Your Name (first last) SID

EECS 151/251A Fall 2020 Midterm 1

October 6, 2020

Question	1	2	3	4	5	Total
Sugg. time (mins)	15	15	15	20	15	**80**
Max. points	15	15	16	18	16	**80**

Exam Notes:

Between 3:30pm PST and 3:40pm PST, you may set up your recording, print the exam or transfer it to another device as needed, etc., but you may NOT begin working.

You have 80 minutes to work, starting at 3:40pm PST and ending at 5:00pm PST.

Please keep the Google Doc page that you received at 3:30pm PST open during the exam. It contains the following information:

1. A link to the exam PDF
2. A form for exam questions and reporting technical difficulties
3. A form for your exam recording link
4. Gradescope submission link
5. Exam errata
6. Summary of exam steps

Problem 1: It's all logical... [15 points, 15 minutes]

(a) Given $F = \overline{(\bar{a} + c)}b + c\bar{d}$, use De Morgan's law to derive $\bar{F}$. Write the equation in product-of-sums form.

(b) Use a K-map to simplify the following expression and leave in product-of-sums form:

$$F = \bar{b}d + b\bar{c}\bar{d} + ab\bar{c}\bar{d} + abc\bar{d}$$

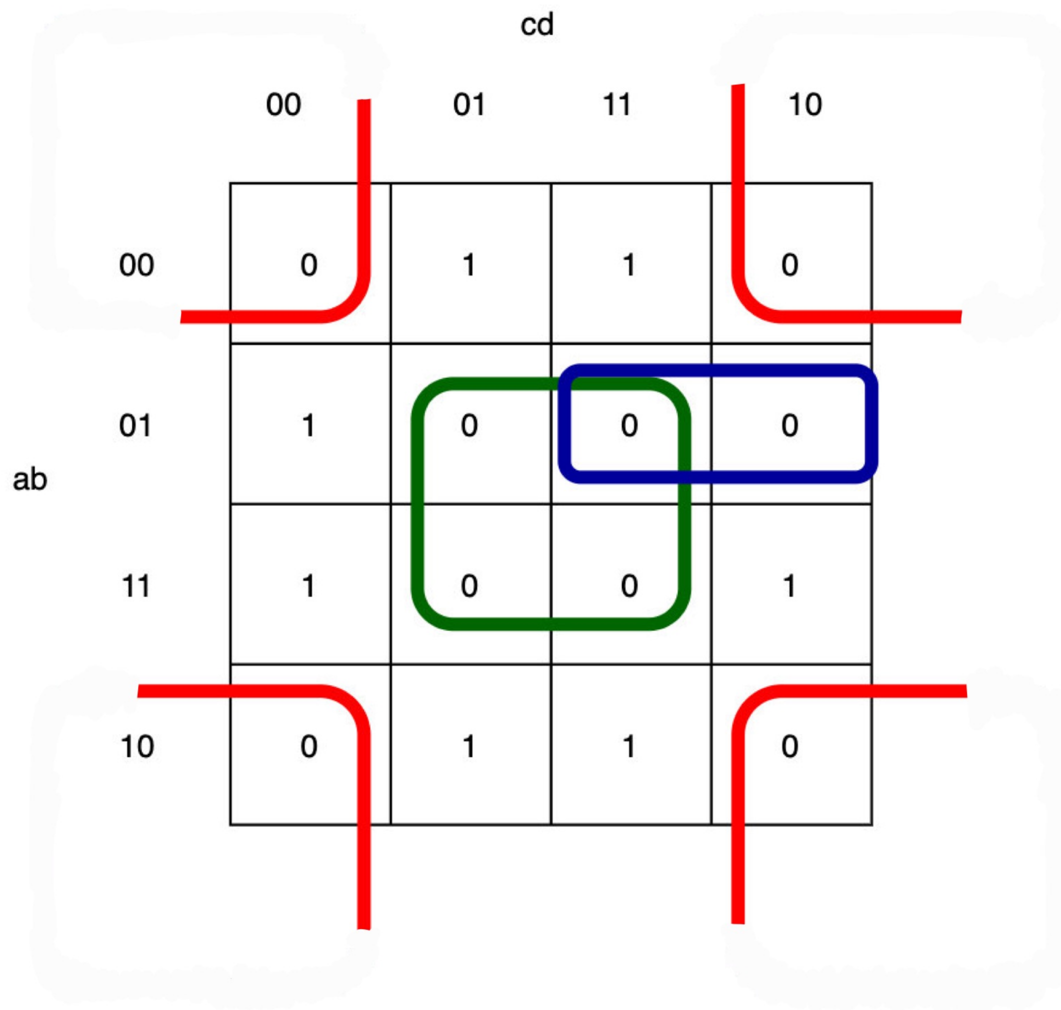
Your expression should have no more than 5 terms. Include the K-map in your solution.

(c) How many unique truth tables are there with `m` inputs and `n` outputs?

Answer: ________

Solution:

1. $\bar{F} = \overline{ab\bar{c} + c\bar{d}}$
   $= (\overline{ab\bar{c}})(\overline{c\bar{d}})$
   $= (\bar{a} + \bar{b} + c)(\bar{c} + d)$



2. $(b + d)(\bar{b} + \bar{d})(a + \bar{b} + \bar{c})$
   Other kmaps, groupings and expressions are possible.

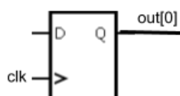
3. There are 2^m rows in a truth table of m inputs. Each row has 2^n possible values. So there are $(2^n)^{(2^m)}$ combinational logic circuits in total.

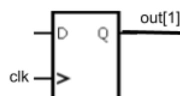
Problem 2: VERIfiably LOGical [15 points, 15 minutes]

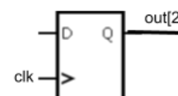
Consider the following Verilog module:

```
module my_module(
    input clk, load,
    input [2:0] in,
    output reg [2:0] out
);
    always @(posedge clk) begin
        if (load) out <= in;
        else begin
            out[0] = ~out[0];
            if (out[0])
                out[1] <= ~out[1];
            if (out[0] & out[1])
                out[2] <= ~out[2];
        end
    end
endmodule
```

(a) Draw the circuit diagram for this design. You may use the module inputs (e.g. `in[0]`), constants, muxes, inverters, and/or 2-input logic gates.







(b) Say we load 3'b011 using our `load` and `in` input signals. We then deassert `load`. What is the value of `out` for the first 6 cycles?

Cycle	`out`
0	011
1	
2	
3	
4	
5	

Now, consider this similar module:

```
module my_module(
    input clk, load,
    input [2:0] in,
    output reg [2:0] out
);
    always @(posedge clk) begin
        if (load) out = in;
        else begin
            out[0] = ~out[0];
            if (out[0])
                out[1] = ~out[1];
            if (out[0] & out[1])
                out[2] = ~out[2];
        end
    end
endmodule
```

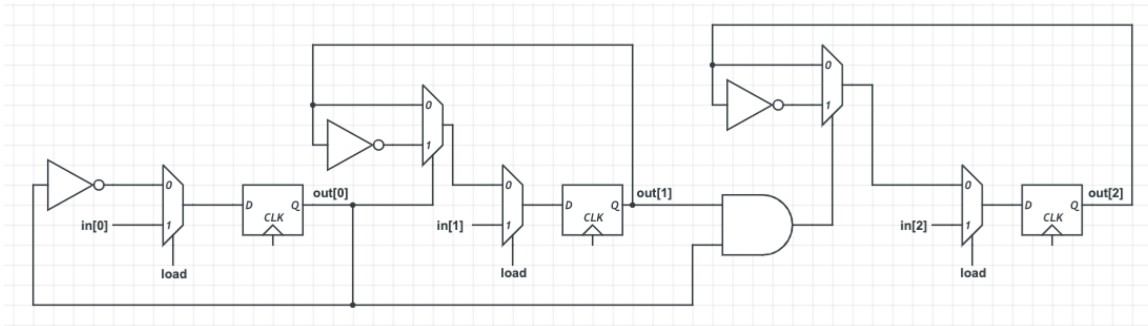
(c) Say we load 3'b011 again using our `load` and `in` input signals. What is the value of `out` for the first 6 cycles?

Cycle	`out`
0	011
1	
2	
3	
4	
5	

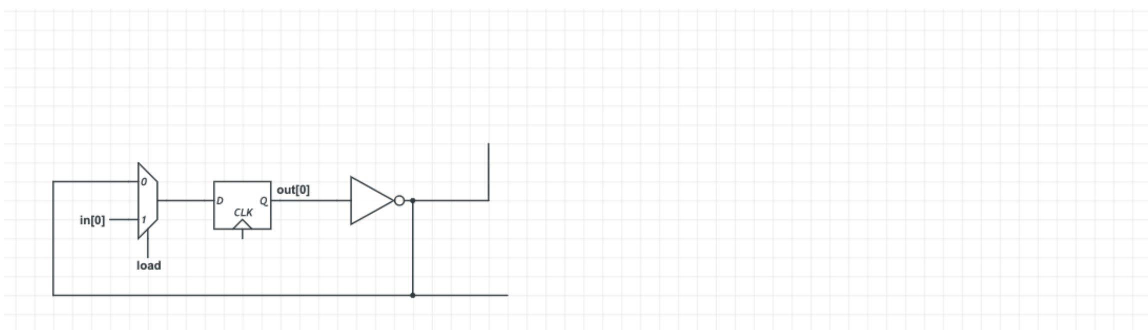
Solution:

(a) During the exam, a clarification was made that all assignments in this part were meant to be non-blocking. However, credit was awarded for those who solved the question as written.

Here is the circuit diagram for the corrected (non-blocking) code:



The code as written produces a slightly different circuit (assume the rest of the circuit is the same as in the previous diagram).



(b) With the corrected code (all assignments non-blocking), this Verilog describes a counter.

Cycle	`out`
0	011
1	100
2	101
3	110
4	111
5	000

As written (`out[0] = ~out[0]`):

Cycle	`out`
0	011
1	010
2	101
3	100
4	111
5	110

(c) With all assignments blocking, the module now describes a downward counter.

Cycle	`out`
0	011
1	010
2	001
3	000
4	111
5	110

Problem 3: State of the Machine [16 points, 15 minutes]

Pleased with your work on the charger and battery system from HW1, 151Laptops & Co. has decided to enlist your help once again. You are tasked with building a Mealy-type FSM for managing the laptop's clock frequency depending on load and temperature. The requirements are as follows:

1. Input: The FSM has two 1-bit inputs (temperature and load), that are asynchronous to the clock controlling the FSM (clk). These can be concatenated into a 2-bit value {temperature, load} (temperature is the leftmost bit).
   (a) For temperature a value of 1'b0 represents COOL, a value of 1'b1 represents HOT.
   (b) For load a value of 1'b0 represents IDLE, a value of 1'b1 represents BUSY.

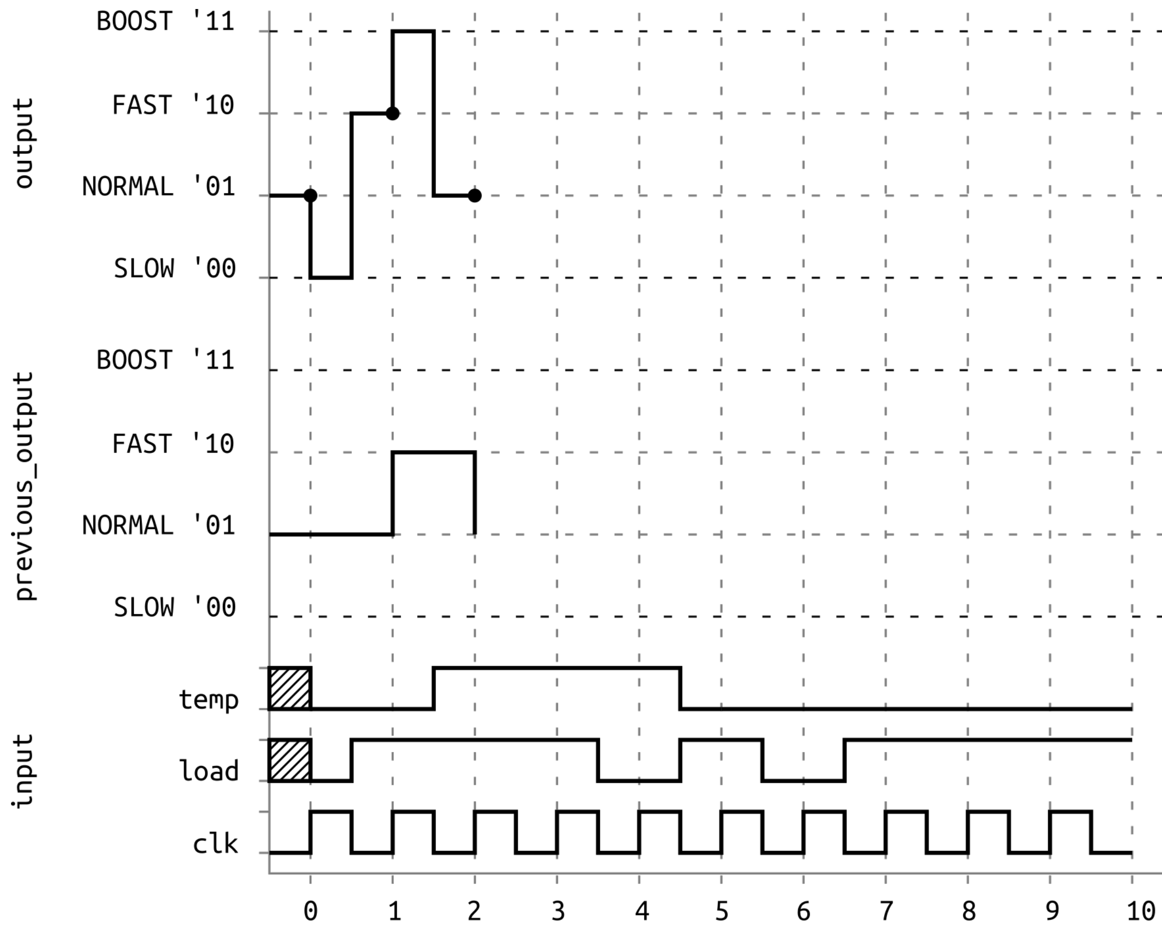
COOL & IDLE	COOL & BUSY	HOT & IDLE	HOT & BUSY
2'b00	2'b01	2'b10	2'b11

2. Output: The FSM has a 2-bit output indicating the frequency that should be used. 2'b00 is the slowest clock and 2'b11 is the fastest clock.

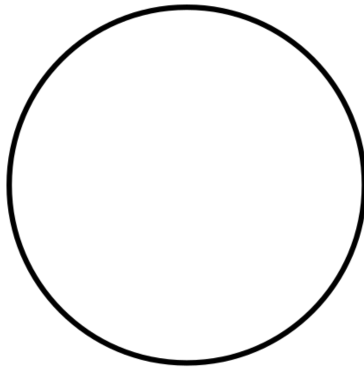
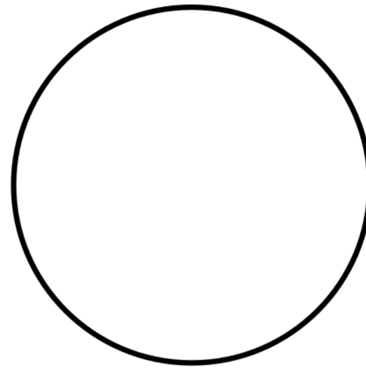
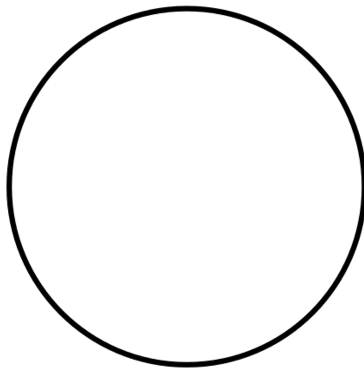
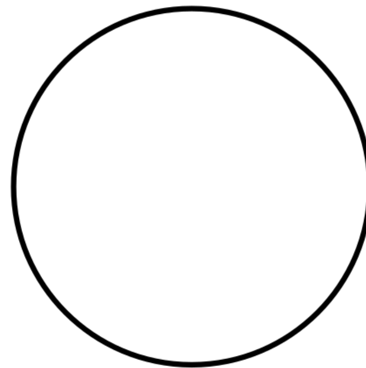
SLOW	NORMAL	FAST	BOOST
2'b00	2'b01	2'b10	2'b11

3. A useful quantity in thinking about this FSM is the **previous output**.
   (a) **Previous output** at any time is defined as the value of the output sampled just before the most recent posedge clk.
   (b) **Previous output** is initially NORMAL.
4. In the following scenarios, the output frequency will be dropped to one-level lower than the **previous output**, unless the previous output is already SLOW:
   (a) If the temperature is HOT, or
   (b) If the load is IDLE
5. In the following scenario, the output frequency will be increased to one-level higher than the **previous output**, unless the previous output is already BOOST:
   (a) If temperature is COOL and load is BUSY

(a) **Demonstrate your understanding:** Fill in the waveform with the values of output and previous output. If you are doing this on a separate paper, it is acceptable to draw only output and previous output so long as you label your axis clearly and mark time.

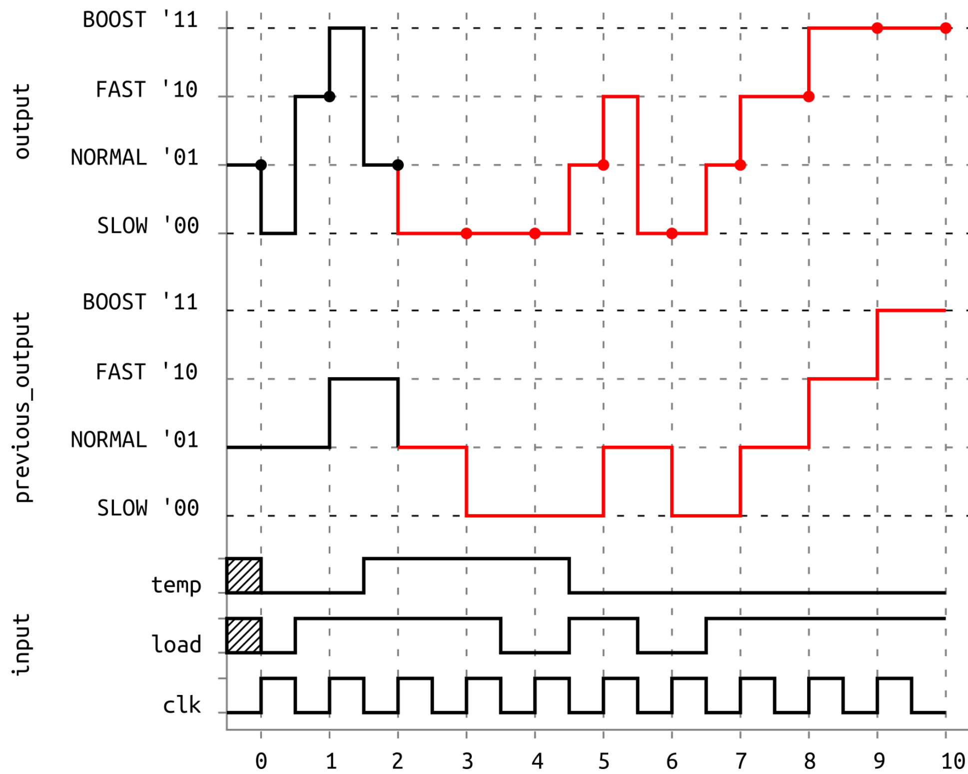


(b) **Design it:** Draw the state-transition diagram for your Mealy machine. You may use asterisks (*) to represent "don't care" values: an input of 2'b1* would indicate temperature=HOT and load is anything.

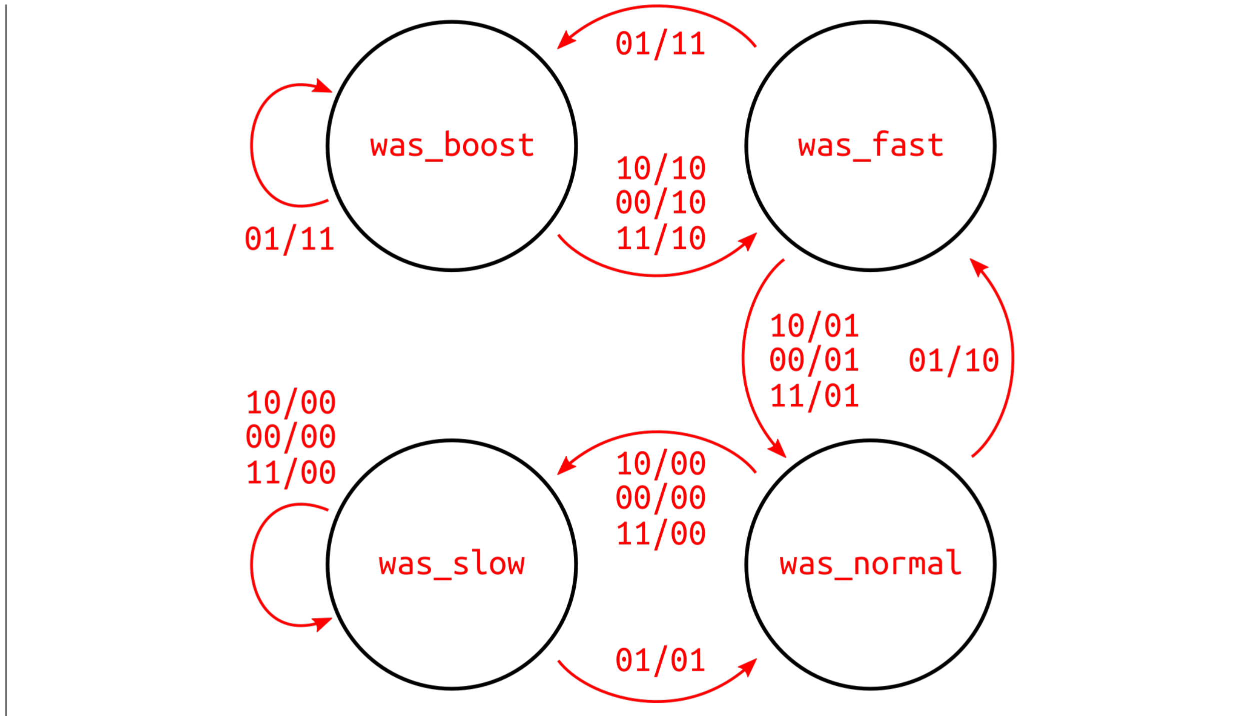





Solution:

(a) **Demonstrate your understanding:** Fill in the waveform with values of the output



(b) **Design it:** Draw the state-transition diagram for your mealy machine. You may use asterisks (*) to represent "don't care": an input of 2'b1* would indicate temperature=HOT and load is anything.


01/11
was_boost
was_fast
10/10
00/10
11/10
01/11
10/01
00/01
11/01
01/10
10/00
00/00
11/00
10/00
00/00
11/00
was_slow
was_normal
01/01

Problem 4: Follow the Instructions [18 points, 20 minutes]

You may refer to the RISC-V Green Card on the last pages of this exam.

(a) RISC-V assembly defines a set of standard pseudo-instructions that can be implemented with the base RV32I instructions. Out of the branching pseudo-instructions, select the ones that can be implemented with a single `blt` (branch less than) instruction.

- ☐ `beqz rs, offset` (Branch if $= 0$)
- ☐ `bnez rs, offset` (Branch if $\neq 0$)
- ☐ `blez rs, offset` (Branch if ≤ 0)
- ☐ `bgez rs, offset` (Branch if ≥ 0)
- ☐ `bltz rs, offset` (Branch if < 0)
- ☐ `bgtz rs, offset` (Branch if > 0)
- ☐ `bgt rs, rt, offset` (Branch if `rs` $>$ `rt`)
- ☐ `ble rs, rt, offset` (Branch if `rs` $\leq$ `rt`)
- ☐ `bgtu rs, rt, offset` (Branch if `rs` $>$ `rt`, unsigned)
- ☐ `bleu rs, rt, offset` (Branch if `rs` $\leq$ `rt`, unsigned)

Solution:

`bltz`, `bgtz`, and `bgt`

(b) `JALR` (jump and link register) uses 12 bits of the immediate, meaning we can only jump to an offset of within 2^{11} or 2KiB from the base address. Give the combination of 2 instructions that would allow us to jump to:

(i) A "PC-absolute" address, i.e. a PC address specified by an absolute 32-bit constant

(ii) A "PC-relative" address, i.e. a PC address specified by a 32-bit offset from the current PC

Solution:

(i) `LUI` + `JALR`

(ii) `AUIPC` + `JALR`

(c) Notice how `imm[0]` is discarded (always 0) in B-type instructions. Select all statements below that are TRUE for this ISA design choice.

- ☐ This explains the difference between S-type and B-type instruction formats.
- ☐ The branch target offset range is $\pm 2^{12} = \pm 4096$ instructions.
- ☐ An ISA extension with only 16 integer registers prohibits also discarding `imm[1]`.
- ☐ An ISA extension with 16-bit instructions prohibits also discarding `imm[1]`.

Solution:

Choices #1 and #4
#2: bytes, not instructions
#3: doesn't change instruction length of 32 bits (4 byte increments could allow for lower 2 bits to be discarded)

(d) Based on (c), the `JALR` instruction could theoretically also discard `imm[0]`. Select all statements below that would be TRUE if a modified `JALR` also discarded `imm[0]`.

- ☐ `imm[12]` would become redundant based on part (b).
- ☐ This modified `JALR` could be encoded as a B-type instruction.
- ☐ The jump target address range would be expanded by 2× without being invalid.
- ☐ The jump target address of this modified `JALR` alone would have the same range as regular `JAL`, which also discards `imm[0]`.

Solution:

Choices #1 and #3
#2 would have useless `rs2` field that takes away `imm` bits
#4 cannot because `JAL` has much larger range (J-type inst.)

(e) Below is the encoding of the "C" (compressed, 16-bit instruction) RISC-V extension. Select all statements below that are TRUE based on the encoding of the fields across the formats.

<table>
<tr><th>Format</th><th>Meaning</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>CR</td><td>Register</td><td colspan="4">funct4</td><td colspan="5">rd/rs1</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CI</td><td>Immediate</td><td colspan="3">funct3</td><td>imm</td><td colspan="5">rd/rs1</td><td colspan="5">imm</td><td colspan="2">op</td></tr>
<tr><td>CSS</td><td>Stack-relative Store</td><td colspan="3">funct3</td><td colspan="6">imm</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CIW</td><td>Wide Immediate</td><td colspan="3">funct3</td><td colspan="8">imm</td><td colspan="3">rd′</td><td colspan="2">op</td></tr>
<tr><td>CL</td><td>Load</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1′</td><td colspan="2">imm</td><td colspan="3">rd′</td><td colspan="2">op</td></tr>
<tr><td>CS</td><td>Store</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1′</td><td colspan="2">imm</td><td colspan="3">rs2′</td><td colspan="2">op</td></tr>
<tr><td>CA</td><td>Arithmetic</td><td colspan="6">funct6</td><td colspan="3">rd′/rs1′</td><td colspan="2">funct2</td><td colspan="3">rs2′</td><td colspan="2">op</td></tr>
<tr><td>CB</td><td>Branch/Arithmetic</td><td colspan="3">funct3</td><td colspan="3">offset</td><td colspan="3">rd′/rs1′</td><td colspan="5">offset</td><td colspan="2">op</td></tr>
<tr><td>CJ</td><td>Jump</td><td colspan="3">funct3</td><td colspan="11">jump target</td><td colspan="2">op</td></tr>
</table>

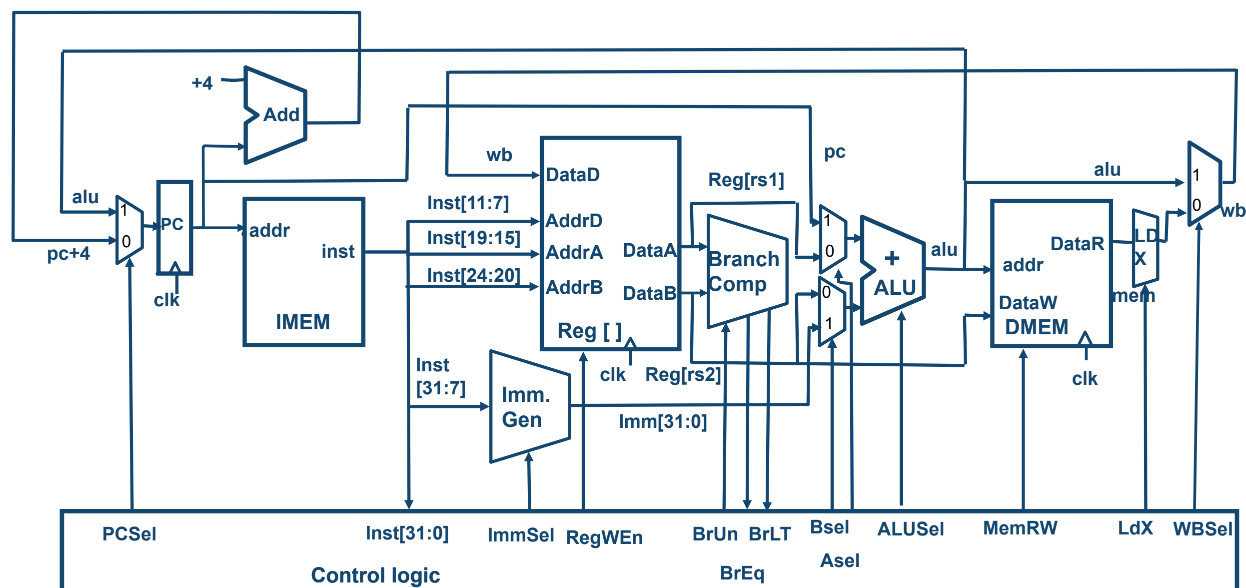
- [ ] An arithmetic instruction result must overwrite one operand register in the register file.
- [ ] Only 8 integer registers are available for all compressed instructions.
- [ ] Jumping to absolute/relative 32-bit addresses is not possible with 2 compressed instructions like in part (b).
- [ ] Branching instructions (CB format) can only compare against 0, not between 2 registers.

Solution:

Choices #1, #3, and #4
#2 is false for CR, CI, and CSS format compressed instructions

Problem 5: Datapathology [16 points, 15 minutes]

You may refer to the RISC-V Green Card on the last pages of this exam.



The single-cycle datapath above implements a subset of the RV32I instruction set.

(a) **Datapath functionality:** The Verilog code for an incomplete ALU is given below:

```
wire signed [31:0] in1s, in2s;
assign in1s = in1;
assign in2s = in2;
always @(*) begin
    case (ALUSel)
        ADD:         alu = in1 + in2;
        SUB:         alu = in1 - in2;
        SHIFT_LEFT:  alu = in1 << in2[4:0];
        LESS_THAN_S: alu = (in1s < in2s) ? 32'b1 : 32'b0;
        SHIFT_RIGHT: alu = in1 >> in2[4:0];
        OR:          alu = in1 | in2;
        AND:         alu = in1 & in2;
        PASS:        alu = in2;
    endcase
end
```

Select all instructions below that are supported by the given datapath diagram and the given ALU module:

☐ `LUI rd, imm`

☐ `AUIPC rd, imm`

☐ `BLT rs1, rs2, imm`

- ☐ `JALR rd, rs1, imm`
- ☐ `LW rd, rs1, imm`
- ☐ `SLTU rd, rs1, rs2`
- ☐ `SRL rd, rs1, rs2`
- ☐ `SRA rd, rs1, rs2`
- ☐ `XOR rd, rs1, rs2`
- ☐ `AND rd, rs1, rs2`

Solution:

`LUI, AUIPC, BLT, LW, SRL, AND`
There's no writeback datapath for current address in JALR.
SLT is supported but not SLTU.
SRL is supported but not SRA.
There's no XOR in ALU case.

(b) **New instruction:** In homework 4, we implement ReLU (defined as $y = max(0, x)$) as an R-type instruction:

```
relu rd, rs1, rs2
```

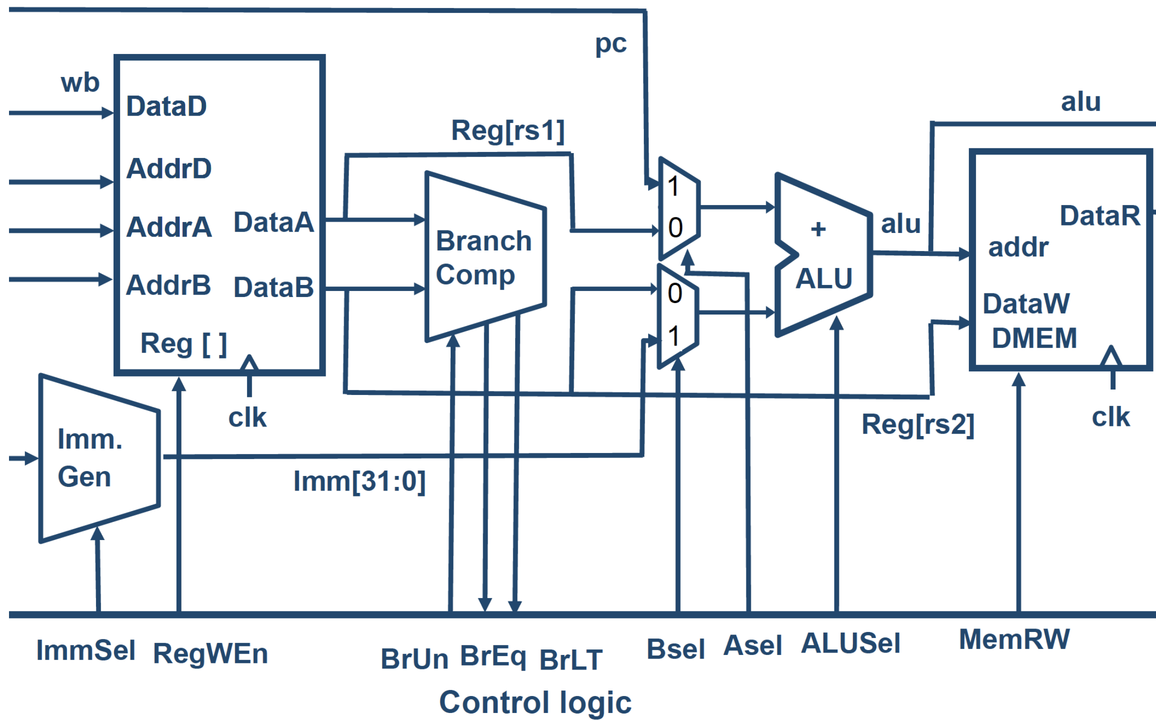
where `rs2` is a constant 0 (register `x0`). We use the branch comparator output to extend the control logic while keeping the ALU in the datapath untouched:

```
// Part of the control logic module
always @(*) begin
    ALUSel = ADD; // default
    if (opcode == OP || opcode == OP-IMM) begin
    // general instructions
    end
    else if (opcode == CUSTOM-0) begin
        ALUSel = BrLT ? AND : OR;  // x and 0 = 0, x or 0 = x
    end
end
```

Based on this design, now we would like to implement a new R-type instruction, noisy ReLU. Noisy ReLU includes Gaussian noise: $y = max(0, x + x_{noise})$. Assume register `x1` stores x, register `x2` stores x_{noise}, and we want `x3` to have y. `noisyrelu x3, x1, x2` is equivalent to the following instructions:

```
add x4, x1, x2
relu x3, x4, x0
```

Your Task: How would you modify the datapath to accommodate this instruction? Please draw on the datapath below and label any new control signal you want to use. Try to add as little hardware as possible. Available components (muxes, adders, constants, logic gates) are given below the datapath. Please also explain your design briefly. (If you are doing this on a blank paper, you only need to draw the necessary surroundings so that we can understand you)



Available components:

Solution:

```
Branch_Comp_input_1 = (NoisyReLU) ? DataA + DataB : DataA;
Branch_Comp_input_2 = (NoisyReLU) ? 32'b0         : DataB;
```

These signals also go to the input of ALU.

(c) **Timing:** The write ports of `Register File (RF)` and `Data Memory (DMEM)` are synchronized. Your teammate suggests two timing schemes:

(1) Both writing ports are triggered by the negative edge of clock.

(2) Both writing ports are triggered by the positive edge of clock.

`PC` is still updated at the rising edge of clock for both. Will they work properly? Which method do you prefer? Please explain the reason.

Solution:

Both work. For (1), `IF, ID, EX` have to fit in half clock cycle, while (2) can use the full clock cycle. For (2), `MA` for save instructions and `WB` actually happen in the next cycle, but asynchronous read can guarantee the correctness. Option (2) can clock faster and is preferred.

Spare page. Will not be graded. Feel free to tear off and use for scratch work.

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

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

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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

© 2018 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*

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

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

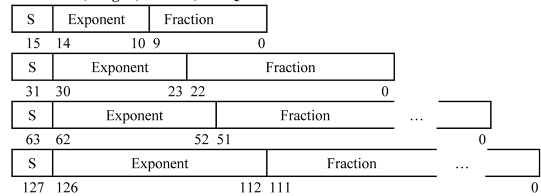
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return 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

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

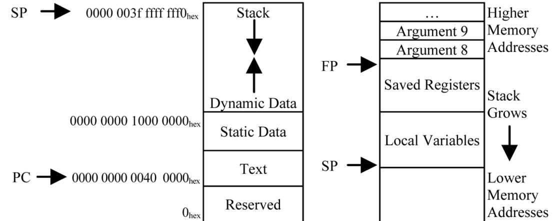
where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

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

© 2018 by Elsevier, Inc. All rights reserved. From Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition*

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

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