EECS151 EECS251A

Your Name (first last) Your Class (Circle one) SID

EECS 151/251A Fall 2020 Final

December 18, 2020

Question	1	2	3	4	5	6	7	8	9	10	**Total**
Sugg. time (mins)	10	10	22	20	20	20	20	24	24	10	**180**
151 Max. points	12	12	24	18	18	18	12	24	16	12	**166**
251A Max. points	12	12	24	18	18	18	18	24	24	12	**180**

Exam Notes:

The ten problems are NOT organized in the order of increasing difficulty. If you find yourself taking excessive time to work out a solution, consider skipping the problem and move on to the next one.

Before 6:50pm 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 180 minutes to work, starting at 7:00pm PST and ending at 10:00pm PST.

Please keep the Google Doc page that you received at 6:50pm 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: FSMs (Midterm 1 Clobber) [12 pts, 10 mins]

From your input in Midterm 2, 151Laptops & Co. has decided to use a 2-core processor in their next generation of laptops. Now they need your help designing the cache controller. Each core will have its own L1 cache, but both cores will share an L2. Specifically, you need to design an arbiter FSM that will take requests from each L1 cache and grant L2 access to one cache per cycle.
The details of the FSM's behavior are as follows:

- The FSM is a MEALY machine.
- The FSM has 2 bits of input, where the nth bit denotes a request from the nth core's L1.
  eg) an input of 2b'01 denotes a request from cache 0.
- The FSM has 2 bits of output, where the nth bit denotes a grant to the nth core's L1.
  eg) an output of 2b'10 denotes a grant to cache 1.
- Initially, the FSM should prioritize requests from cache 0.
- If there are no outstanding requests, the FSM should output 0.
- If there are outstanding requests the FSM must grant exactly 1 request.
- If there are multiple outstanding requests, the FSM should prioritize the cache with the least recent grant.

a) **Demonstrate your understanding:** Fill in the table with values of output given the sequence of requests.

Solution:

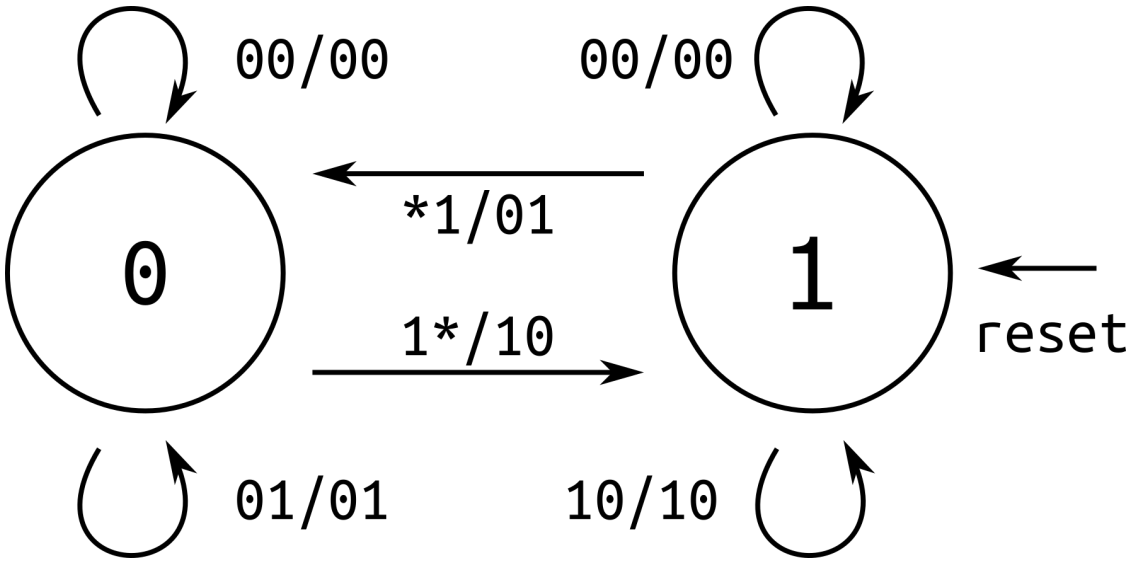
Cycle	Requests	Output
0	00	00
1	11	01
2	11	10
3	01	01
4	00	00
5	11	10

b) **Design it:** Draw the state-transition diagram for your Mealy machine. Indicate the initial state. You may use asterisks, with caution, to represent "don't care" values: an input of 2'b*1

indicates both 2'b01 and 2'b11. Let the state be a 1-bit value indicating the cache with the most recent grant.

State	0	1
Most recent grant	\$0	\$1

Solution:



c) **Boolean logic:** Write out the logic equation for each bit of output in product-of-sums form in terms of in[1:0] and state

out[0] =

out[1] =

Solution:

out[0] = $(in[0])(state + \overline{in[1]})$

out[1] = $(in[1])(\overline{state} + \overline{in[0]})$

Problem 2: Verilog (Midterm 1 Clobber) [12 pts, 10 mins]

Now that our cache controller is ready, let's build the CPU! We'll instantiate the modules `core`, containing a CPU core and its L1 cache, `fsm`, the FSM we just designed, and `l2_cache`, the unified L2 cache. We will implement the ability for each core's L1 cache to read in data from the L2 cache. Some additional details:

- The processor has 2 cores.
- Each core's L1 cache holds `req` high and `addr` to the requested address while a read request is outstanding. When the arbiter grants its request, the input `ack` should be set high.
- The L2 cache has a 1-cycle read latency. This means that if we set `rd_en` to `1` and `addr` to `0x10000000` in cycle 0, `data` has the data corresponding to memory address `0x10000000` in cycle 1.
- Each core also has an L1 cache write enable signal, `wr_en`. This should be asserted on the rising edge where the correct L2 read data is available. You can assume that the core knows the correct write address.
- We ignore memory writes (we only handle reads). We also ignore L2 cache misses.
- The `$clog2` function may come in handy.

On the next page, fill in the blanks to finish implementing the top level of the CPU.

```
module CPU_Top (
    input clk, rst
    );

wire [1:0] fsm_input;
wire [1:0] fsm_output;
wire [31:0] data;
wire [31:0] addr [1:0];

reg [1:0] seq_element;
always @( ___(1)___ ) begin
    if (rst) seq_element <= 0;
    else begin
        ___(2)___ ;
    end
end

genvar i;
generate for ( ___(3)___ ) begin: loop
    core gen_core (
        .clk(clk), .rst(rst),
        .wr_en( ___(4)___ ), // input
        .data(data), // input [31:0]
        .ack( ___(5)___ ), // input
        .req( ___(6)___ ), // output
        .addr(addr[i]) // output [31:0]
    );
endgenerate

fsm l1_l2_arbiter (
    .clk(clk), .rst(rst),
    .in(fsm_input),     // input [1:0]
    .out(fsm_output)    // output [1:0]
);

l2_cache l2 (
    .clk(clk), .rst(rst),
    .rd_en( ___(7)___ ),  // input
    .addr( ___(8)___ ),   // input [31:0]
    .data(data)           // output [31:0]
);

endmodule
```

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____

Solution:

```
module CPU_Top (
    input clk, rst
    );

wire [1:0] fsm_input;
wire [1:0] fsm_output;
wire [31:0] data;
wire [31:0] addr [1:0];

reg [1:0] seq_element;
always @(posedge clk) begin
    seq_element <= fsm_output;
end

genvar i;
generate for ( i = 0; i < 2; i = i+1 ) begin: loop
    core gen_core (
        .clk(clk), .rst(rst),
        .wr_en(seq_element[i] ), // input
        .data(data),             // input [31:0]
        .ack(fsm_output[i]),   // input
        .req(fsm_input[i]),      // output
        .addr(addr[i])           // output [31:0]
    );
endgenerate

fsm l1_l2_arbiter (
    .clk(clk), .rst(rst),
    .in(fsm_input),     // input [1:0]
    .out(fsm_output)    // output [1:0]
);

l2_cache l2 (
    .clk(clk), .rst(rst),
    .rd_en(fsm_output != 0),         // input
    .addr(addr[$clog2(fsm_output)]), // input [31:0]
    .data(data)                      // output [31:0]
);

endmodule
```

Problem 3: RISC-V (Midterm 1 Clobber) [24 pts, 22 mins]

a)

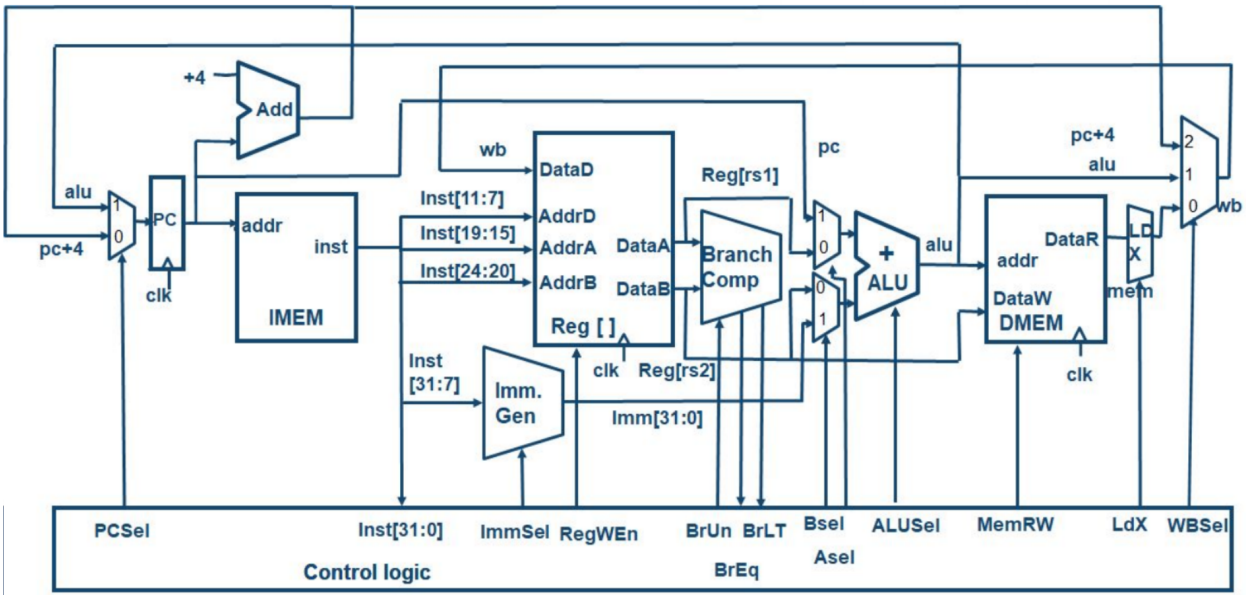


Figure 1: Correct single stage RISC-V datapath & control

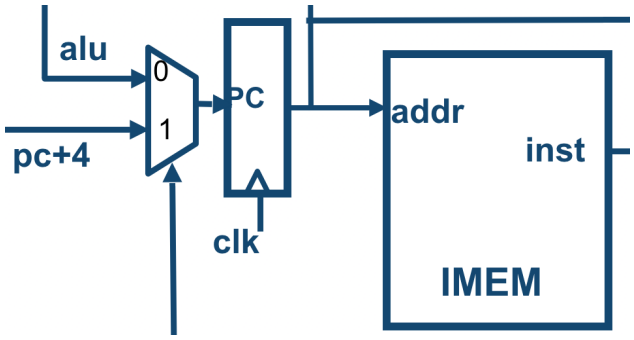


Figure 2: Buggy PC mux

After implementing the 2-core processor, we move on to testing it. Based on the testbench behaviors, we suspect that the `PCSel` mux has its 0 and 1 inputs switched.
Figure 1 shows the correct datapath behavior: `PCSel == 0` selects `pc + 4` and `PCSel == 1` selects the `alu` output.
Figure 2 shows the incorrect pc mux: `PCSel == 0` selects the `alu` output and `PCSel == 1` selects `pc + 4`.

Assuming the rest of the datapath and control are implemented correctly, and the PC mux has its inputs switched, step through the following assembly code. Fill in the table below. If `pc > 0x20`, write `pc = z` and stop.
Write down the values of the specified registers after the assembly code has been executed. All immediates are in decimal.

```
0x0  li x10, 4
0x4  addi x11, x10, 16
0x8  beq x10, x0, 8
0xc  sw x11, 40(x10)
0x10 li x12, 32
0x14 blt x10, x11, 8
0x18 addi, x11, x10, 4
0x1c slt x10, x10, x11
0x20 jal, x12, -4
```

cycle	1	2	3	4	5	6	7	8	9	10
pc	0x0									

`x10` = _____

`x11` = _____

`x12` = _____

b) Next, 151Laptops & Co. wants to add a `storeReLUN` instruction which stores the max of `x` (pre-loaded to `rs1`) and `n`(pre-loaded to `rs2`) to the address in `rd`:

```
storeReLUN rd, rs1, rs2: mem[R[rd]] = max(R[rs1], R[rs2])
```

To enable this instruction, we need extra hardware on the datapath. As shown in figure 3, an extra `DataRd` output port is added to the RegFile. A 2-input mux is added before the `DMEM addr` port, and a 2-input mux is added before the `DMEM DataW` port.
Write down each mux's inputs and control signal. You are allowed to use all signals in Figure 3, except for the `dataWSel` and `addrSel`, which you are asked to define. The opcode of `storeReLUN` is a parameter `CUSTOM-1`.

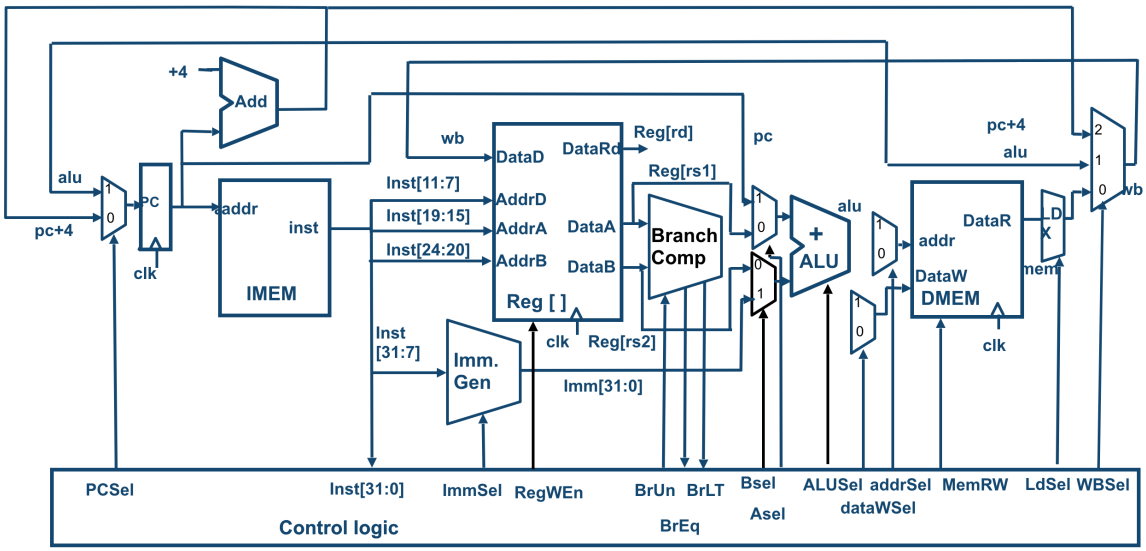


Figure 3: Modified single stage RISC-V datapath & control

`addr mux input 0` = _____
`addr mux input 1` = _____
`addrSel` = _____

`DataW mux input 0` = _____
`DataW mux input 1` = _____
`dataWSel` = _____

Solution:

a)

cycle	1	2	3	4	5	6	7	8	9	10
pc	0x0	0x4	0x14	0x18	0x8	0x10	0x20	z	z	z

`x10` = 4
`x11` = 8
`x12` = 36

b) `addr mux input 0` = alu
`addr mux input 1` = R[rd]
`addrSel` = inst[6:0] == CUSTOM-1

`DataW mux input 0` = R[rs2]
`DataW mux input 1` = R[rs1]
`dataWSel` = !BrLT && (inst[6:0] == CUSTOM-1)

Problem 4: Pipelining (Midterm 2 Clobber) [18 pts, 20 mins]

Consider a 4-stage pipeline as shown below. Both instruction memory and data memory are combinational read and write. The register file is also combinational, and read-after-write in the same cycle is permitted. Only consider the explicit forwarding path (dashed lines) in the diagram.

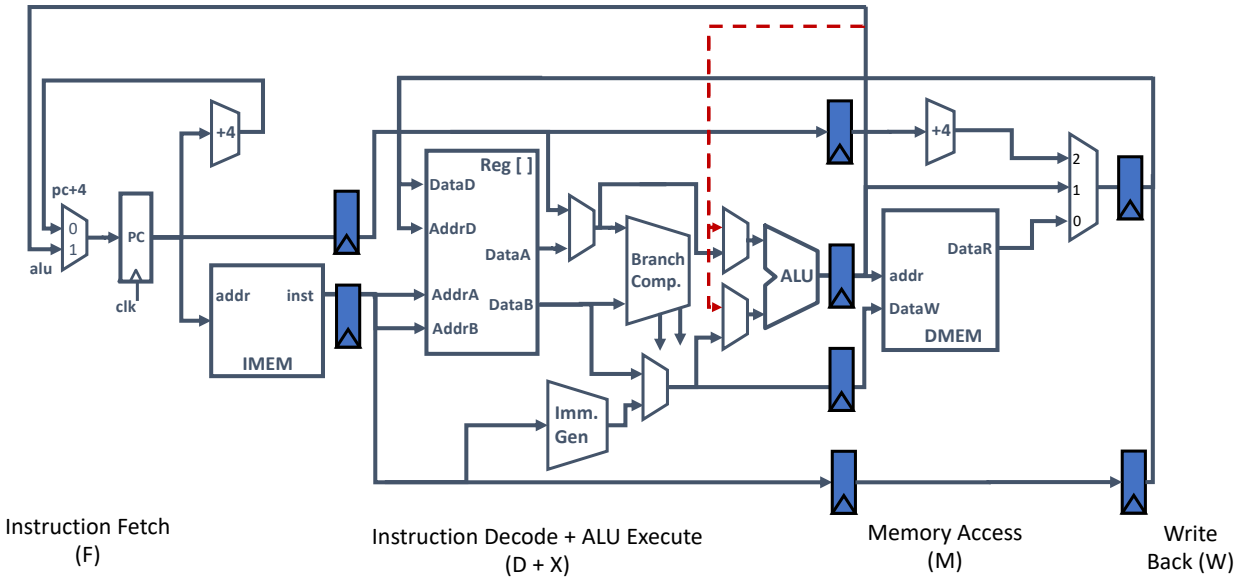


Figure 4: 4-stage pipeline with incomplete forwarding path

a) For each individual assembly code below, how many stalls (NOPs) will be inserted? No branching strategy is used in this part (always stall).

i) Number of stalls between 1 and 2: _____

```
1        add x3, x1, x2
2        and x4, x1, x3
```

ii) Number of stalls between 2 and 3: _____

```
1        add x3, x1, x2
2        xor x4, x1, x2
3        sub x5, x1, x3
```

iii) Number of stalls between 1 and 2: _____

```
1        add x3, x1, x2
2        blt x1, x3, Label1
```

iv) Number of stalls between 1 and 2: _____

```
1        lw x3, imm1(x1)
2        sw x2, imm1(x3)
```

v) Number of stalls after 1: _____

```
        jalr x3, x1, imm2 # R[x1] + imm2 -> Label2
        ...
Label2:addi x4, x3, 1
```

vi) Number of stalls after 1: _____

```
        bne x3, x1, Label3 # R[x3] = 2, R[x1] = 1
        ...
Label3:addi x4, x3, 1
```

Solution:

0; 0; 1; 1; 2; 2

i) This is handled by the forwarding path.

ii) $x3$ can be read after write in the D+X stage. No stall is needed.

iii) The inputs of branch comparator are not forwarded. Still need 1 stall.

iv) Need to wait 1 more cycle to get $x3$.

v) The address is connecting to the PC register, and will be available at the end of M stage.

vi) Same as above.

b) For each statement below, evaluate it as true (T) or false (F).

i) _____ If the register file is asynchronous read, synchronous write, we can remove the explicit registers between the memory access stage and the writeback stage, and the functionality remains the same as before.

ii) _____ For the existing forwarding path, forwarding to the output of register file (instead of the input of ALU) can help eliminate some stalls we identified in part a), without increasing the critical path.

iii) _____ If the critical path is located in the memory access stage, adding a forwarding path to solve memory-to-memory data hazard (e.g. sw after lw) will not increase the critical path.

iv) _____ If the critical path is located in the memory access stage, adding one more stage to form a 5-stage pipeline (F, D, X, M, W) can help increase the clock speed.

v) _____ For B-format instructions, if we assume branch always taken, we don't need extra hardware to avoid injecting stalls.

vi) _____ If a program takes time N by an 1-instruction per cycle datapath, it cannot be finished by an M-stage pipeline in time N/M, even if we have eliminated all stalls. Assume the maximum performance for both.

Solution:

T; T; F; F; F; T

i) This is equal to moving the registers "into" the register file.

ii) The new forwarding path can handle iii) in a), and the critical path will not be

longer than the second stage.

iii) The critical path will increase, since we are adding more components (ALU, muxes) to the critical path.

iv) Pipelining the non-critical path cannot improve the performance.

v) We need extra hardware in F stage to calculate the new address

vi) Unless each stage has exactly the same critical path, which is not possible in real world.

Problem 5: Path Delay (Midterm 2 Clobber) [18 pts, 20 mins]

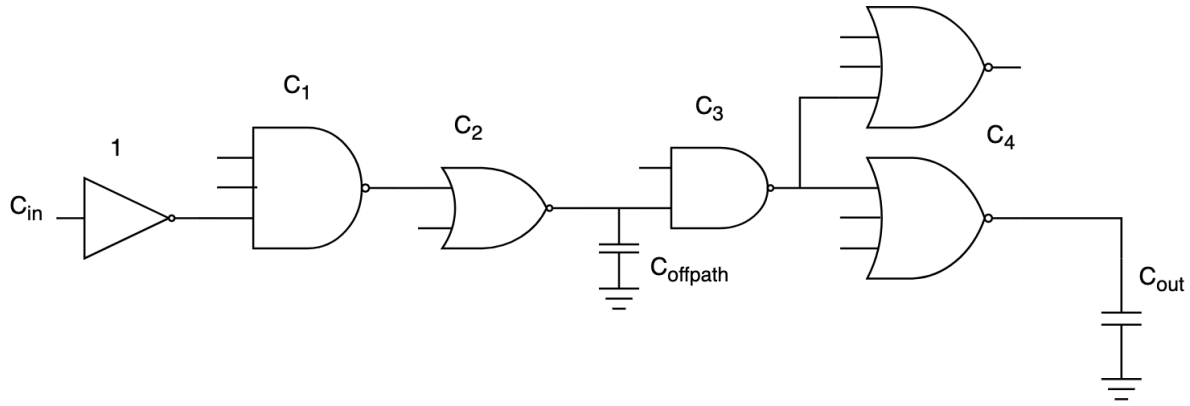


Figure 5: Path delay circuit

a) The circuit above is implemented in a process where $R_n = R_p$ and $\gamma = 1$. The inverter has an input capacitance of 1. $C_{out} = 9C_{in}$. $C_{offpath} = \frac{1}{2}C_3$. Size the gates using logical effort to minimize the path delay. Show your work.

b) What is the minimized path delay?

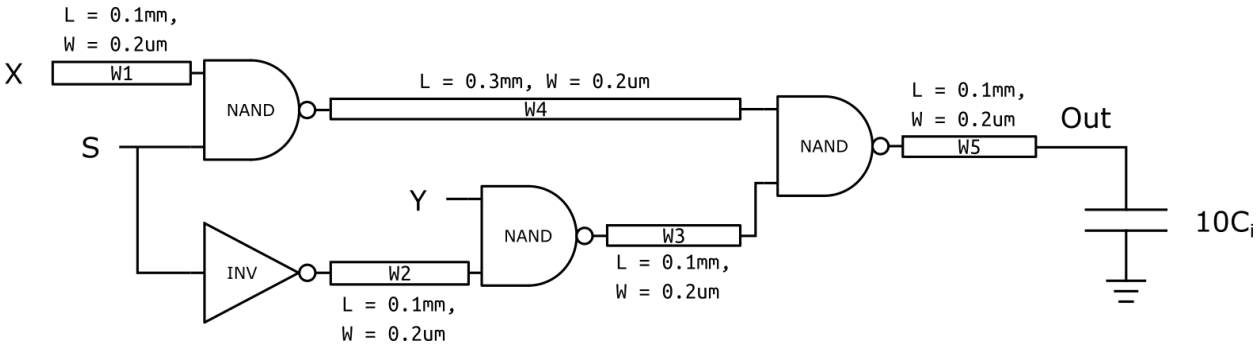
Solution:

a)
$G = 1 \cdot 2 \cdot \frac{3}{2} \cdot \frac{3}{2} \cdot 2 = 9$
$B = 1 \cdot 1 \cdot \frac{3}{2} \cdot 2 \cdot 1 = 3$
$F = 9$
$H = 9 \cdot 9 \cdot 3 = 243$
$EF = \sqrt[5]{243} = 3$
$C_4 = 9 \cdot \frac{1}{3} \cdot 2 = 6$
$C_3 = 2 \cdot 6 \cdot \frac{1}{3} \cdot \frac{3}{2} = 6$
$C_2 = \frac{3}{2} \cdot 6 \cdot \frac{1}{3} \cdot 2 = \frac{9}{2}$
$C_1 = \frac{9}{2} \cdot \frac{1}{3} \cdot 1 = 3$

b)
minimized path delay $= 5 \cdot EF + \Sigma p_i = 15 + (1 + 3 + 2 + 2 + 3) = 26$

Problem 6: Elmore Delay (Midterm 2 Clobber) [18 pts, 20 mins]

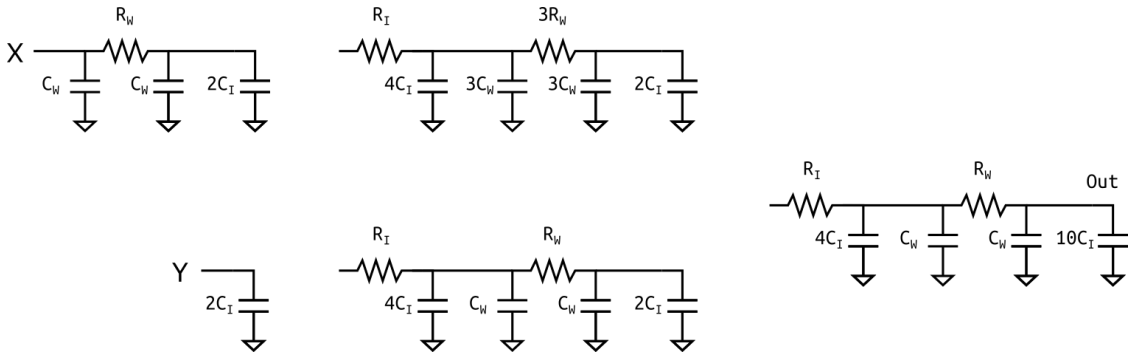
In this problem, we will analyze the delay of the following unidentified circuit.



a) Draw the equivalent RC switch model for the circuit in the figure above for signals X and Y (you may ignore S). Label the values of resistors and capacitors using the following assumptions:

- Wire 1 has a resistance of R_w and parasitic capacitance $2C_w$
- Inverters have input capacitance C_i, parasitic capacitance $2C_i$, and output resistance R_i
- NAND gates have input capacitance $2C_i$, parasitic capacitance $4C_i$, and output resistance R_i

Solution:



b) If S has been held at a value of 1 for a long time, what is the propagation delay from the inputs to the output? You may assume that X and Y arrive at the same time and are driven by sources with 0 time constant. *Hint: What is this circuit doing?*

Solution:

The mystery circuit is a 2-to-1 logic gate multiplexer. If we look at the logic of this circuit, when S is 1 the output will have the same value as X regardless of Y. So we are interested in the delay from X to output.

The signal from X travels through 3 sections.
$\tau_1 = R_w * (C_w + 2 * C_i)$
$\tau_2 = R_i * (4 * C_i + 3 * C_w) + (R_i + 3 * R_w)(3 * C_w + 2 * C_i)$
$\tau_3 = R_i * (4 * C_i + C_w) + (R_i + R_w)(C_w + 10 * C_i)$

$delay = ln(2) * (\tau_1 + \tau_2 + \tau_3) = ln(2)(11 * R_w C_w + 18 * R_w C_i + 8 * R_i C_w + 20 * R_i C_i)$

Problem 7: Arithmetic [12/18 pts, 20 mins]

Let's explore various ways to build an 8 × 8 bit unsigned multiplier. The following delays will be used in your delay expressions and are visualized below:

- t_{pp}: The delay of partial product generation (AND gate).
- t_{FA}: The delay of a full adder. For simplification, assume the carry and sum calculation have the same delay.
- t_{pg}: The delay of calculating the bitwise or group propagate and/or generate in a tree adder. Assume the delay is unaffected by fanout in a prefix tree.

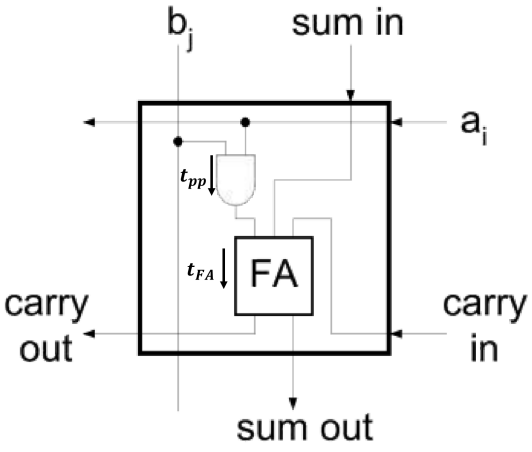


Figure 6: t_{pp} and t_{FA}

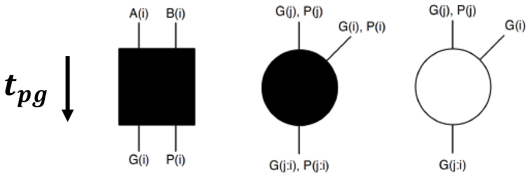


Figure 7: t_{pg}

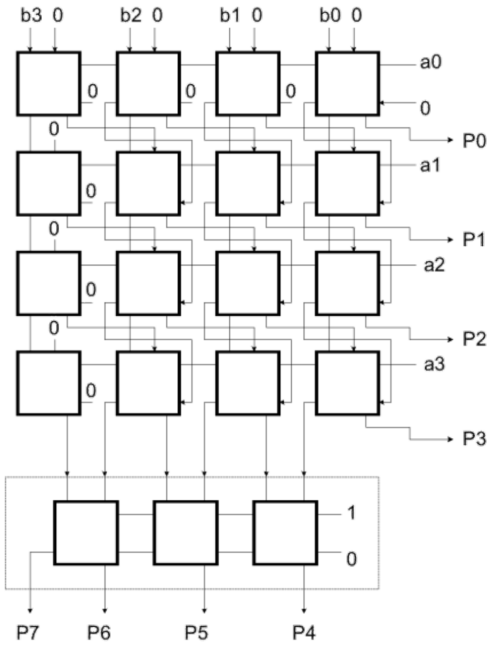


Figure 8: 4 x 4 CSA Array Multiplier

a) Let's start with a low-performance multiplier. Derive an expression the maximum delay of an 8 x 8 CSA array multiplier with a ripple-carry final adder. A 4 x 4 CSA array multiplier from lecture is shown in Fig. 8 for reference. If we pipeline the multiplier between the CSA array and final adder, which part has a longer critical path?

Solution:

The structure is a 8x8 CSA array followed by a 7-bit ripple-carry adder and the critical path is the carry rippling through the CSA array, then the carry rippling through the ripple-carry adder. The delay is: $t_{pp} + 8t_{FA} + 7t_{FA}$
If pipelined, the CSA array would have the longer critical path since an AND gate practically has less delay than a full adder.

It is also safe to assume that the full adder accepted P G as inputs, so solutions with a single t_{pg} added to the adder term are also accepted.

It is also correct to realize that the first row of an array multiplier can add the first 3 partial products together. This reduces the number of rows in the array by 2 to get $t_{pp} + 6t_{FA} + 7t_{FA}$. In this case, the ripple-carry adder has the longer critical path.

b) Carry-bypass adders significantly reduce delay compared to ripple-carry adders at the expense of just a bit more hardware. If we break up our ripple-carry final adder into a carry-bypass adder grouped by 4 bits, name the 2 types of logic gates that are added, a concise description of their function, and the quantity of each.

Solution:

A 7-bit carry-bypass adder broken into groups of 4 would have 2 groups, one with 4 bits and the other with 3 bits. This problem was primarily looking for these 2 gates:

- AND to calculate bypass = $\prod P_i$
- MUX to select generated carry or carry bypass

However, the problem did not state the assumption that the bitwise propagate/generate P_i and G_i was available in the full adder, so these are also valid gates that are added:

- XOR to calculate $P_i = A_i \oplus B_i$ (OR approximation is also valid: $P_i = A_i + B_i$)
- AND to calculate $G_i = A_i B_i$

Hence, credit is given for any two of these rows:

Logic gate	Function	Quantity
MUX	carry bypassing	2
AND	bypass = $\prod P_i$; $G_i = A_i B_i$	***see note
XOR (OR)	$P_i = A_i \oplus B_i$ ($P_i = A_i + B_i$)	7

***This could be any of:

- one 3-input AND + one 4-input AND for bypass (or 2 4-input)

- 5 2-input ANDs for bypass
- 7 2-input ANDs for bitwise generates
- 12 2-input ANDs for bypass + bitwise generates

c) **(251A only)** We can use a Wallace Tree and final parallel prefix tree adder for higher performance. A reference 4 x 4 Wallace Tree multiplier from lecture is shown below.

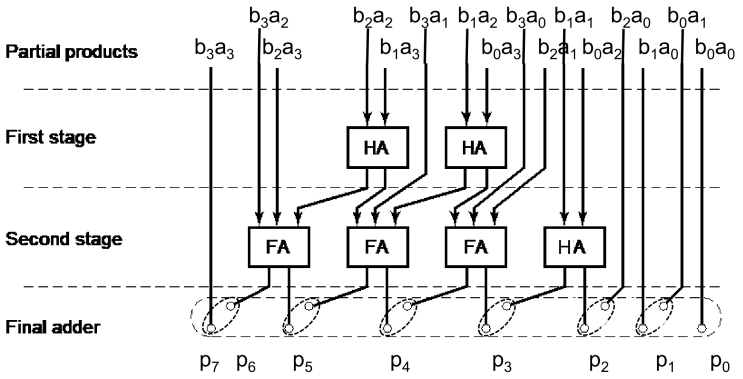


Figure 9: 4 x 4 Wallace Tree Multiplier

i) Derive an expression for the delay through an 8 x 8 Wallace Tree multiplier with 3:2 compression and a radix-2 Kogge-Stone final adder. You may leave the expression in terms of $log_{(...)}N$. Assume the parameter α for the Wallace tree as given in lecture is 1 and the half adder delay is equal to t_{FA}.

ii) If we use radix-4 Booth recoding, describe concisely how the overall multiplier area and delay changes.

iii) If we use a radix-4 Kogge-Stone final adder, describe concisely what the area/delay tradeoff is for group P/G calculation.

Solution:

i) Wallace tree: $t_{pp} + ceil(log_{3/2}N/2) \cdot t_{FA}$ where N = 8
Kogge-Stone: $t_{pg} + ceil(log_2(N-1)) \cdot t_{pg} + t_{FA}$ where N = 15
Total delay is the sum of the above terms.
Note: credit also given for a 16-bit final adder or $log_{3/2}8$ term for Wallace Tree since that was given in lecture.

ii) Radix-4 Booth recoding reduces the number of partial products by about 2 (down to $ceil(\frac{8+1}{2}) = 5$ partial products to be exact) with signed partial product accumulation. This reduces partial product HA/FA area, but incurs an area overhead from the recoding logic. This also reduces the delay of the Wallace Tree to be roughly equal to the final adder (reducing the critical path length in a pipelined case), even when factoring the delay overhead of recoding logic.

iii) A Radix-4 adder reduces the number of stages of group P/G calculation by a factor of 2, but each calculation block has larger delay because they take in 4 P/G groups as input.

Problem 8: Flip-Flop Timing [24 pts, 24 mins]

In this problem, you are asked to perform setup and hold timing analyses. Consider the circuit given in the diagram. Each flip-flop has a clock-to-q delay of $t_{clk-q} = 80ps$, setup time of $t_{su} = 40ps$, hold time of $t_h = 60ps$.
Note: you do **not** need to consider any specific instruction in this problem.

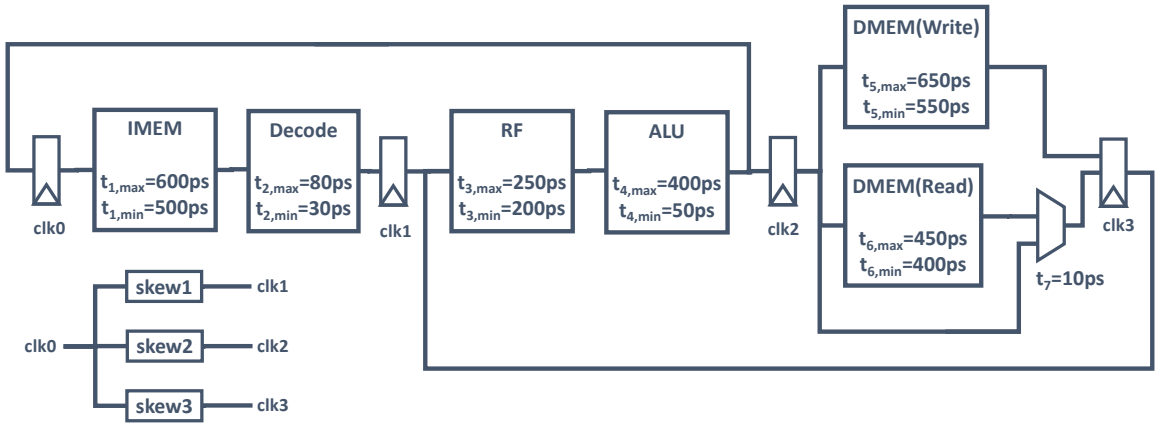


Figure 10: Circuit for setup/hold time analyses

a) Assume there is no skew and jitter between the clocks. What is the minimum clock period this circuit can operate with? Is there any hold time violation? Denote your hold time analysis in terms of hold slacks, where a negative slack would mean a violation.

$T_{clk} =$ $\quad ps$

$Hold\ Slack =$ $\quad ps$

Solution:

$$\begin{aligned} T_{clk} &> t_{clk-q} + t_{max} + t_{su} \\ T_{clk} &= 80ps + 680ps + 40ps = 800ps \\ Hold\ Slack &= t_{clk-q} + t_{crit,min} - t_h \\ &= 80ps + 10ps - 60ps \\ &= 30ps \end{aligned}$$

b) Now, if the circuit operates at $T_{clk} = 820ps$, and we have $t_{skew1} = 20ps$, $t_{skew2} = -10ps$, $t_{skew3} = 10ps$. Instead of being a certain value, the cycle-to-cycle t_{clk-q} of each flip-flop presents a random distribution between $70ps$ and $90ps$. Assume there is no clock jitter. Denote your timing analysis in terms of setup and hold slacks, where a negative slack would mean a violation.

$Setup\ Slack =$	ps
$Hold\ Slack =$	ps

Solution:

With some analyses (no need to show this work), you should find the critical path starts from clk1 and ends at clk2:

$$\begin{aligned} Setup\ Slack &= T_{clk} + t_{skew1,2} - (t_{clk-q,max} + t_{crit,max} + t_{su}) \\ &= 820ps - 30ps - (90ps + 650ps + 40ps) \\ &= 10ps \end{aligned}$$

Critical path for hold time starts from clk2 and ends at clk3:

$$\begin{aligned} Hold\ Slack &= t_{clk-q,min} + t_{crit,min} - t_{skew2,3} - t_h \\ &= 70ps + 10ps - 20ps - 60ps \\ &= 0ps \end{aligned}$$

c) If you are free to set the value of t_{skew1} and t_{skew2}, what value will you use so that the circuit can operate at minimum clock period without any violation? What is the optimum hold time slack under this clock period? (i.e. You should achieve the minimum clock period first, then try to maximize the hold time slack without increasing the clock period) Assume no clock jitter and use $t_{clk-q} = 80ps$ in this part.

$Skew1 =$	ps
$Skew2 =$	ps
$T_{clk} =$	ps
$Hold\ Slack =$	ps

Solution:

Since there's no skew between clk0 and clk3, the circuit actually has 3 loop boundaries:
1) clk0 - clk1 - clk2- clk3;
2) clk0 - clk1 - clk0;
3) clk2 - clk3 - clk2.
The circuit will be limited by the second one. Skew clk1 by 15ps to average 680ps and

650ps in loop 2). The clock period will be:

$$\begin{aligned} T_{clk} + t_{skew1,2} &> t_{clk-q} + t_{max0to1} + t_{su} \\ T_{clk} &= 80ps + 680ps + 40ps - 15ps \\ &= 785ps \end{aligned}$$

or

$$\begin{aligned} T_{clk} + t_{skew2,1} &> t_{clk-q} + t_{max1to0} + t_{su} \\ T_{clk} &= 80ps + 650ps + 40ps + 15ps \\ &= 785ps \end{aligned}$$

With $T_{clk} = 785ps$, $skew2$ can be set from $0ps$ to $15ps$ without any setup violation. However, a larger negative skew between clk2 and clk3 can favour the hold time slack. So we choose $skew2 = 15ps$. The resulted hold time slack is:

$$\begin{aligned} Hold\ Slack &= t_{clk-q} + t_{min2to3} - t_{skew2,3} - t_{hold} \\ &= 80ps + 10ps - (-15ps) - 60ps \\ &= 45ps \end{aligned}$$

We clarified during the exam in errata that you should use $t_{skew3} = 0$ for simplicity. However, if you are assuming $t_{skew3} = 10ps$ from part (b), you'll still get full credit for this part.

Problem 9: SRAMs and Decoders [16/24 pts, 24 mins]

a) Given the 6T SRAM shown below, evaluate the following statements as true (T) or false (F):

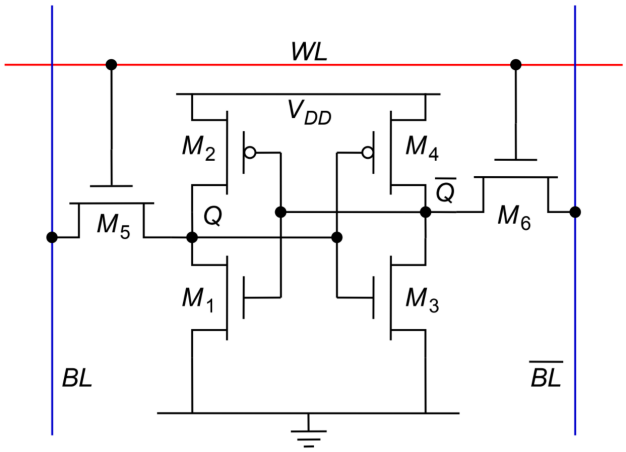


Figure 11: 6T SRAM

i) _____ This SRAM array can only support 1 read and 1 write port.

ii) _____ SRAM cells with more than 6 transistors will always support arrays with more than 1 read and/or write ports.

iii) _____ The bitline that stays high is the one primarily involved in flipping the cell state during a write operation.

iv) _____ In a FinFET implementation of a 6T SRAM, the ratio of $(W/L)_2 : (W/L)_5 : (W/L)_1$ can be 1:2:3 for good read stability and writability.

v) _____ In a 6T SRAM, circuit techniques that improve read stability inevitably hurt writability, and vice versa.

vi) _____ SRAM cell leakage degrades read access time.

Solution:

i) T, it cannot support more than 1 of each port.

ii) F, some are used to decouple read and write operations, others to improve power, etc. instead of enabling adding additional read/write ports.

iii) F, the BL that is pulled low flips the state through the access transistor. Recall that NMOS transistors can't pass a good '1'.

iv) T, $(W/L)_2 < (W/L)_5$ is necessary for writability. $(W/L)_5 < (W/L)_1$ is necessary for read stability. This is not to be confused with sizing (ratio of W's only), where there is a distinction between FinFET (1:2:3 due to equal P/N resistance) and planar

(1:2:2 due to 2x more PMOS resistance).

v) T, techniques include adjusting voltages of wordline, bitlines, or the latch pair. As shown in discussion, tweaking for read stability and writability are fundamentally opposing goals in a 6T SRAM. Decoupled read/write cells (e.g. 8T) do not have this tradeoff.

vi) T, the leakage of bitcells pulls both bitlines down simultaneously and unevenly, reducing the ability for the cell being read to generate a difference in bitline voltage as easily.

b) Consider an 256-word SRAM array where each word is 256 bits wide. The row decoding logic is placed to the left of the array, as shown in lecture. The array has the following properties:

- The 6T SRAM cell area is $0.2\mu m \times 0.2\mu m$.
- Access transistors have $C_g = C_d = 20aF$.
- The decoding scheme consists of 4-bit predecoders and final row decoders. The circuit model for each predecoder is shown below (Fig. 12).
- C_W models the wire capacitance between the predecoder and final decoders.
- C_{WL} models the total load on each final decoder.
- The wordline has capacitance per unit length of $0.1fF/\mu m$.
- In this technology, $R_p = R_n$ for a unit inverter and $\gamma = 1$.

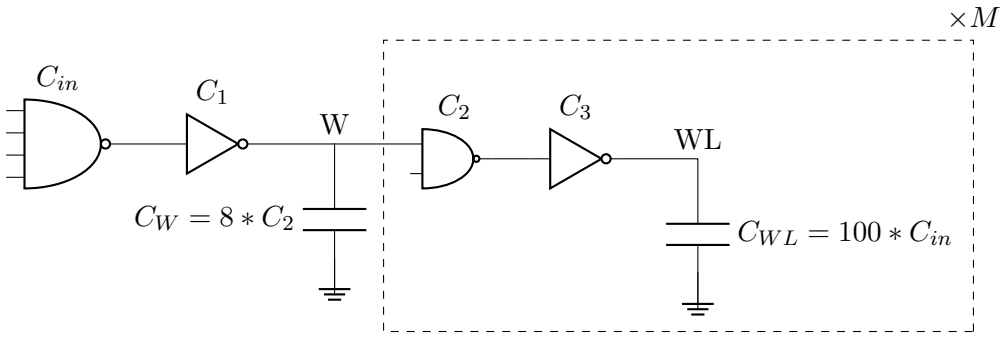


Figure 12: Row decoder model

Calculate:

i) the total number of final decoders each predecoder drives (i.e. the factor M in Fig. 12)

ii) the total capacitance per wordline

iii) the stage effort (you may leave this expression in terms of a root)

Solution:

i) $M = 256/2^4 = 16$

ii) Wordline capacitance comes from the wire and all of the gates of the access transistors.
$C_{WL} = 256 * 2 * 20aF + 256 * 0.2\mu m * 0.1fF/\mu m = 15.36fF$

iii) LE of 4-input NAND is 5/2 and 2-input NAND is 3/2. The branching factor at node W is 8 + M where M is 16 (above).

$$N = 4, F = 100, B = 24, G = 15/4$$
$$H = GFB = 9000$$
$$SE = \sqrt[N]{H} = \sqrt[4]{9000}$$

To check:

$$SE = \sqrt[4]{9000} = 9.74$$
$$C_3 = C_{WL}/SE = 1.577fF$$
$$C_2 = C_3/SE * 3/2 = 0.243fF$$
$$C_1 = C_2/SE * 24 = 0.598fF$$
$$C_{in} = C_1/SE * 5/2 = 0.1536fF = C_{WL}/100$$

c) **(251A only)** Now let's split the SRAM words into two halves and place the decode circuitry down the middle. The final decoder is split into two, each driving half of the word line. This new array decoding configuration is modeled in Fig. 13 and supposedly has a lower minimum decoding delay compared to Fig. 12, especially for SRAMs with large word sizes. Pay special attention to C_W – recall that it models a wire that spans the entire array height, which is unchanged from part b).

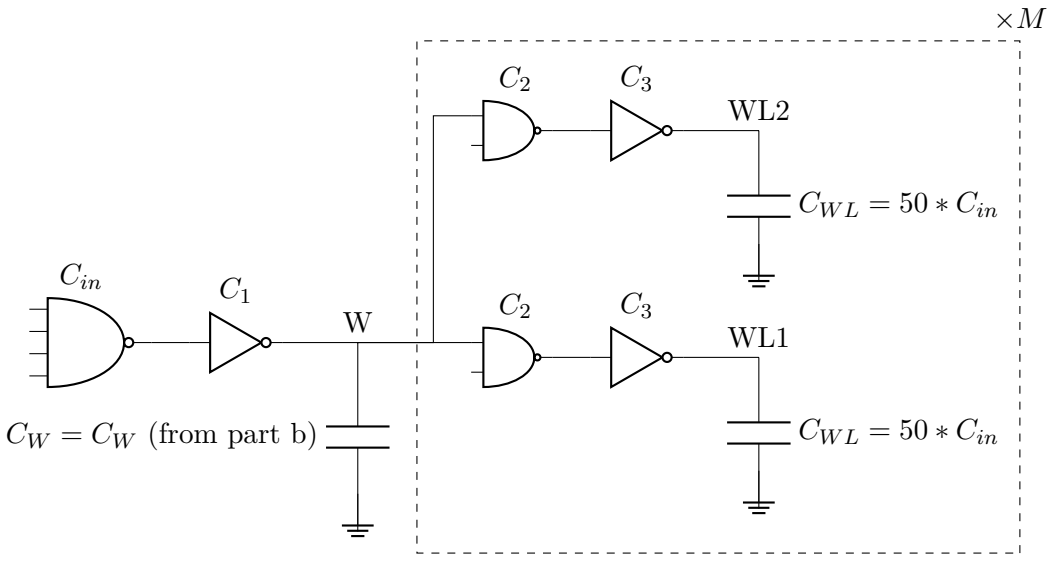


Figure 13: Split final decoder model

Your classmate analyzed this new circuit using the Path Delay method and found that its minimum delay is exactly the same as that of the circuit in Fig. 12. The only difference they found for minimum delay is that C_2 and C_3 are halved. Concisely explain why your classmate could not support the claim of lower delay and identify what was omitted (*hint*: should they analyze this differently?).

Solution:

It is important to first redo your classmate's work. It turns out they did the calculations flawlessly. Intuitively:

- The wordline capacitance is correctly halved because there are half the number of cells and half the wire length
- Note that C_W is the same value as what would be calculated in part b)
- Essentially, by halving the load on the final decoder but doubling the number of final decoders, halving C_2 and C_3 and keeping the same value of C_W means the branching factor at node W is unchanged
- As a result, all of the factors in path effort calculation (N, F, B, G) is the same as the original circuit, and hence the minimum calculated path delay is the same.

So, it turns out the contribution of wire resistance was omitted from their analysis. Since the wordline length is halved, its resistance is half of what it was before. When these circuits are analyzed as an Elmore delay problem, the RC time constant contributed by the wordline wire is reduced, which supports the claim of lower decoding delay.

Problem 10: Caches [12 pts, 10 mins]

a) A direct-mapped cache is 8KB in size, with 64B blocks. Memory addresses are 32 bits. In a memory access, how many address bits are used for:

i) The byte-select offset? _____

ii) The cache block index? _____

iii) The cache tag? _____

Solution:

Offset bits: 64-byte blocks = 2^6 bytes $\rightarrow$ 6 offset bits.

Index bits: Cache size is 8 KB = 2^{13} bytes

2^{13} B / 2^6 B/block = 2^7 blocks $\rightarrow$ 7 index bits

Tag bits: 32 - 6 - 7 = 19 tag bits

For parts b–d, consider the following program, written in pseudocode, that loops twice over an array of 1-byte numbers (for clarity, RISC-V assembly is also provided at the end of the problem). Assume N is very large and divisible by 32, and that `arr` starts at a memory address divisible by 32.

```
byte arr[N];

for (int j = 0; j < 2; j++) {
    for (int i = 0; i < N; i++) {
        process(arr[i]);
    }
}
```

b) Suppose we have an LRU (evict least recently used), 32-byte block, fully associative cache of size N bytes.

i) In terms of N, how many memory accesses are cache hits? _____

ii) Misses? _____

Solution:

In the first iteration, every 32 memory accesses, we get one compulsory miss. All the rest of the N memory accesses are cache hits. At this point, the entire array has been stored in the cache.

In the second iteration, all N memory accesses are cache hits.

Hits: $\frac{31}{32}$N + N = $\frac{63}{32}$N

Misses: $\frac{1}{32}$N

c) Suppose we have an LRU (evict least recently used), 32-byte block, fully associative cache of size N / 2 bytes.

i) In terms of N, how many memory accesses are cache hits? _____

ii) Misses? _____

Solution:

In the first iteration, the pattern is the same as for the cache of size N. Every 32 memory accesses, we get one compulsory miss. All the rest of the N memory accesses are hits. However, once the cache fills up, we evict the block we used least recently.

When we begin the second iteration, only the second half of the array can be found in the cache. So we still get 1 out of 32 misses. Then, once we reach the second half of the array, the cache has been filled with the first N/2 elements, so we continue to get 1 out of 32 misses.

So, we get 1 miss per 32 accesses for the entire 2N memory accesses in the program.

Hits: $\frac{31}{32} \times 2N = \frac{31}{16}N$

Misses: $\frac{1}{32} \times 2N = \frac{1}{16}N$

d) Suppose we take our LRU cache of size N / 2, and change its replacement policy to MRU, meaning that when we need to evict a cache block, we evict the most recently accessed block. For the given program, would this cache perform the same, better, or worse than its LRU counterpart? Why?

Solution:

Better. In the first iteration, the hit/miss pattern is the same as before. However, we get some cache hits in the first half of the array for the second iteration, so we get more hits and fewer misses than the LRU cache.

The reason for this is that in the first iteration, once we start accessing the second half of the array, rather than replacing the entire first half of the array we only replace the most recent 32-byte block, leaving the rest of the array in the cache. So, when we begin the second iteration, most of the first half of the array is in the cache, so every memory access is a cache hit.

For clarity, we provide RISC-V assembly equivalent to the pseudocode above:

```
    li t0, arr      # arr is the address where the array starts
    li t1, 2
    li t2, N        # N is a very large number
    li t3, 0        # t3 = j
Loop1:
    bge t3, t1, Loop1End
    li t4, 0        # t4 = i
Loop2:
    bge t4, t2, Loop2End
    add t5, t0, t4
    lb a0, 0(t5)
    ...             # process a0
    addi t4, t4, 1
    j Loop2
Loop2End:
    addi t3, t3, 1
    j Loop1
Loop1End:
```

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}

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

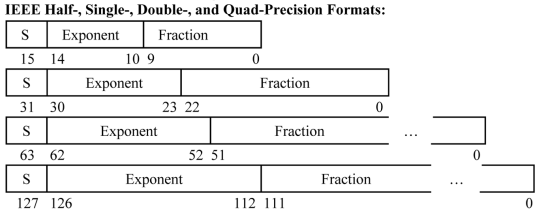
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return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

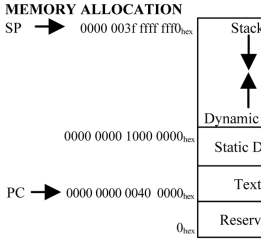
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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

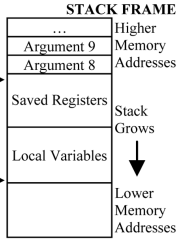
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



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

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