151 251A

Your Name (first last) Circle One SID

EECS 151/251A Fall 2020 Midterm 2

November 10, 2020

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

Exam Notes:

Before 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.

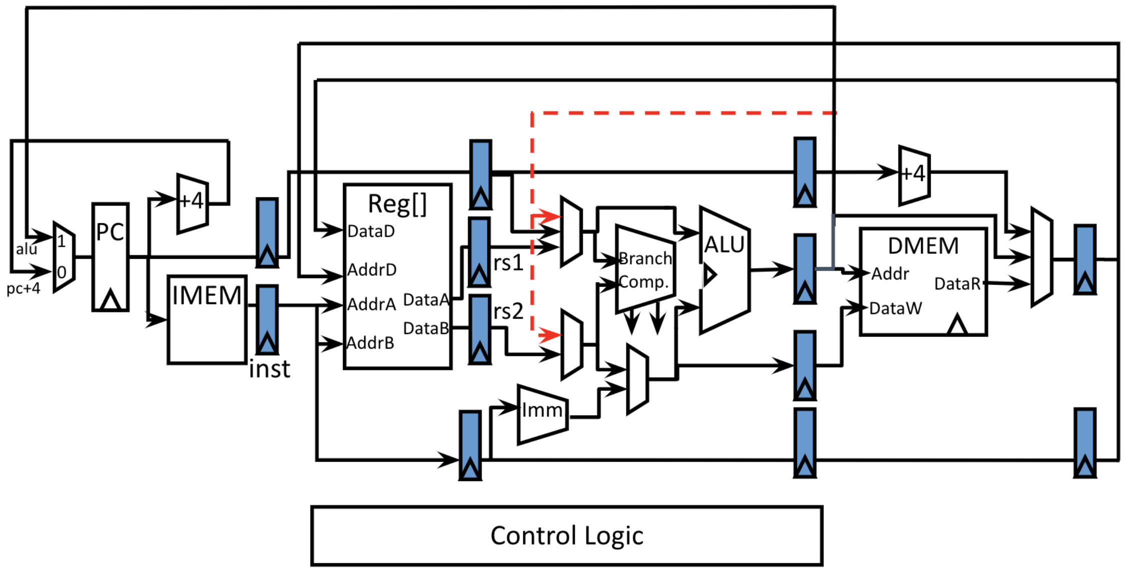
You have 80 minutes to work, starting at 3:40pm PST and ending at 5:00pm PST.

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 clarifications and errata
6. Summary of exam steps

Problem 1: Pipelining [15 points, 15 minutes]

Consider the in-order, single-issue 5-stage pipeline presented in lecture with combinational-read IMEM and DMEM. Assume we write to the register file in the first half of the clock cycle and read in the second half of the clock cycle (asynchronous write, asynchronous read), so we can read updated register values in the same cycle. In the diagram, we include forwarding paths (dashed lines) to eliminate some stalls.



Consider the following program. Assume that at the beginning of the program, R[a0] = 0x151515100, and that the integer 0x10 = 16 is stored at memory address 0x15151500.

(Clarification during exam: R[a0] = 0x15151500, not 0x151515100.)

```
        lw t0, 0(a0) # R[a0] = 0x15151500, Mem[R[a0]] = 0x10
        lw t2, 0(t0)
        slli t1, t0, 4
        sw t1, 0(a0)
        beq a0, t1, Label
        add a1, t2, t3
Label: ...
```

(a) When we run this program on the CPU, how many stalls (NOPs) will be inserted between each pair of instructions? Count the number of stalls without the forwarding paths and with the forwarding paths. 1 → 2 means between the instructions on lines 1 and 2.

You may use the pipeline tables in the Appendix to help figure out the number of stalls, but only the table below will be graded.

Instructions	No. of stalls (no forwarding)	No. of stalls (with forwarding)
1 → 2		
2 → 3		
3 → 4		
4 → 5		
5 → 6		

Solution:

No forwarding:

1 → 2: 2. Instruction 2 can only exit the instruction decode/register read stage, it takes 2 cycles before instruction 1 starts writing the new value of t0 in the register file, which can then be read out by instruction 2 in the same cycle.

2 → 3: 0. Instruction 3 does not have any dependencies on instruction 2.

3 → 4: 2. Same reason as 1 → 2.

4 → 5: 0. t1 already has the correct value by the time instruction 3 gets to the instruction decode/register read stage.

5 → 6: 2. The branch will be resolved in the execute stage, and at that point the next PC (PC+4) is immediately available. The instruction fetch, and instruction decode stages will be filled with NOPs in the meantime.

With forwarding: 1 → 2: 2. Does not change, since the value read out of DMEM doesn't get forwarded.

2 → 3: 0.

3 → 4: 0. The result of the `slli`, in the memory stage, will be available to `sw`, in the execute stage, through the forwarding path.

4 → 5: 1. Despite the fact that `t1` was immediately available to instruction 4, it is not immediately available to instruction 5, since there is no forwarding path from the writeback to execute stage. So, instruction 5 must stall for one cycle in the decode stage, inserting a NOP into the execute stage while instruction 3 is writing back.

5 → 6: 2. The forwarding path does not affect control hazards.

(b) Evaluate the following statements as true (T) or false (F).

i. _____ In the unforwarded datapath, if DMEM loads and stores took 3 cycles rather than 1, the number of cycles stalled between instructions 2 and 3 would increase.

Solution:

False. Instruction 3 has no dependency on instruction 2, so there would be no new NOPs inserted between the two instructions.

ii. _____ In the unforwarded datapath, if we moved the branch comparator to the decode stage, the number of cycles stalled between instructions 5 and 6 would decrease.

Solution:

True. Since the branch is not taken, we would now only have to insert a NOP in the instruction fetch stage, as the branch is resolved in the decode stage.

iii. _____ In the unforwarded datapath, combining the instruction fetch and instruction decode stages into one, resulting in a 4-stage pipeline, would eliminate some control hazards (i.e. it would fully remove the need for stalling in some cases).

Solution:

False. Branches still have to stall at least one cycle, and jumps must wait until the memory stage to get the PC for the next instruction.

iv. _____ Generally, pipelining increases the time it takes to execute a single instruction.

Solution:

True. First, the pipeline registers introduce overhead in the form of setup time and hold time that increase the time it takes for an instruction to propagate from PC to writeback. Second, it is unlikely that the pipeline stages are perfectly balanced. So, a 5-stage pipeline will have more than $\frac{1}{5}$ of the cycle time of an equivalent single-cycle processor. Pipelining does decrease the average amount of time taken per instruction, since it allows multiple parts of the processor to work in parallel. However, the question asks about a single instruction, not an average over many instructions. This illustrates the difference between latency and throughput.

Problem 2: CMOS Gate [15 points, 15 minutes]

In this problem, use the process that has $W_n = W_p = 1$ in a reference inverter, and $\gamma = 1$.

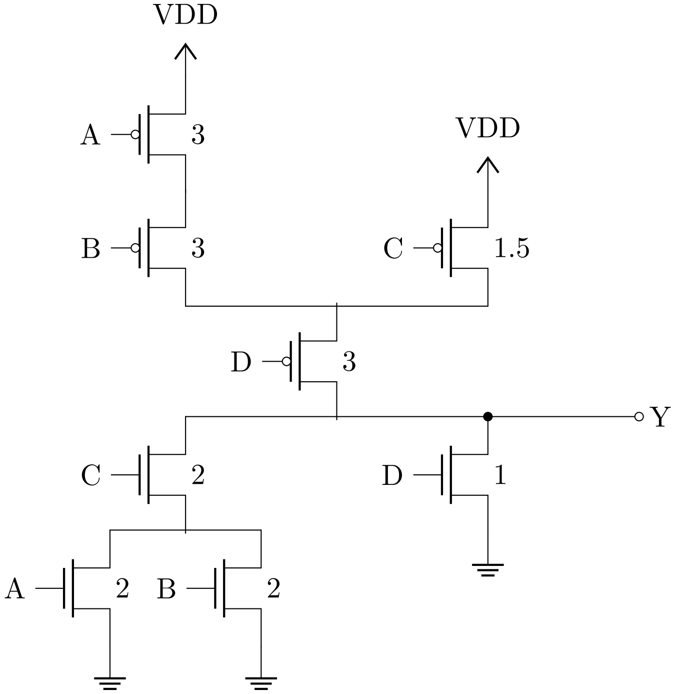
(a) Design a complex CMOS gate: $Y = \overline{(A + B) \cdot C + D}$. Size the transistors so that the gate has the same pull-up and pull-down strength as a reference inverter.

251A only: Also make sure the parasitic delay (P) is minimized in your design.

(b) Calculate the logic effort for each input (LE_A, LE_B, LE_C, LE_D).

(c) **251A only:** Calculate the optimized parasitic delay P.

Solution:



Sizing from worst-case path will guarantee the minimum overall area. In PMOS, A-B-D is the worst case and should be sized to 3. In NMOS, A/B-C is the worst case and should be sized to 2. The overall area (width) for PMOS is: $3 + 3 + 3 + 1.5 = 10.5$.

$$LE_A = LE_B = \frac{3+2}{2} = 2.5$$
$$LE_C = \frac{1.5+2}{2} = 1.75$$
$$LE_D = \frac{3+1}{2} = 2$$
$$P = \frac{3+2+1}{2} = 3$$

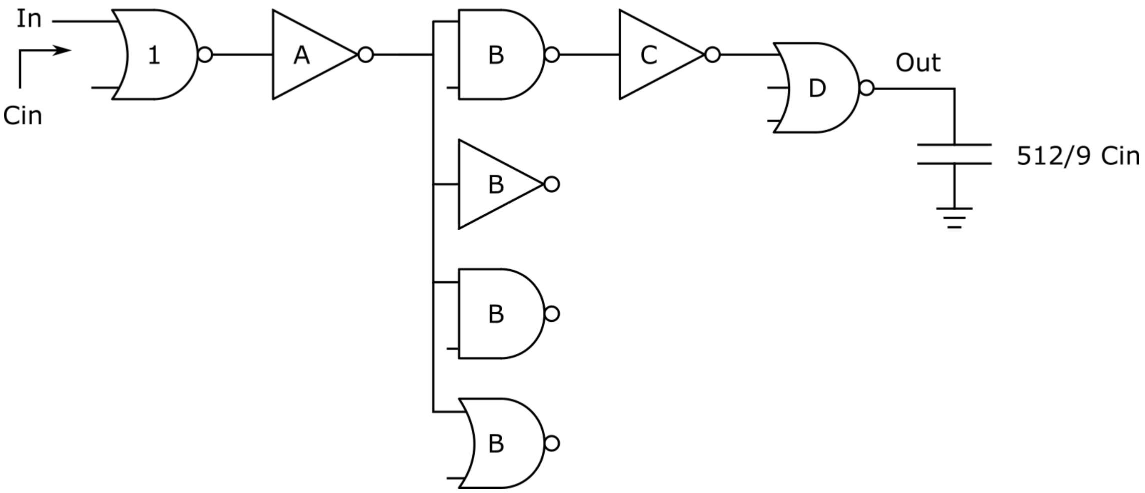
To minimize parasitic delay, we need to reduce the total parasitic capacitance at the output node. Moving smaller sized transistors in serial to the output can help. For example, in the schematic above, if NMOS A//B switch with NMOS C, the parasitic delay will be $P = (3 + 2 + 2 + 1)/2 = 4$ instead.

We didn't require the minimum area in the description so other solutions will also be taken. For example, in PMOS network, if you size C and D to 2, then A in serial with B should have the same resistance with C, so $A = B = 4$ $(1/4 + 1/4 = 1/2)$. The overall area for PMOS is: $4 + 4 + 2 + 2 = 12 > 10.5$.

As long as the logic effort calculation matches your schematic, you'll also get full credit in the second (and third) part.

Problem 3: Path Delay [15 points, 15 minutes]

Size the gates in the circuit below for minimum path delay. The circuit is implemented in a process where $R_n = R_p$ and $\gamma = 1$. Gates sized as '1' have an input capacitance of 1. For the NOR3 gate, use $LE = 2, P = 3$. You may leave your answers as fractions. Show your work. *Hint:* $2^{10} = 1024$



A =

B =

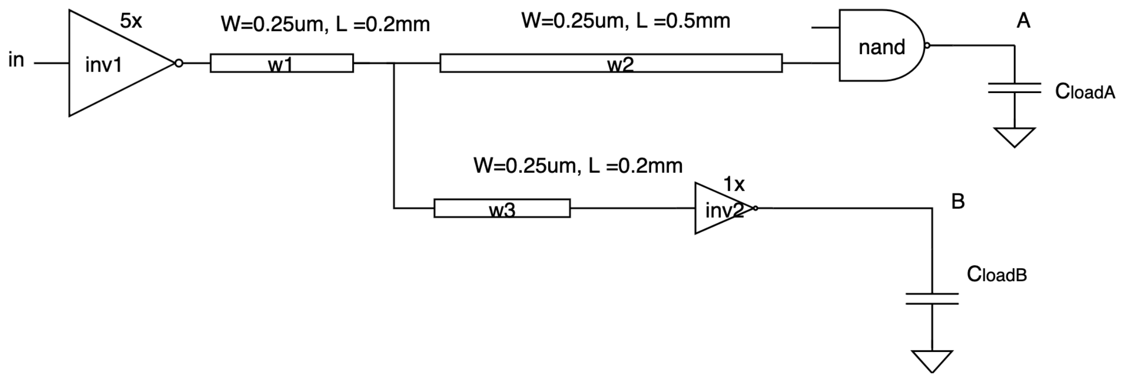
C =

D =

Solution:

$$\begin{aligned}
G &= \tfrac{3}{2} * 1 * \tfrac{3}{2} * 1 * 2 &&= \tfrac{9}{2} \\
B &= 1 * 1 * 4 * 1 * 1 &&= 4 \\
F &= \tfrac{512}{9} && \\
H &= \tfrac{9}{2} * 4 * \tfrac{512}{9} &&= 1024 \\
SE &= \sqrt[5]{1024} &&= 4 \\
D &= \tfrac{512}{9} * 2 * 1/4 &&= \tfrac{256}{9} \\
C &= \tfrac{256}{9} * 1 * 1/4 &&= \tfrac{64}{9} \\
B &= \tfrac{64}{9} * \tfrac{3}{2} * 1/4 &&= \tfrac{8}{3} \\
A &= \tfrac{8}{3} * 1 * 4/4 &&= \tfrac{8}{3} \\
1 &= \tfrac{8}{3} * \tfrac{3}{2} * 1/4 &&= \tfrac{1}{1}
\end{aligned}$$

Problem 4: Elmore Delay [20 points, 20 minutes]



(a) Find the resitance and total capacitance of wire 2 (`w2`) given the parameters below:

- wire resistance $R_w = 0.2\Omega/\square$
- parallel plate capacitance $C_{pp} = 40aF/(\mu m)^2$
- fringing capacitance per each side of wire $C_{fr} = 18aF/\mu m$

You may refer to the table of SI prefixes in the Appendix of this exam.

Solution:

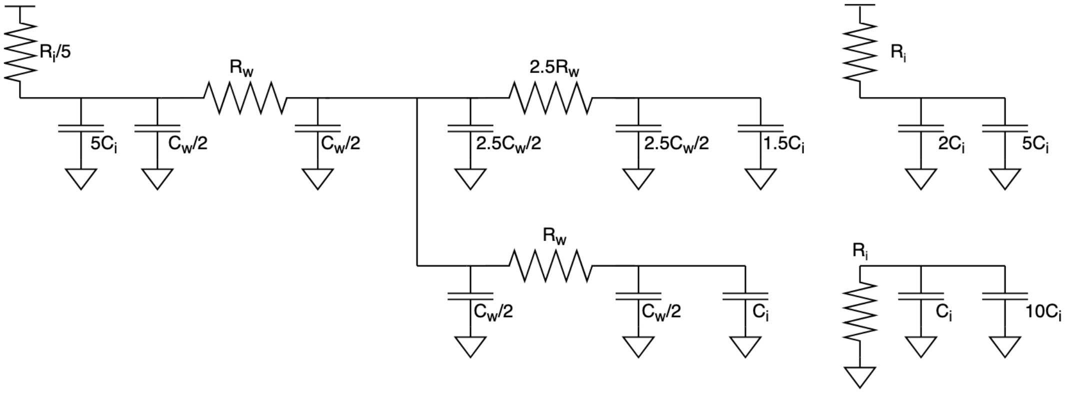
$R_{w2} = R_w \cdot \frac{L}{W} = 0.2\Omega/\square \cdot \frac{0.5mm}{0.25\mu m} = 0.2\Omega/\square \cdot \frac{500\mu m}{0.25\mu m} = 400\Omega$
$C_{w2} = C_{pp} \cdot L \cdot W + 2 \cdot C_{fr} \cdot L = 40aF/(\mu m)^2 \cdot 500um \cdot 0.25um + 2 \cdot 18aF/\mu m \cdot 500um = 23000aF = 23fF$

(b) Using the π wire model, draw the equivalent RC switch model. Label the values of resistors and capacitors using the assumptions below:

- wire 1 (`w1`) and wire 3 (`w3`) each have resistance R_w and total capacitance C_w
- 1x inverter has resistance R_i, input and parasitic capacitance $C_{i,1x} = C_{p,1x} = C_i$
- 5x inverter has 5 times the width of a 1x inverter
- nand gate has resistance R_i, input capacitance $1.5C_i$ and parasitic capacitance $2C_i$
- $C_{loadA} = 5C_i$ and $C_{loadB} = 10C_i$

Solution:

$Lw2 = 2.5L_{w1}$ so $R_{w2} = 2.5R_{w1} = 2.5R_w$ and $C_{w2} = 2.5C_{w1} = 2.5C_w$
The 5x inverter has resistance $R_{5x} = \frac{R_i}{5}$, parasitic capacitance $C_{p,5x} = 5C_{p,1x} = 5C_i$



(c) Find the delay from `in` to output `A`. Show your work and write your answer in terms of the parameters R_w, C_w, R_i and C_i.
(Hint: you can derive the resistance and total capacitance of wire 2 (`w2`) from those of wire 1 and wire 3)

Solution:

$$\begin{aligned}\tau = &\frac{R_i}{5}(5C_i + 2\cdot\frac{C_w}{2} + 2\cdot\frac{2.5C_w}{2} + 1.5C_i + 2\cdot\frac{C_w}{2} + C_i)\\ &+ R_w(\frac{C_w}{2} + 2\cdot\frac{2.5C_w}{2} + 1.5C_i + 2\cdot\frac{C_w}{2} + C_i)\\ &+ 2.5R_w(\frac{2.5C_w}{2} + 1.5C_i)\\ &+ R_i(2C_i + 5C_i)\\ &= \frac{R_i}{5}(7.5C_i + 4.5C_w) + R_w(2.5C_i + 4C_w) + 2.5R_w(1.5C_i + 1.25C_w) + R_i(7C_i)\\ &= 8.5R_iC_i + 0.9R_iC_w + 7.125R_wC_w + 6.25R_wC_i\end{aligned} \tag{1}$$

Problem 5: Power and Energy [15 points, 15 minutes]

151Laptops & Co. is consulting your power and energy analysis expertise for its next generation of processors. To begin, here are the specs for the current generation:

- Dual-core processor
- Capacitive load: 35nF per-core, 5nF between cores
- Clock frequency: 2GHz @ 1.0V
- Leakage power: 1W @ 1.0V per core

The benchmark workload generates the following specs:

- Per-core CPI: 2
- Max. activity factor: 0.1
- Parallelization factor: 80%

Operational Notes:

- For single-core instructions, assume the idle core has no switching activity (i.e. it is clock-gated) but the full capacitance between cores is still seen.
- For parallel instructions, the dual-core CPI is half that of single-core (perfect parallelization).

Finally, the following equations may be useful:

$$CPI = \frac{Clocks}{Instruction}$$

$$\frac{Seconds}{Program} = \frac{Instructions}{Program} * \frac{Cycles}{Instruction} * \frac{Seconds}{Cycle}$$

(a) Compute the maximum total power consumption for the current generation processor. Short-circuit current is negligible.

Solution:

Dynamic power for a dual-core instruction: $0.1 * 75nF * 1^2 * 2GHz =$ **15W**
Total power = dynamic + static power = **17W**

(b) If the benchmark workload has 1000 total instructions, how much energy does computing the entire workload consume?

Solution:

800 instructions are dual-core, 200 are single-core.
Dual-core CPI is 1, single-core is 2.
Therefore, 800 cycles are spent on dual-core instructions, 400 on single-core.
Single-core dynamic power $= 0.1 * 40nF * 1^2 * 2GHz =$ 8W → 10W total.
$E_{1000} = P_{dual} * t_{dual} + P_{single} * t_{single} = 17 * 800/2GHz + 10 * 400/2GHz = \mathbf{8.8\mu J}$

(c) Engineering wants to **reduce the energy consumption while maintaining the same benchmark compute time**, but only has the resources to implement one of the following:

Option #1: Move the existing design to the next (i.e. smaller) technology node.

Option #2: Design a power management unit (PMU) that power gates 1 core when not needed (i.e. for single-core instructions).

Option #3: Improve the processor's architecture (e.g. better branch predictor) to attain a 10% CPI improvement and correspondingly a 10% frequency reduction.

The predicted spec changes are summarized below. An — means unchanged or none.

Parameter	**Option #1**	**Option #2**	**Option #3**
Capacitance per core	-10%	—	+10%
Capacitance b/w cores	-10%	—	—
Supply voltage	-10%	—	-10%
Clock frequency	—	—	-10%
Leakage	+10%	—	+10%
Add'l. overhead	—	+0.1W†	—

† PMU static power consumption

Which option provides the greatest benchmark energy savings? Briefly explain. Numerical calculation should not be necessary!

Solution:

Option #1.

Option #1's dynamic power savings is on the order of $CV^2 = 0.9^3$.

Option #3's dynamic power savings is on the order of $V^2 = 0.9^2$ due to capacitance canceling frequency decrease.

Option #2 only cuts leakage power 20% of the time, so it is much worse because for this design, where dynamic power dominates over leakage power.

Note: all options maintain the total compute time.

(d) Evaluate the following statements as true (T) or false (F) based on your analysis.

______ All options would increase the processor die area.

______ Option #1 is leakier primarily because of a lower threshold voltage.

______ Only option #2's energy savings depends on workload parallelization.

Solution:

F (not option #1), T, F (all are)

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

Appendix

Table of SI Prefixes:

Prefix	Symbol	Magnitude
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^{9}
mega	M	10^{6}
kilo	k	10^{3}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}

Pipeline tables for Problem 1:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1. lw t0, 0(a0)	F	D	X	M	W															
2. lw t2, 0(t0)																				
3. slli t1, t0, 4																				
4. sw t1, 0(a0)																				
5. beq a0, t1, Label																				
6. add a1, t2, t3																				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1. lw t0, 0(a0)	F	D	X	M	W															
2. lw t2, 0(t0)																				
3. slli t1, t0, 4																				
4. sw t1, 0(a0)																				
5. beq a0, t1, Label																				
6. add a1, t2, t3																				

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

RISC-V Reference Data ①

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

ARITHMETIC CORE INSTRUCTION SET ②

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

<table>
<tr><th></th><th>31 27</th><th>26 25</th><th>24 20</th><th>19 15</th><th>14 12</th><th>11 7</th><th>6 0</th></tr>
<tr><td>R</td><td colspan="2">funct7</td><td>rs2</td><td>rs1</td><td>funct3</td><td>rd</td><td>Opcode</td></tr>
<tr><td>I</td><td colspan="3">imm[11:0]</td><td>rs1</td><td>funct3</td><td>rd</td><td>Opcode</td></tr>
<tr><td>S</td><td colspan="2">imm[11:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:0]</td><td>opcode</td></tr>
<tr><td>SB</td><td colspan="2">imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1|11]</td><td>opcode</td></tr>
<tr><td>U</td><td colspan="5">imm[31:12]</td><td>rd</td><td>opcode</td></tr>
<tr><td>UJ</td><td colspan="5">imm[20|10:1|11|19:12]</td><td>rd</td><td>opcode</td></tr>
</table>

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

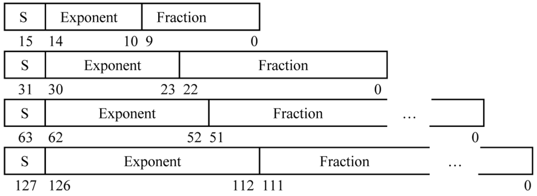
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return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

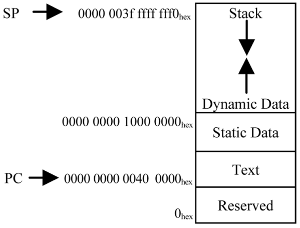
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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

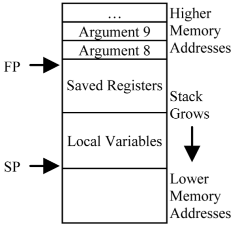
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



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

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