	micro-	µ	10^{6}	mega-	M	2^{20}	mebi-	Mi
10^{-9}	nano-	n	10^{9}	giga-	G	2^{30}	gibi-	Gi
10^{-12}	pico-	p	10^{12}	tera-	T	2^{40}	tebi-	Ti
10^{-15}	femto-	f	10^{15}	peta-	P	2^{50}	pebi-	Pi
10^{-18}	atto-	a	10^{18}	exa-	E	2^{60}	exbi-	Ei
10^{-21}	zepto-	z	10^{21}	zetta-	Z	2^{70}	zebi-	Zi
10^{-24}	yocto-	y	10^{24}	yotta-	Y	2^{80}	yobi-	Yi

Laws of Boolean Algebra

$$x \cdot \overline{x} = 0 \qquad x + \overline{x} = 1 \qquad (xy)z = x\,(yz)$$
$$x \cdot 0 = 0 \qquad x + 1 = 1 \qquad (x + y) + z = x + (y + z)$$
$$x \cdot 1 = x \qquad x + 0 = x \qquad x\,(y + z) = xy + xz$$
$$x \cdot x = x \qquad x + x = x \qquad x + yz = (x + y)\,(x + z)$$
$$x \cdot y = y \cdot x \qquad x + y = y + x \qquad \overline{x \cdot y} = \overline{x} + \overline{y}$$
$$xy + x = x \qquad (x + y)x = x \qquad \overline{(x + y)} = \overline{x} \cdot \overline{y}$$