mp_verif

Welcome to ECE 411!

Welcome to ECE 411!

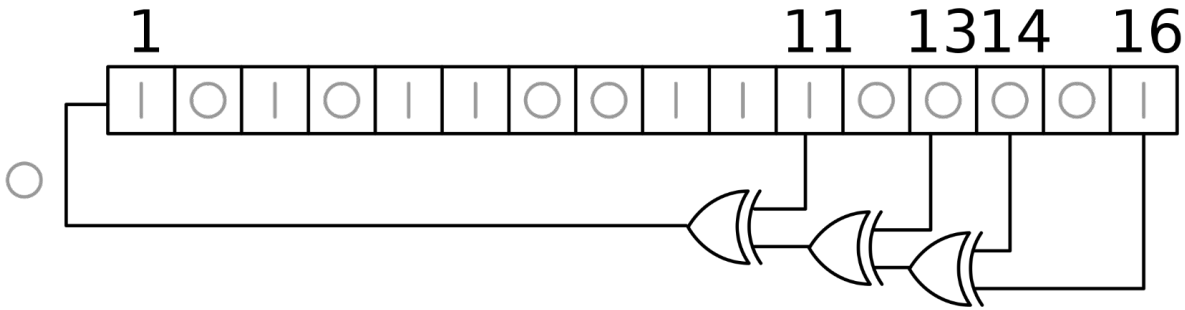
- 5 MPs:
  - mp_verif
  - mp_pipeline
  - mp_cache
  - mp_bp
  - mp_ooo (team)
- Lab session every week.
- Office hours in ECEB 2022 and DCL 440.

Autograder / Submission

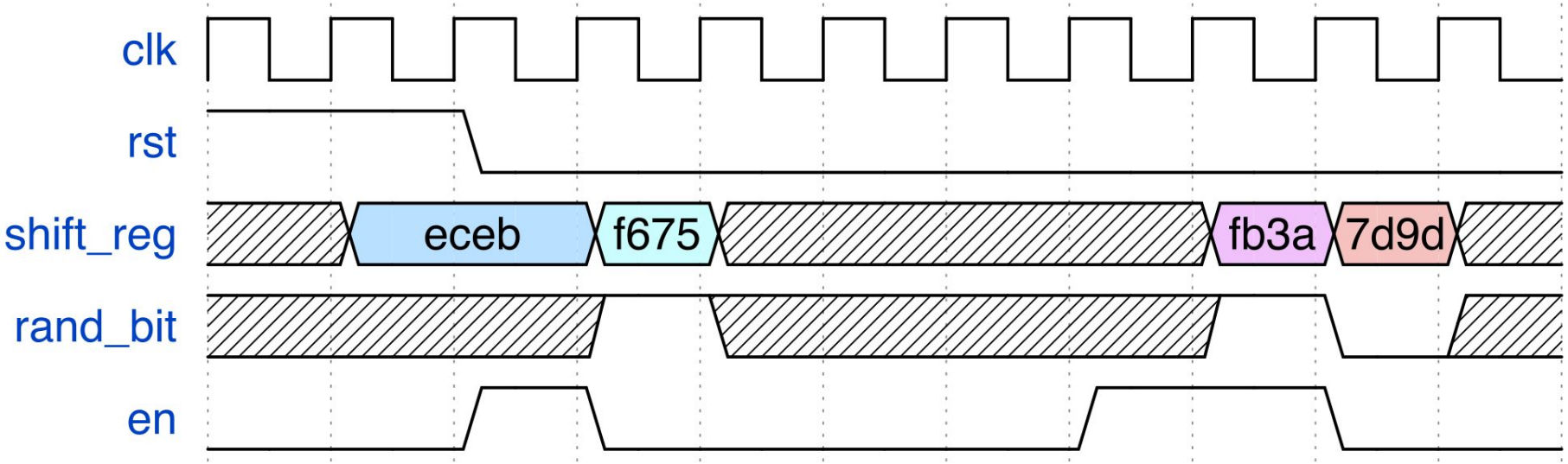
- Push your code on your GitHub main branch before the deadline
- AG will test run sometime before the deadline
- Test run does not include all test cases
- Report will be pushed to your GitHub _grade branch

Part 1: SystemVerilog Refresher

- SystemVerilog is an HDL and an HVL.
- In ECE 385, you used it mostly as an HDL to write RTL.
- This part of the MP is an RTL refresher: implementing an LFSR, TB provided:



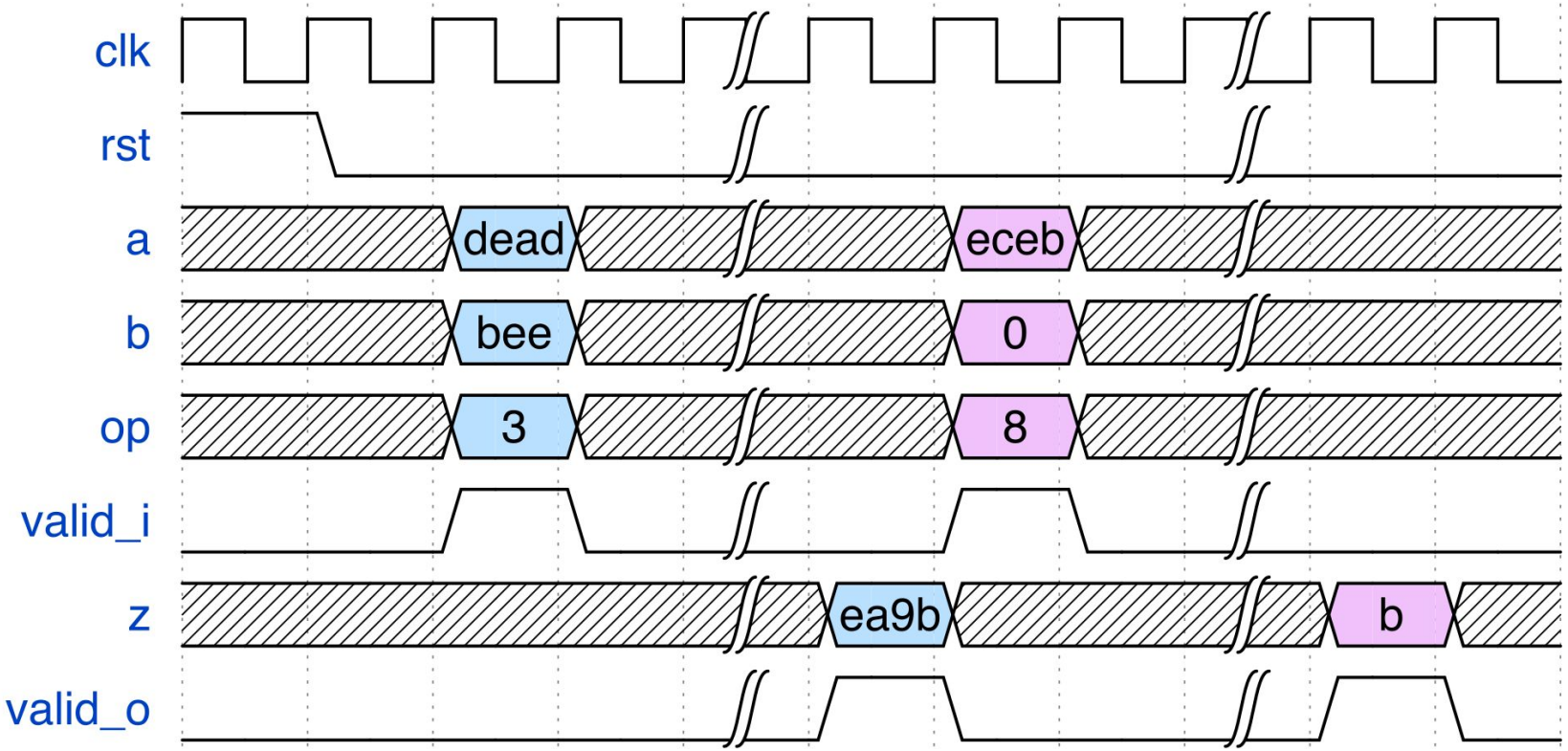
LFSR Waveform



Part 2.1: Common Errors

- Common errors we've seen in past semesters:
  - Logical/functional errors
  - Accidentally inferred latches
  - Long critical paths
- You are provided with a buggy ALU design and a partial testbench.
- Functional errors can be found by completing the testbench and using the same simulation/debug flow as part 1.

ALU Waveform



Latch??

- What is a inferred latch?

Why does this infer a latch?

```
always_comb begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
    endcase
end
```

Latch fix 1

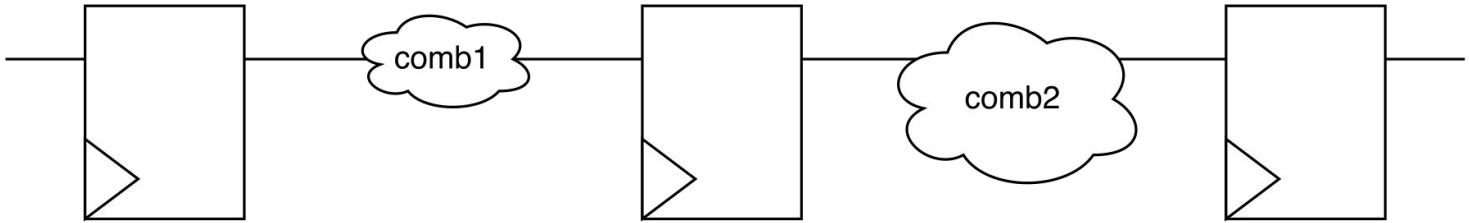
```
always_comb begin
    out = 'x;
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
    endcase
end
```

Latch fix 2 (Recommended)

```
always_comb begin
    unique case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default: out = ‘x;
    endcase
end
```

Critical paths

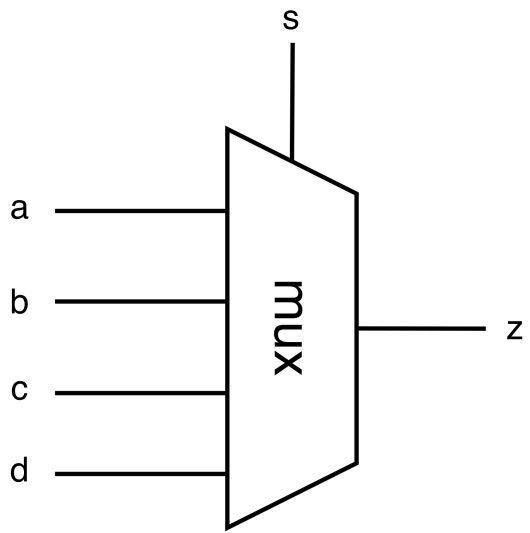
- Logic gates have delay
- Flip-flops need the input signal be stable for some time before the clock edge
- Most design can be abstracted as follows:

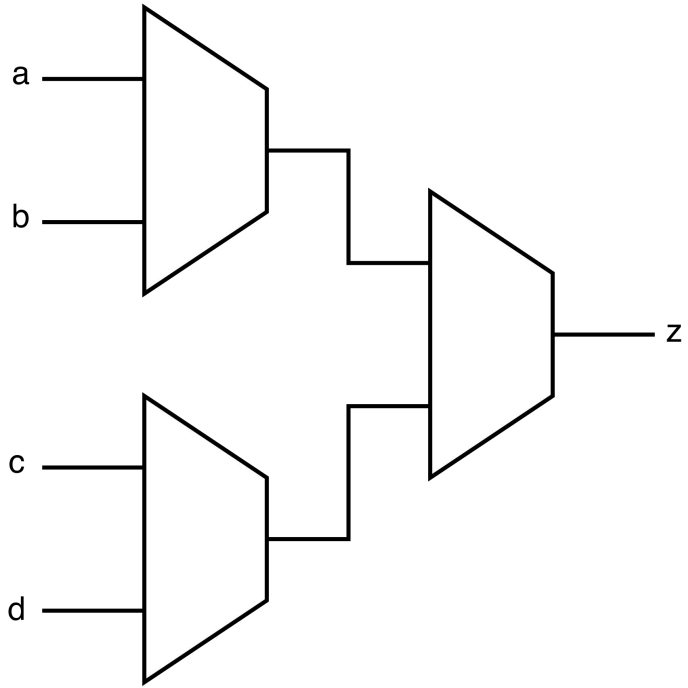


Critical paths

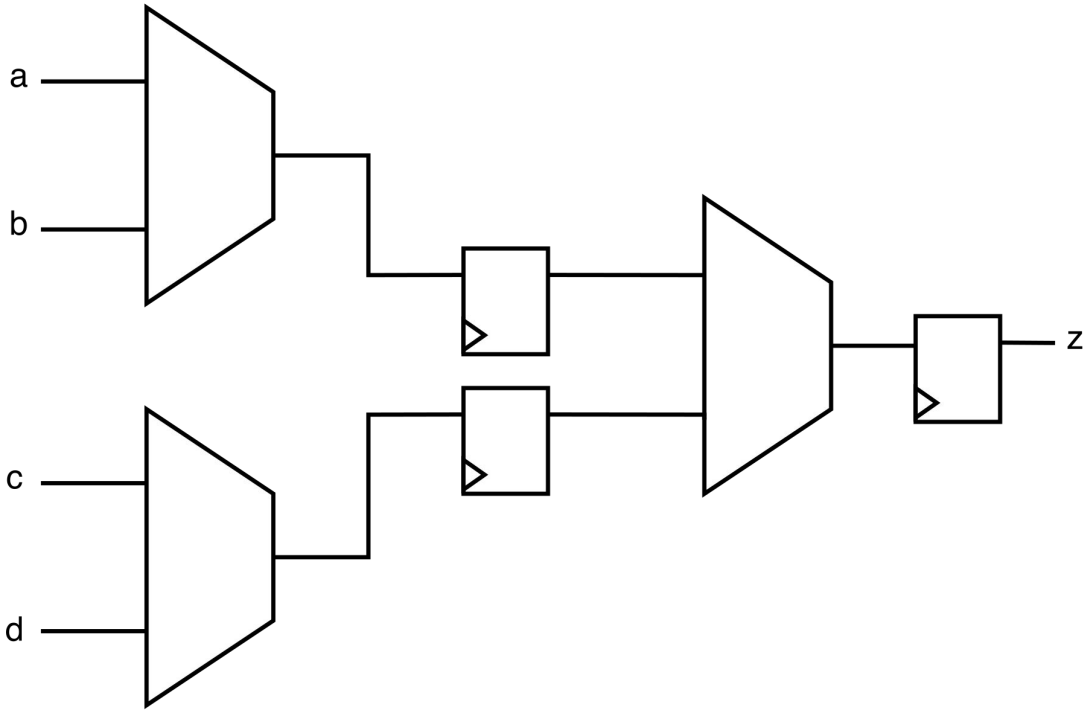
- Clock speed is dictated by the longest such path in the design
  - setup time + combinational time < clock time
- For a higher clock speed:
  - Use better technology (duh)
  - Reduce combinational delay for the critical path
  - Redesign to move delay somewhere else
  - Add registers
    - Add more registers
    - Add all the register you have
    - Maybe this is called pipelining? More will be covered in lecture later.

Pipelining to fix long paths





Pipelining to fix long paths



Part 2.2: Combinational Loop

- A combinational loop is formed when the input of some comb logic depends on its output.
- Some of these configurations can have a stable equilibrium (SR latch from ECE 120), but others do not.
- Comb loops are disallowed in ECE 411: you must use FFs and pure comb logic only.
- Accidental comb loops can cause simulation hangs.
- Easy to find and fix by running SpyGlass (lint).

Part 3: Constrained Random & Functional Coverage

- Task is to:
  - Generate random, valid RISC-V (RV32I) instructions using SystemVerilog.
  - Collect coverage to make sure the random instructions cover interesting cases.
- You will use the random instructions to verify a RISC-V core in Part 4.

Constrained Random Value Generation

```
class RandOp;
  rand bit [3:0] op;
  constraint op_valid_c {
    op < 4'd9;
  }
endclass : RandOp
```

Constrained Random Value Generation

```
class RandEx;
  rand bit [31:0] a;
  rand bit [31:0] b;
  constraint ab_c {
    $clog2(a) < b;
    b < 32'hffffeceb;
    solve b before a;
  }
endclass : RandEx
```

Functional Coverage

- How do we know our randomized testing is ‘good’
  - SystemVerilog random is stable, re-running tb does not give us more coverage
  - Not guaranteed that randomness is able to ‘find’ edge cases
  - Randomness will never be perfect, but can be better
- How? -> “statistics collection” on input stimulus (aka functional coverage)
  - Just like code coverage from software.
- Coverage reports are fun to read
  - Tell you where the randomness ‘looked’

Sample coverage report

- A “bin” is a set of values, any of which needs to be hit at least N times to be “covered”
- Here, a variable **a** needs to cover 64 bins to touch 100% coverage.
  - Bins not covered will show up in red, clearly indicating a shortcoming in your tb
- Can specify minimum number of relevant stimuli for a given bin (if it works 10 times it must be perfect right???)

Summary for Variable a

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	64	0	64	100.00

Automatically Generated Bins for a

Bins

NAME	COUNT	AT LEAST
auto[0000000000000000:03fffffffffffff]	136	1
auto[0400000000000000:07fffffffffffff]	129	1
auto[0800000000000000:0bfffffffffffff]	134	1
auto[0c00000000000000:0fffffffffffffff]	133	1
auto[1000000000000000:13fffffffffffff]	151	1
auto[1400000000000000:17fffffffffffff]	148	1
auto[1800000000000000:1bfffffffffffff]	135	1
auto[1c00000000000000:1fffffffffffffff]	143	1
auto[2000000000000000:23fffffffffffff]	138	1
auto[2400000000000000:27fffffffffffff]	143	1
auto[2800000000000000:2bfffffffffffff]	139	1
auto[2c00000000000000:2fffffffffffffff]	156	1
auto[3000000000000000:33fffffffffffff]	147	1
auto[3400000000000000:37fffffffffffff]	131	1
auto[3800000000000000:3bfffffffffffff]	128	1

How to write a covergroup

```
covergroup cg with function sample(...);
  coverpoint a;
  coverpoint b;

  coverpoint op {
    bins range[] = {[0:8]};
  }
endgroup : cg
```

Cross coverage

- Sometimes, covering combinations of variables can be useful.
- For example, you may want to cover a case where a variable “a” has values {0, 1, 2, 3} AND b has values {0, 1}.
- Thus we get a cross product: {(0,0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1))}, each pair needs to be covered.
- SystemVerilog allows us to mark some members of that cross product as illegal or ignored.

funct7_cross

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

funct7_cross

```
funct7_cross : cross instr.r_type.opcode,
instr.r_type.funct3, instr.r_type.funct7 {


    ignore_bins OTHER_INSTS = funct7_cross with

    (!(instr.r_type.opcode inside {op_reg, op_imm}));

}
```

Part 4: Verify a CPU

- Provided a RISC-V32I CPU
  - Multi cycle design
  - With bugs!
- Fix this CPU so that it is RISC-V32I compliant
  - With the exception of not implementing FENCE*, ECALL, EBREAK, and CSRR*
- Read the RISC-V Specification
  - And then read it again :)
  - You'll need to be intimately familiar with it for mp_pipeline and mp_ooo anyway

How?

- Use your random testbench from part 3
- Write RISC-V assembly to do manual test
  - Sample assembly file provided
  - Write your own
  - Compile script included and automatically called on make

RVFI

- Golden model written in SystemVerilog
- Concurrent execution with your CPU
- Check commits to your architectural states
  - Any modification to register and memory in this context
- Prints error if your result is different from its
  - Prints what it sees from your CPU (prefixed with “rvfi”)
  - Prints what it expected (prefixed with “spec”)

Spike

- Golden model written in C++
- Run separately from the command line
- Interactive debugging (like gdb)
- Prints out commit log
  - Your CPU will also print out a commit log in the same format
  - Use diff to verify compliance