

Intro to Digital Design

FSM Design, MUXes, Adders

Instructor: Chris Thachuk

Teaching Assistants:

Eujean Lee

Stephanie Osorio-Tristan

Nandini Talukdar

Wen Li

Relevant Course Information

- ❖ Lab 6 – Connecting multiple FSMs in Tug of War game
 - *Bigger* step up in difficulty from Lab 5
 - Putting together complex system – interconnections!
 - Bonus points for smaller resource usage

Clock Divider (not for simulation)

❖ Why/how does this work?

CLOCK_50: 50 MHz clock on your FPGA

→ use divided_clocks [#] everywhere else in your system

```
// divided_clocks[0]=25MHz, [1]=12.5Mhz, ...
module clock_divider (clock, divided_clocks);
  input logic      clock;
  output logic [31:0] divided_clocks;

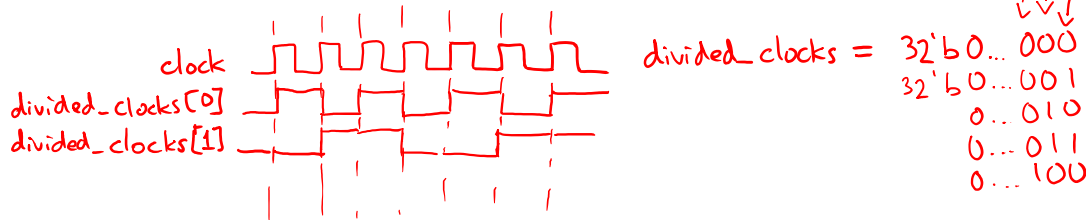
  initial
    divided_clocks = 0;

  always_ff @(posedge clock)
    divided_clocks <= divided_clocks + 1;

endmodule // clock_divider
```

↑ 32 slower "clocks"


changes every fourth trigger
changes every other trigger
changes every clock trigger



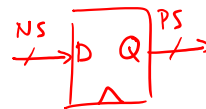
Outline

- ❖ **FSM Design**
- ❖ Multiplexors
- ❖ Adders

FSM Design Process

- 1) Understand the problem 
- 2) Draw the state diagram
311 knowledge
- 3) Use state diagram to produce state table
read off transitions
- 4) Implement the combinational control logic

CL + SL
gates + registers



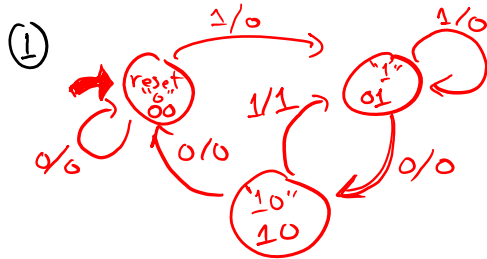
Practice: String Recognizer FSM

- ① Draw the FSM
- ② Truth Table
- ③ Simplify Logic
- ④ Circuit Diagram

❖ Recognize the string 101 with the following behavior

- Input: 1 0 0 1 0 1 1 1 0 0 1 0
- Output: 0 0 0 0 0 1 0 1 0 0 0 0 0

❖ State diagram to implementation:



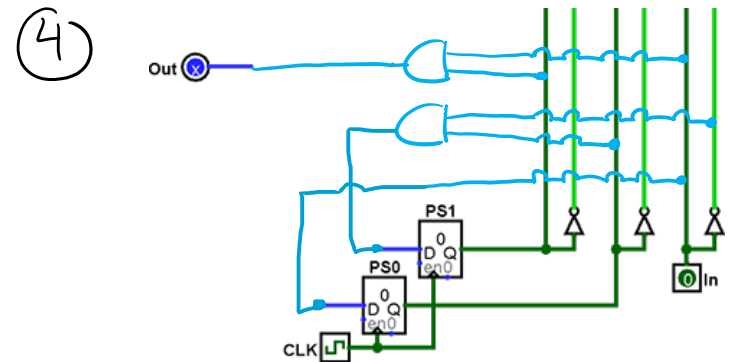
②

PS	In	NS	Out
00	0	00	0
00	1	01	0
01	0	10	0
01	1	01	1
10	0	00	0
10	1	01	1
X	X	X	X

③

In	PS	00	01	11	10
0	0	0	1	X	0
1	0	0	0	X	0
0	1	1	1	X	1
1	1	0	0	X	1

$NS_1 = \bar{In} \cdot PS_0$ $NS_0 = In$ $Out = In \cdot PS_1$



HDL Organization

- ❖ Most problems are best solved with multiple pieces – how to best organize your system and code?
- ❖ Everything is computed in parallel
 - We use routing elements (next lecture) to select between (or ignore) multiple outcomes/parts
 - This is why we use block diagrams and waveforms
- ❖ A module is not a *function*, it is closest to a *class*
 - Something that you *instantiate*, not something that you *call* – hardware cannot appear and disappear spontaneously
 - Should treat modules as *resource managers* rather than temporary helpers
 - This can include having internal modules

Block Diagrams

- ❖ Block diagrams are the basic design tool for digital logic.
 - The diagram itself is a module → **inputs and outputs shown and connected**
 - Major components are represented by blocks (“black boxes”) with their internals abstracted away → **each block becomes its own module**
 - All ports for each block should be shown and labeled and connected to the appropriate part(s) of the rest of the system → **sets your port connections**
 - Wires and other basic building blocks can be added/shown as needed
- ❖ From [Wikipedia](#): The goal is to “[end] in block diagrams detailed enough that each individual block can be easily implemented.”
 - For designs that involve multiple modules, should always create your block diagram *before* coding anything!

Subdividing FSMs Example

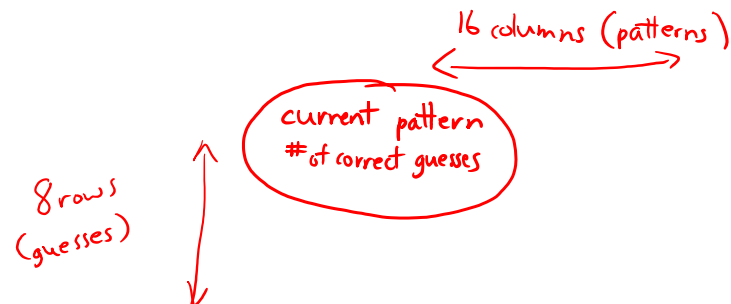
❖ “Psychic Tester”

- Machine generates a 4-bit pattern
- User tries to guess 8 patterns in a row to be deemed psychic

$$2^4 = 16 \text{ patterns}$$

0-7 correct guesses so far (8 total)

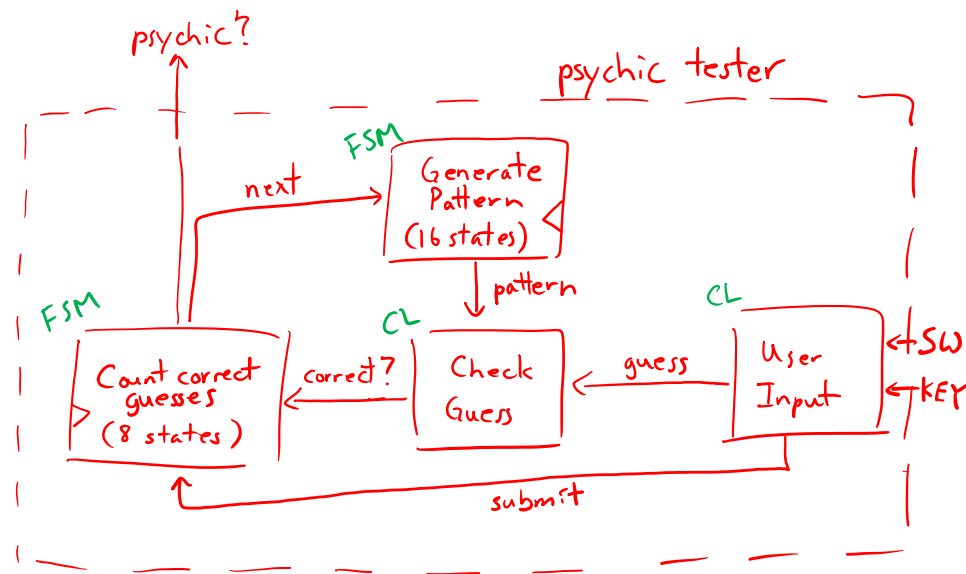
❖ States?



≈ 128 states total

Example: Plan First with Block Diagram

- ❖ Pieces?
 - Generate/pick pattern
 - User input (guess)
 - Check guess
 - Count correct guesses



Example: Blocks → Modules

❖ Pieces?

- Generate/pick pattern
 - `module genPatt (pattern, next, clock);`
- User input (guess)
 - `module userIn (guess, submit, KEY);`
- Check guess
 - `module checkGuess (correct, guess, pattern);`
- Count correct guesses
 - `module countRight (psychic, next, correct, submit, clock);`

Example: Implementation & Testing

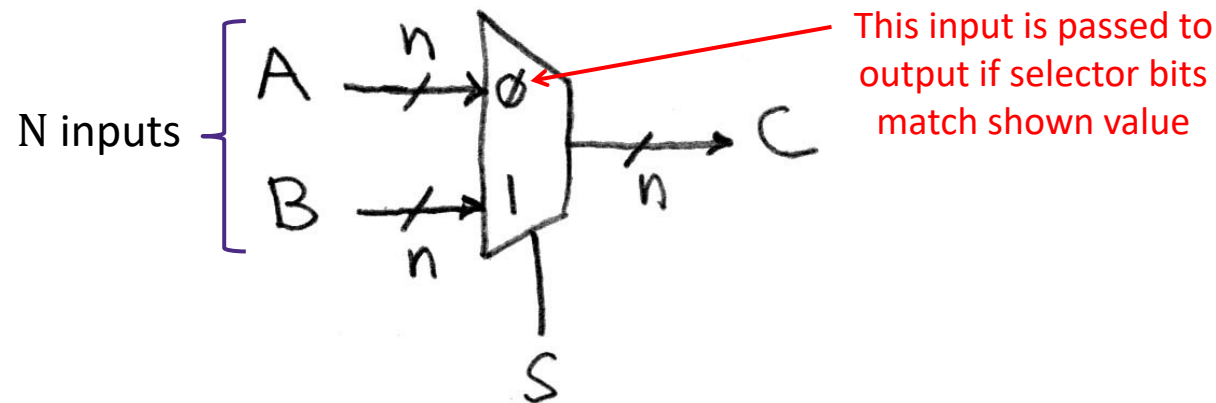
- 1) Create individual submodules
- 2) Create submodules test benches – test as usual
 - CL – run through all input combinations
 - SL – take every transition that you care about
- 3) Create top-level module
 - Create instance of each submodule
 - Create wires/nets to connect signals between submodules, inputs, and outputs
- 4) Create top-level test bench
 - Goal is to check the interconnections between submodules – does input/state change in one submodule trigger the expected change in other submodules?

Outline

- ❖ FSM Design
- ❖ **Multiplexors**
- ❖ Adders

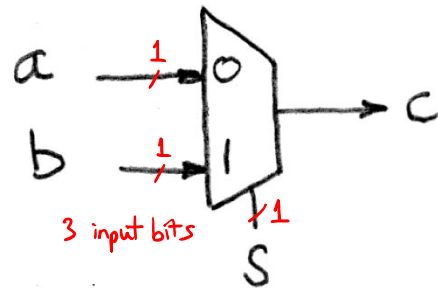
Data Multiplexor

- ❖ Multiplexor (“MUX”) is a *selector*
 - Direct one of many ($N = 2^s$) n -bit wide inputs onto output
 - Called a n -bit, N -to-1 MUX
 - bus widths* ↗
 - ↖ *possible selections*
- ❖ Example: n -bit 2-to-1 MUX
 - Input S (s bits wide) selects between two inputs of n bits each



Review: Implementing a 1-bit 2-to-1 MUX

❖ Schematic:



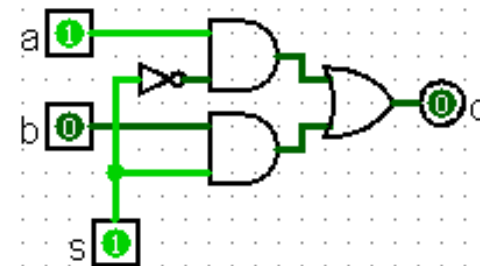
❖ Boolean Algebra:

$$\begin{aligned}
 c &= \bar{s}a\bar{b} + \bar{s}ab + s\bar{a}b + sab \\
 &= \bar{s}(a\bar{b} + ab) + s(\bar{a}b + ab) \\
 &= \bar{s}(a(\bar{b} + b)) + s((\bar{a} + a)b) \\
 &= \bar{s}(a(1) + s((1)b) \\
 &= \bar{s}a + sb
 \end{aligned}$$

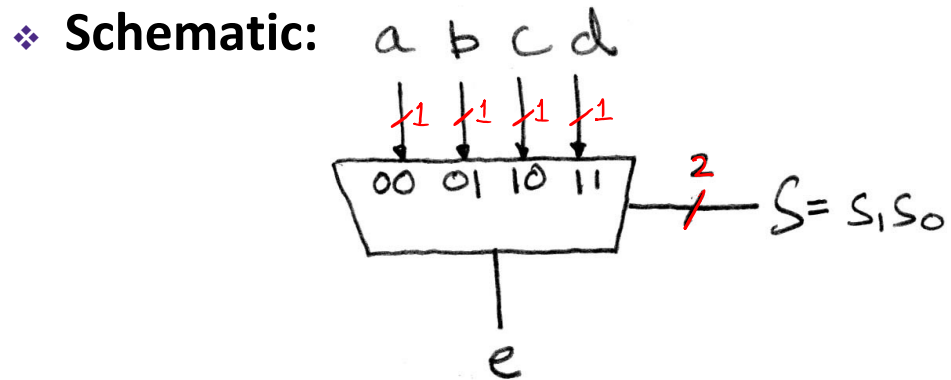
❖ Truth Table:

s	a	b	c
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

❖ Circuit Diagram:



1-bit 4-to-1 MUX



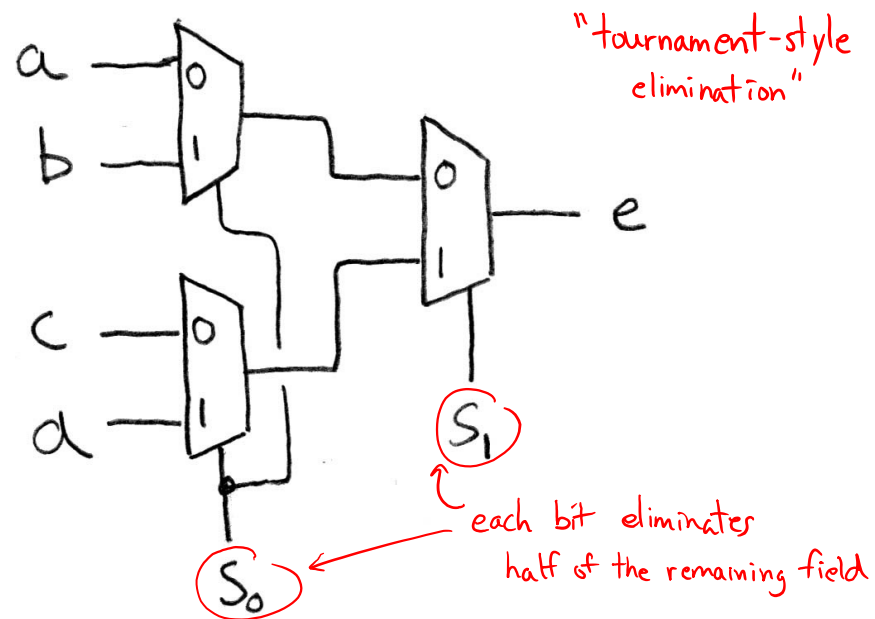
❖ Truth Table: How many rows? 6 inputs $\rightarrow 2^6$ rows

❖ Boolean Expression:

$$e = \bar{s}_1\bar{s}_0a + \bar{s}_1s_0b + s_1\bar{s}_0c + s_1s_0d$$

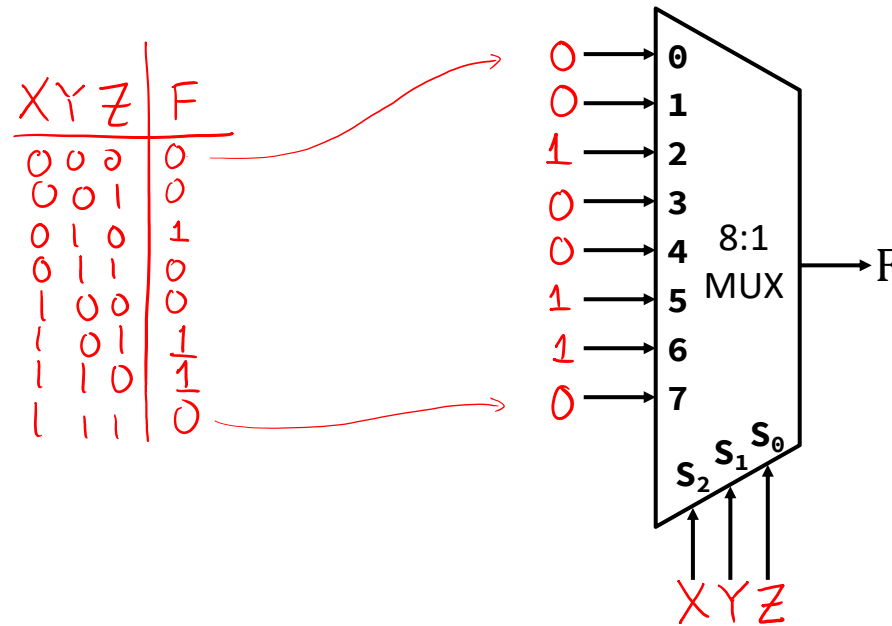
1-bit 4-to-1 MUX

- ❖ Can we leverage what we've previously built?
 - Alternative hierarchical approach:



Multiplexers in General Logic

- Implement $F = \overset{010}{X\bar{Y}Z} + \overset{110}{Y\bar{Z}}$ with a 8:1 MUX



Technology Break

Outline

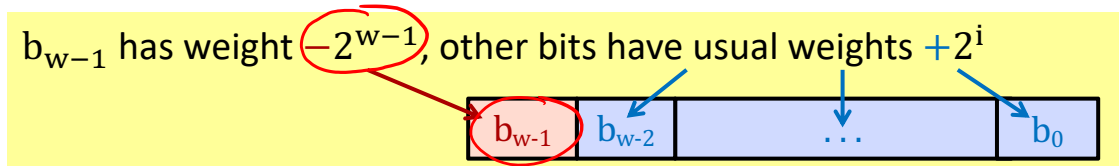
- ❖ FSM Design
- ❖ Multiplexors
- ❖ **Adders**

Review: Unsigned Integers

- ❖ Unsigned values follow the standard base 2 system
 - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_72^7 + b_62^6 + \dots + b_12^1 + b_02^0$
- ❖ In n bits, represent integers 0 to 2^n-1
- ❖ Add and subtract using the normal “carry” and “borrow” rules, just in binary

	$ \begin{array}{r} \text{Carry} \\ 111 \\ 63 \quad 00111111 \\ + 8 \quad +00001000 \\ \hline 71 \quad 01000111 \end{array} $		$ \begin{array}{r} \text{borrow} \\ \curvearrowright 2 \times 2 \\ 64 \quad 01000000 \\ - 8 \quad -00001000 \\ \hline 56 \quad 00111000 \end{array} $
--	---	--	--

Review: Two's Complement (Signed)



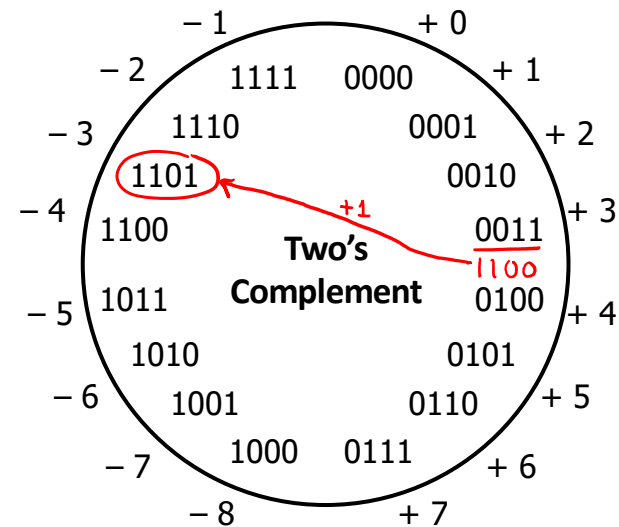
❖ Properties:

- In n bits, represent integers -2^{n-1} to $2^{n-1} - 1$
- Positive number encodings match unsigned numbers
- Single zero (encoding = all zeros)

❖ Negation procedure:

- Take the bitwise complement and then add one

$$(\sim x + 1 == -x)$$



Addition and Subtraction in Hardware

- ❖ The same bit manipulations work for both unsigned and two's complement numbers!
 - Perform subtraction via adding the negated 2nd operand:
 $A - B = A + (-B) = A + (\sim B) + 1$

❖ 4-bit examples:

		Two's	Un		Two's	Un
0 0 1 0		+2	2		1 0 0 0	-8 8
+ 1 1 0 0		-4	12		+ 0 1 0 0	+4 4
1 1 1 0		-2	14		1 1 0 0	-4 12
0 1 1 0		+6	6		1 1 1 1	-1 15
- 0 0 1 0	 	+2	2		- 1 1 1 0	-2 14
+ 1 1 0 1	 				+ 0 0 0 1	
0 1 0 0		+4	4		0 0 0 1	+1 1

Half Adder (1 bit)

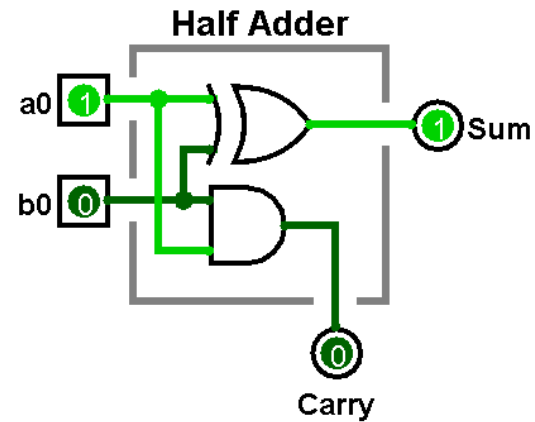
	a_3	a_2	a_1	a_0	$0/1$
+	b_3	b_2	b_1	b_0	$0/1$
	s_3	s_2	s_1	s_0	$0/1/2$

c_0 ↙

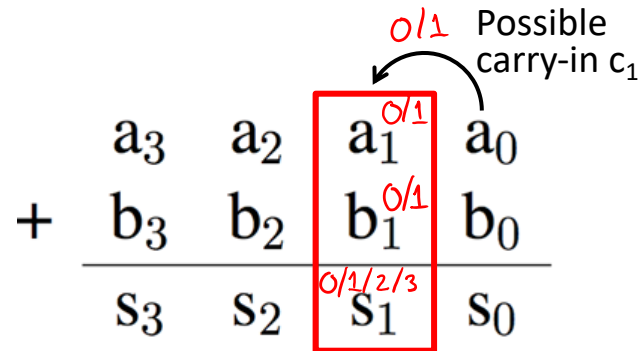
Carry = $a_0 b_0$
 Sum = $a_0 \oplus b_0$

Carry-out bit ↙

a_0	b_0	c_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Full Adder (1 bit)



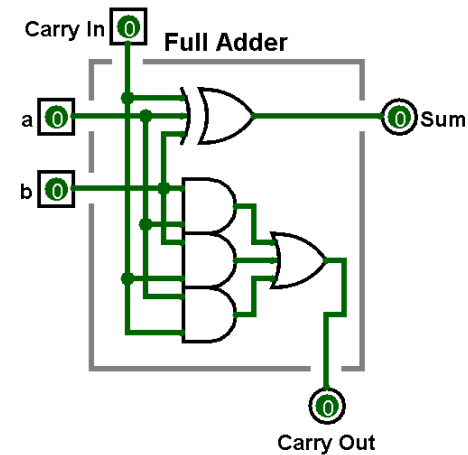
$$s_i = \text{XOR}(a_i, b_i, c_i)$$

$$c_{i+1} = \text{MAJ}(a_i, b_i, c_i)$$

$$= a_i b_i + a_i c_i + b_i c_i$$

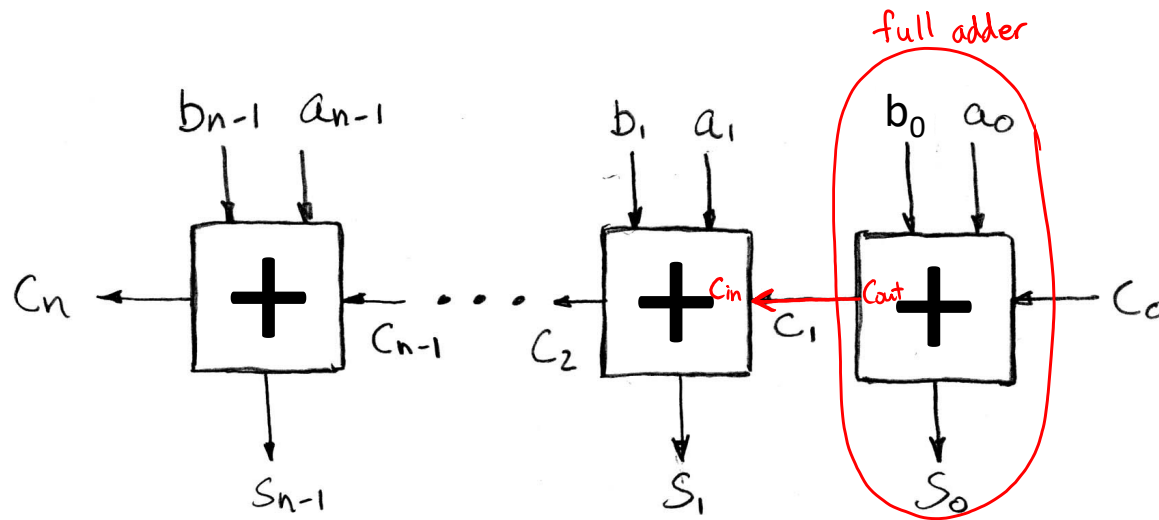
Carry-in Carry-out

c_i	a_i	b_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Multi-Bit Adder (N bits)

- ❖ Chain 1-bit adders by connecting CarryOut_i to CarryIn_{i+1}:

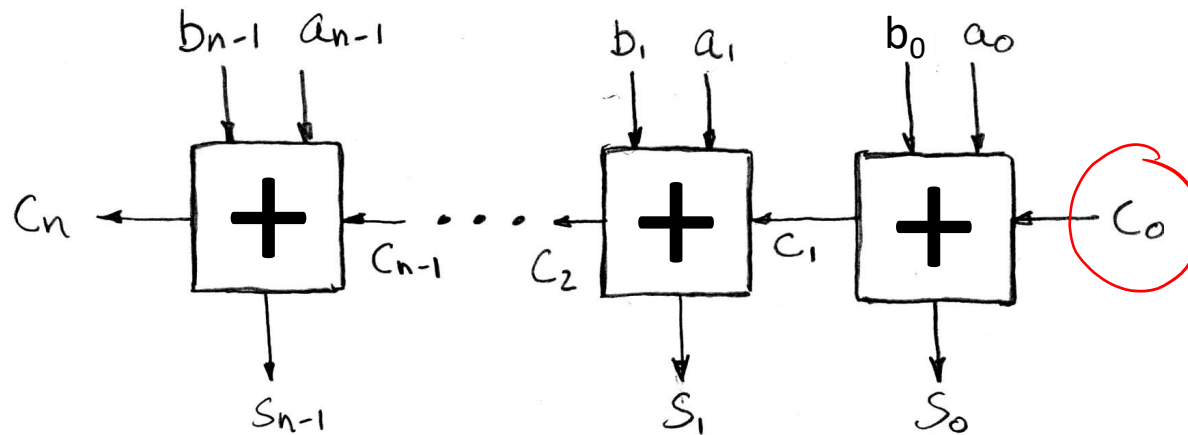


Subtraction?

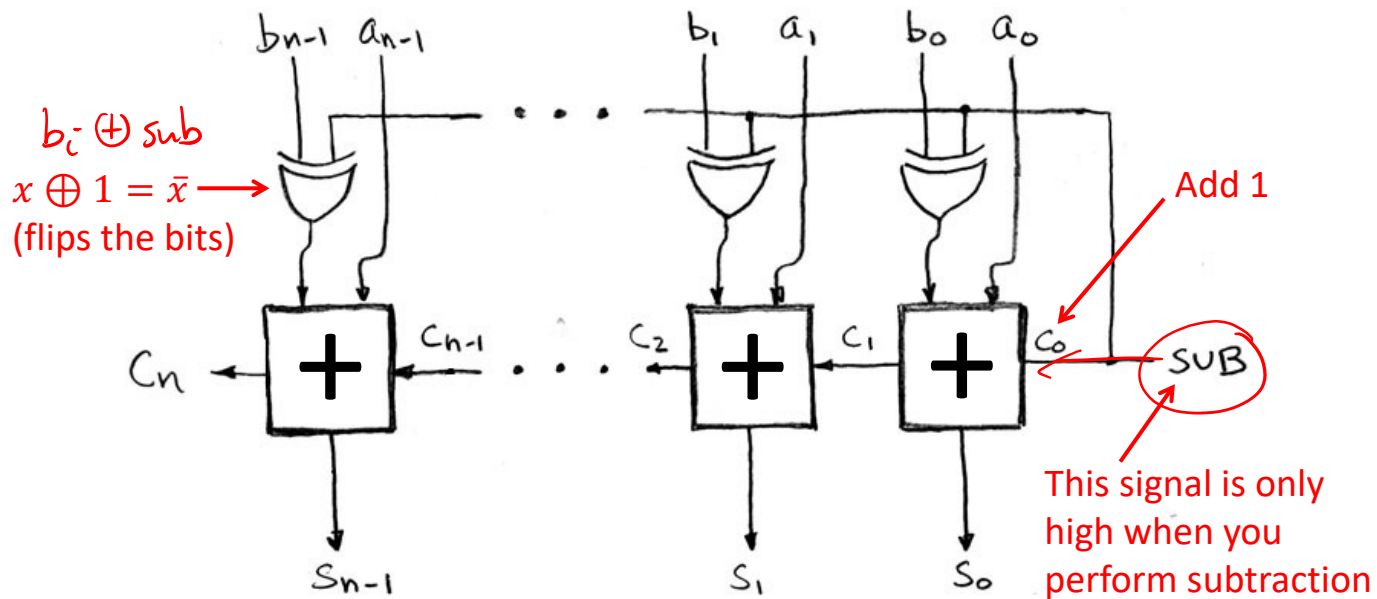
❖ Can we use our multi-bit adder to do subtraction?

- Flip the bits and add 1?
 - $X \oplus 1 = \bar{X}$
 - CarryIn₀ (using full adder in all positions)

$x \& 0 = 0$
 $x \& 1 = x$
 $x | 0 = x$
 $x | 1 = 1$
 $x \wedge 0 = x$
 $x \wedge 1 = \bar{x}$



Multi-bit Adder/Subtractor



Detecting Arithmetic Overflow

- ❖ **Overflow:** When a calculation produces a result that can't be represented in the current encoding scheme
 - Integer range limited by fixed width
 - Can occur in both the positive and negative directions
- ❖ **Unsigned Overflow**
 - Result of add/sub is $> U_{Max}$ or $< U_{min}$
 $0b11\dots1$ $0b00\dots0$
- ❖ **Signed Overflow**
 - Result of add/sub is $> T_{Max}$ or $< T_{Min}$
 $0b01\dots1$ $0b10\dots0$
 - $(+) + (+) = (-)$ or $(-) + (-) = (+)$

Signed Overflow Examples

<table border="0" style="width: 100%;"> <tr> <td style="text-align: right;">0 1 0 1</td> <td style="text-align: left;">Two's</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 0 1 1</td> <td style="text-align: left;">+5</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 0 1 1</td> <td style="text-align: left;">+3</td> <td></td> </tr> <tr> <td style="text-align: right;">-----</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">1 0 0 0</td> <td style="text-align: left;">-8</td> <td style="text-align: right;">overflow</td> </tr> <tr> <td style="text-align: right;">x ~</td> <td></td> <td></td> </tr> </table>	0 1 0 1	Two's		+ 0 0 1 1	+5		+ 0 0 1 1	+3		-----			1 0 0 0	-8	overflow	x ~			<table border="0" style="width: 100%;"> <tr> <td style="text-align: right;">1 0 0 1</td> <td style="text-align: left;">Two's</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 1 1 1 0</td> <td style="text-align: left;">-7</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 1 1 1 0</td> <td style="text-align: left;">-2</td> <td></td> </tr> <tr> <td style="text-align: right;">-----</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">0 1 1 1</td> <td style="text-align: left;">+7</td> <td style="text-align: right;">overflow</td> </tr> <tr> <td style="text-align: right;">~ x</td> <td></td> <td></td> </tr> </table>	1 0 0 1	Two's		+ 1 1 1 0	-7		+ 1 1 1 0	-2		-----			0 1 1 1	+7	overflow	~ x		
0 1 0 1	Two's																																				
+ 0 0 1 1	+5																																				
+ 0 0 1 1	+3																																				

1 0 0 0	-8	overflow																																			
x ~																																					
1 0 0 1	Two's																																				
+ 1 1 1 0	-7																																				
+ 1 1 1 0	-2																																				

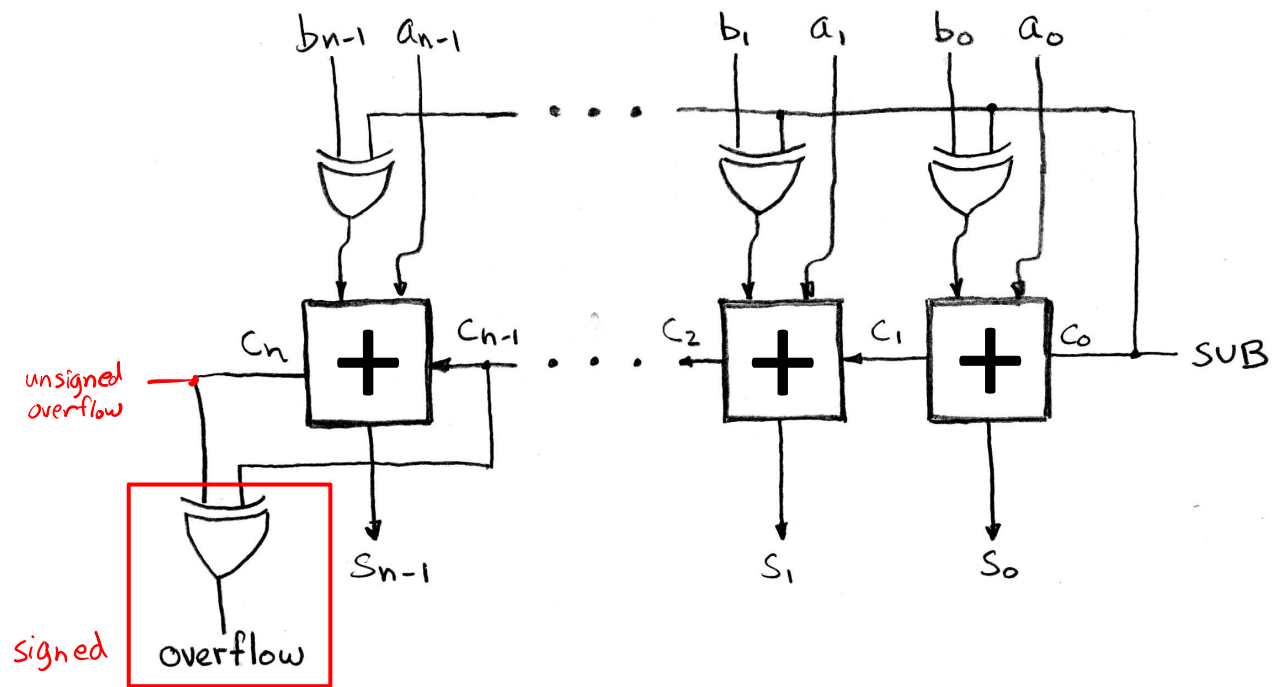
0 1 1 1	+7	overflow																																			
~ x																																					
<p style="color: red; font-style: italic;">sign bit</p>																																					
<table border="0" style="width: 100%;"> <tr> <td style="text-align: right;">0 1 0 1</td> <td style="text-align: left;">Two's</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 0 1 0</td> <td style="text-align: left;">+5</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 0 1 0</td> <td style="text-align: left;">+2</td> <td></td> </tr> <tr> <td style="text-align: right;">-----</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">0 1 1 1</td> <td style="text-align: left;">7</td> <td></td> </tr> <tr> <td style="text-align: right;">x x</td> <td></td> <td></td> </tr> </table>	0 1 0 1	Two's		+ 0 0 1 0	+5		+ 0 0 1 0	+2		-----			0 1 1 1	7		x x			<table border="0" style="width: 100%;"> <tr> <td style="text-align: right;">1 1 0 0</td> <td style="text-align: left;">Two's</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 1 0 0</td> <td style="text-align: left;">-4</td> <td></td> </tr> <tr> <td style="text-align: right;">+ 0 1 0 0</td> <td style="text-align: left;">4</td> <td></td> </tr> <tr> <td style="text-align: right;">-----</td> <td></td> <td></td> </tr> <tr> <td style="text-align: right;">0 0 0 0</td> <td style="text-align: left;">0</td> <td></td> </tr> <tr> <td style="text-align: right;">~ ~</td> <td></td> <td></td> </tr> </table>	1 1 0 0	Two's		+ 0 1 0 0	-4		+ 0 1 0 0	4		-----			0 0 0 0	0		~ ~		
0 1 0 1	Two's																																				
+ 0 0 1 0	+5																																				
+ 0 0 1 0	+2																																				

0 1 1 1	7																																				
x x																																					
1 1 0 0	Two's																																				
+ 0 1 0 0	-4																																				
+ 0 1 0 0	4																																				

0 0 0 0	0																																				
~ ~																																					

$$\boxed{\text{overflow} = C_n \wedge C_{n-1}}$$

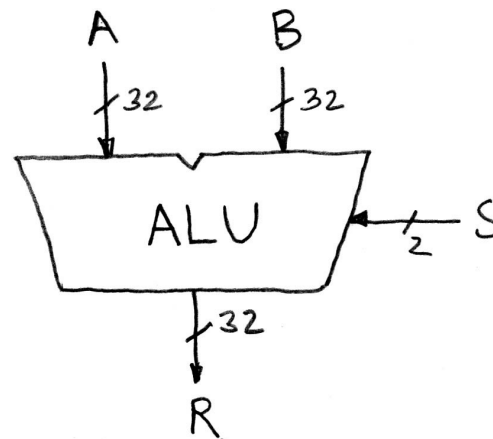
Multi-bit Adder/Subtractor with Overflow



Arithmetic and Logic Unit (ALU)

- ❖ Processors contain a special logic block called the “Arithmetic and Logic Unit” (ALU)
 - Here’s an easy one that does ADD, SUB, bitwise AND, and bitwise OR (for 32-bit numbers)

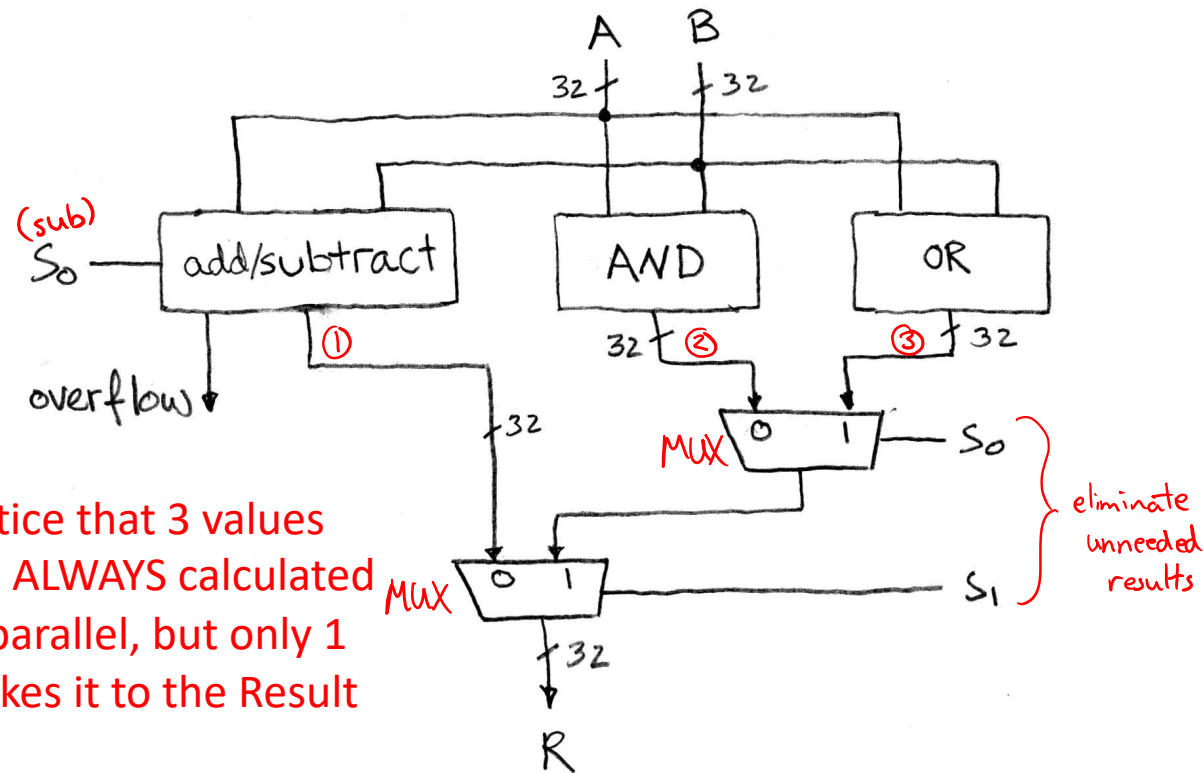
- ❖ **Schematic:**



when $S=00$, $R = A+B$
when $S=01$, $R = A-B$
when $S=10$, $R = A\&B$
when $S=11$, $R = A|B$

*"arbitrary" choice,
but affects
implementation*

Simple ALU Schematic



1-bit Adders in Verilog

❖ What's wrong with this?

- Truncation!

```
module halfadd1 (s, a, b);  
  output logic s;  
  input logic a, b;  
  always_comb begin  
    s = a + b;  
  end  
endmodule
```

single bit

❖ Fixed:

- Use of {sig, ..., sig} for *concatenation*

```
module halfadd2 (c, s, a, b);  
  output logic c, s;  
  input logic a, b;  
  
  always_comb begin  
    {c, s} = a + b;  
  end  
endmodule
```

could have been:
 $s = a \oplus b;$
 $c = a \& b;$

order matters!

Ripple-Carry Adder in Verilog

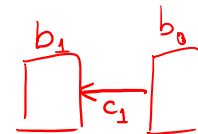
```
module fulladd (cout, s, cin, a, b);
  output logic cout, s;
  input  logic cin, a, b;

  always_comb begin
    {cout, s} = cin + a + b;
  end
endmodule
```

❖ Chain full adders?

```
module add2 (cout, s, cin, a, b);
  output logic cout; output logic [1:0] s;
  input  logic cin;   input  logic [1:0] a, b;
  logic  c1;

  fulladd b1 (cout, s[1], c1, a[1], b[1]);
  fulladd b0 (c1,   s[0], cin, a[0], b[0]);
endmodule
```



Add/Sub in Verilog (parameterized)

- ❖ Variable-width add/sub (with overflow, carry)

```

module addN #(parameter N=32) (OF, CF, S, sub, A, B);
  output logic      OF, CF; // overflow and carry "flags"
  output logic [N-1:0] S;
  input  logic      sub; // parameter changes bus widths
  input  logic [N-1:0] A, B;
  logic [N-1:0] D; // possibly flipped B
  logic      C2; // second-to-last carry-out

  always_comb begin
    D = B ^ {N{sub}}; // replication operator
    {C2, S[N-2:0]} = A[N-2:0] + D[N-2:0] + sub;
    {CF, S[N-1]} = A[N-1] + D[N-1] + C2;
    OF = CF ^ C2;
  end
endmodule // addN

```

default value (points to N=32)
parameter changes bus widths (points to sub)
flip all bits of B if sub==1 (points to D = B ^ {N{sub}})

- Here using OF = overflow flag, CF = carry flag (from condition flags in x86-64 CPUs)

Add/Sub in Verilog (parameterized)

```
module addN_tb ();
  parameter N = 4;
  logic      sub;
  logic [N-1:0] A, B;
  logic      OF, CF;
  logic [N-1:0] S;

  addN #(N) dut (.OF, .CF, .S, .sub, .A, .B);

  initial begin
    #100; sub = 0; A = 4'b0101; B = 4'b0010; // 5 + 2
    #100; sub = 0; A = 4'b1101; B = 4'b1011; // -3 + -5
    #100; sub = 0; A = 4'b0101; B = 4'b0011; // 5 + 3
    #100; sub = 0; A = 4'b1001; B = 4'b1110; // -7 + -2
    #100; sub = 1; A = 4'b0101; B = 4'b1110; // 5 - (-2)
    #100; sub = 1; A = 4'b1101; B = 4'b0101; // -3 - 5
    #100; sub = 1; A = 4'b0101; B = 4'b1101; // 5 - (-3)
    #100; sub = 1; A = 4'b1001; B = 4'b0010; // -7 - 2
    #100;
  end
endmodule // addN_tb
```

test
different
scenarios