



CSE140: Components and Design Techniques for Digital Systems

Register Transfer Level (RTL) Design

Slides from Tajana Simunic Rosing



CAPE

Please submit your evaluations !!!!

About the final

~150 minutes Final Exam

Problems including (but not limited to):

- ALU design
- ALU components (shifters, adders, etc.)
- FSM
 - Moore, Mealy, design of FSM using state table, excitation table
- Timing constraints
- RTL design and HLSM

You can have a cheat sheet with you, but nothing else

Topics for today



- Another example of RTL design
- RTL circuit delays and clock
- HLSM array and Register Files
- HLSM and C-code
- Data-dominated Vs Control-dominated RTL design

Example of RTL design

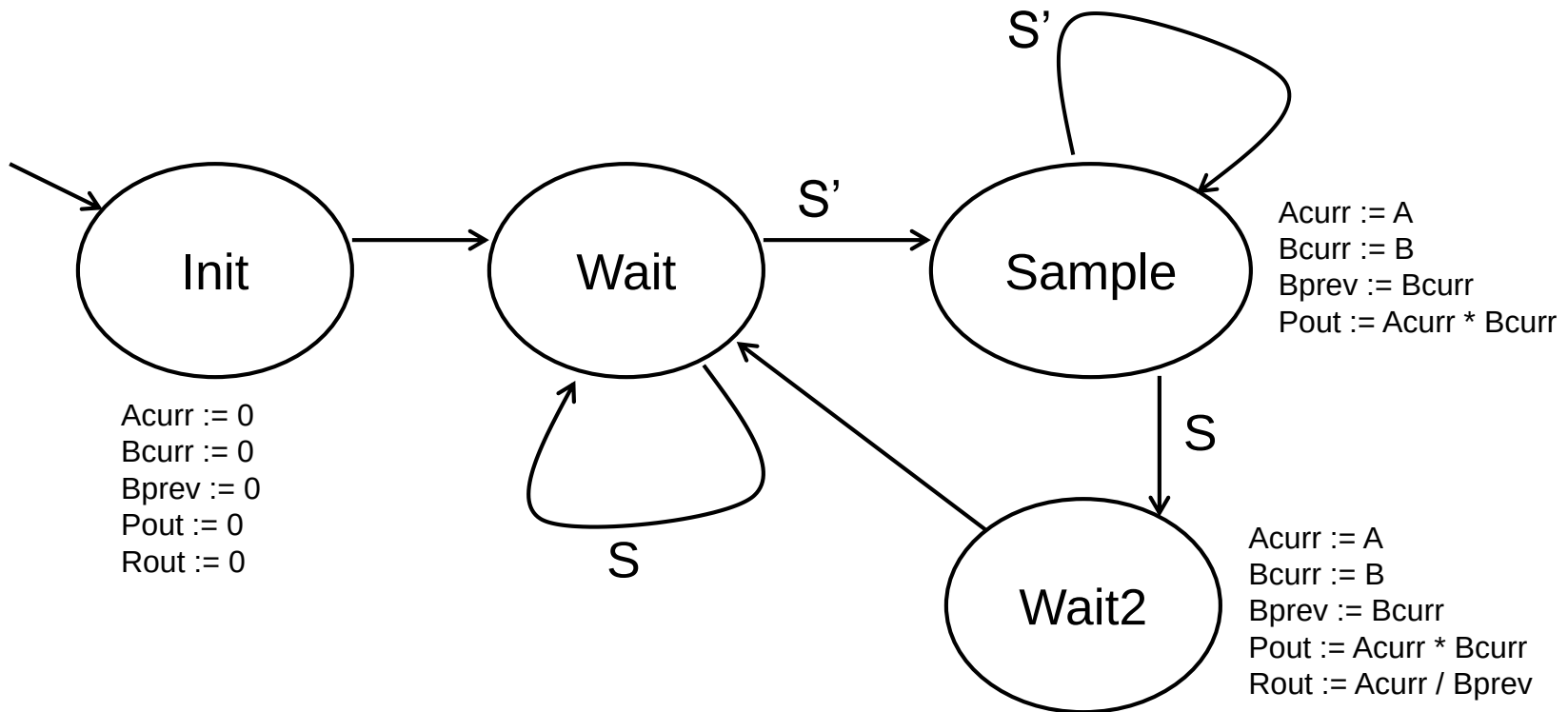
- Use the RTL design process to design a system that outputs the product (P) between:
 - The most recent sample of data input A.
 - The most recent sample of data input B.
- And the ratio (R) between:
 - The most recent sample of data input A.
 - The previous most recent sample of data input B.
- The system has two 8-bit unsigned data input A and B and two 8-bit unsigned outputs P and R.
- The data inputs are sampled when a single bit input S changes from 1 to 0. As long as S is 0, the system keeps on sampling new data and calculating P.
- When S changes to 1 again, in addition the system computes and outputs R for a single clock cycle.
- Choose the internal bitwidths that prevent overflow.

1: HLSM diagram

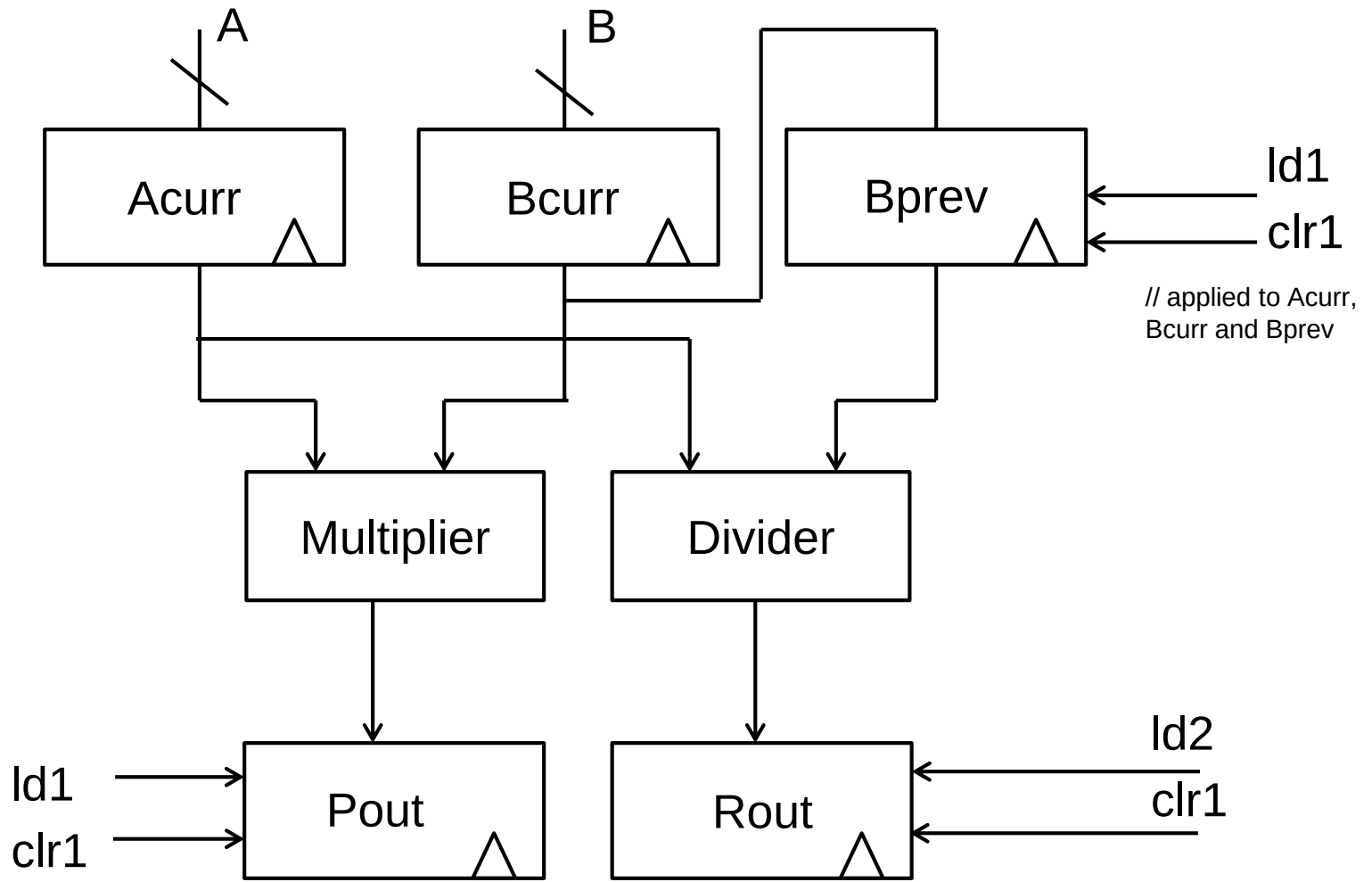
Input: A (8-bit), B (8-bit), S (1-bit)

Output: P (8-bit), R (8-bit)

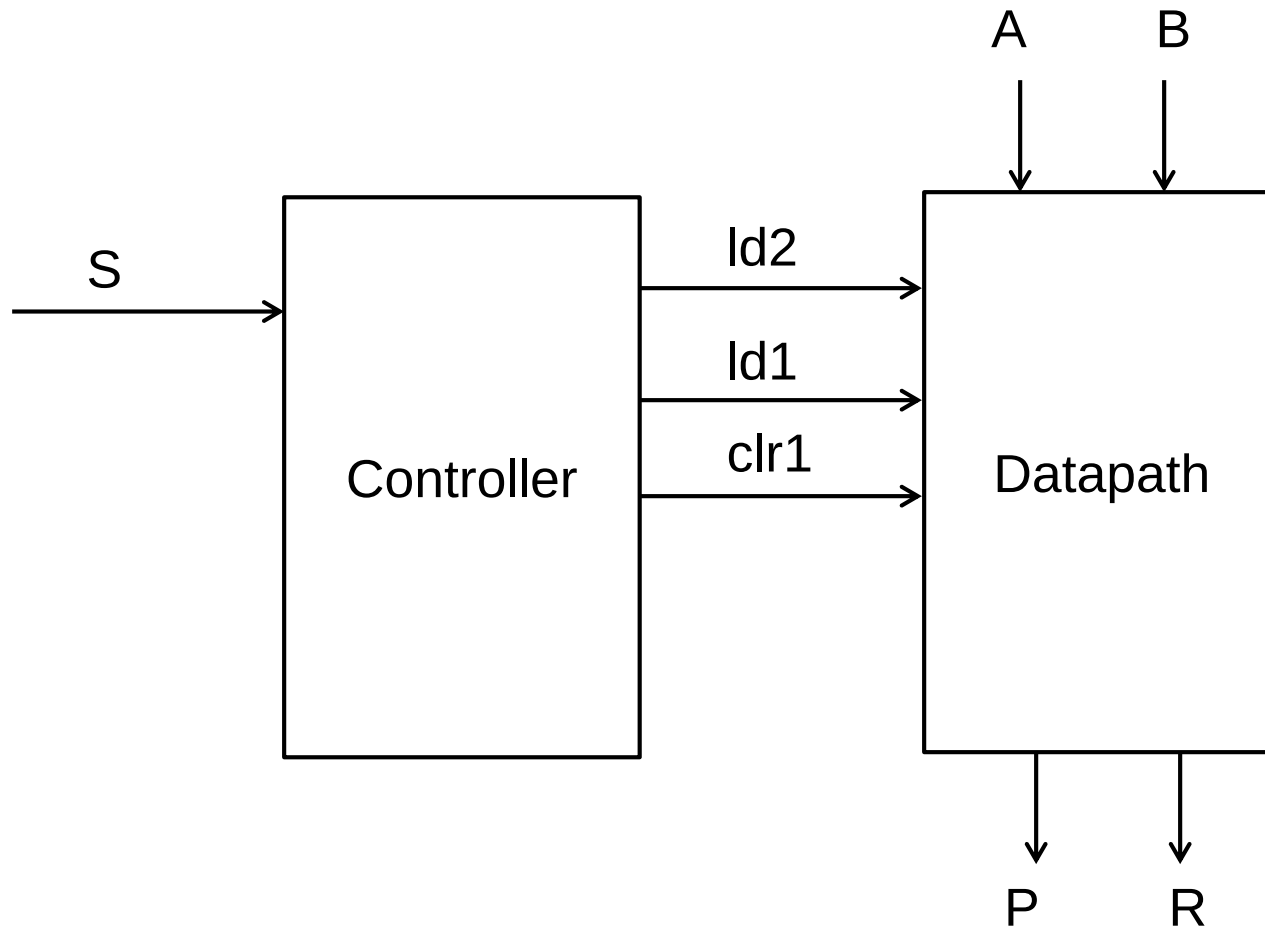
Local registers: Acurr (8-bit), Bcurr (8-bit), Bprev (8-bit), Pout (16-bit), Rout (8-bit)



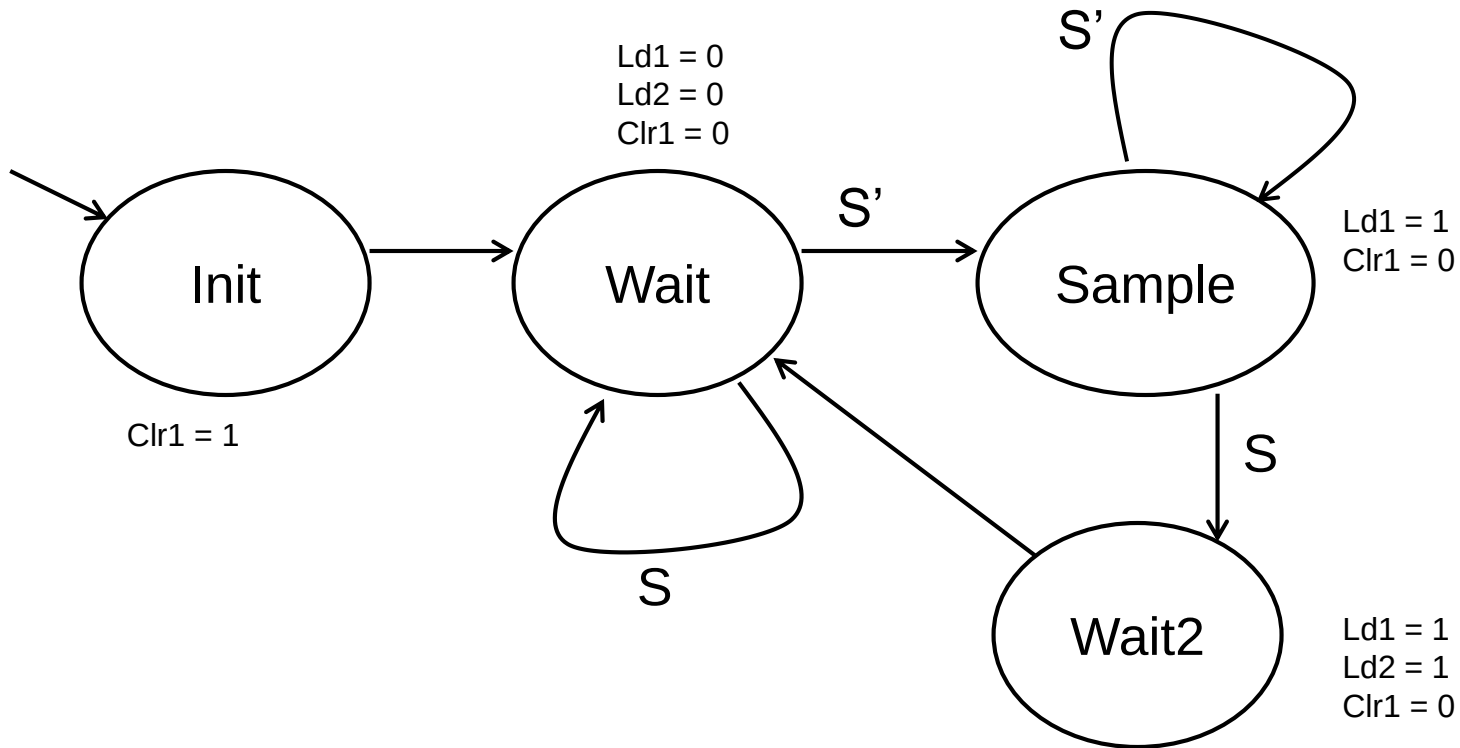
2.a: Datapath



2.b: Connect datapath and controller



2.c: Controller FSM



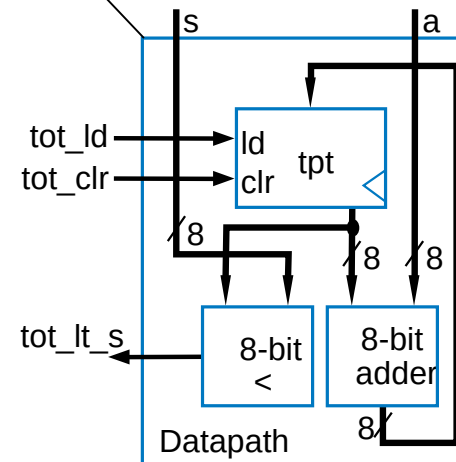
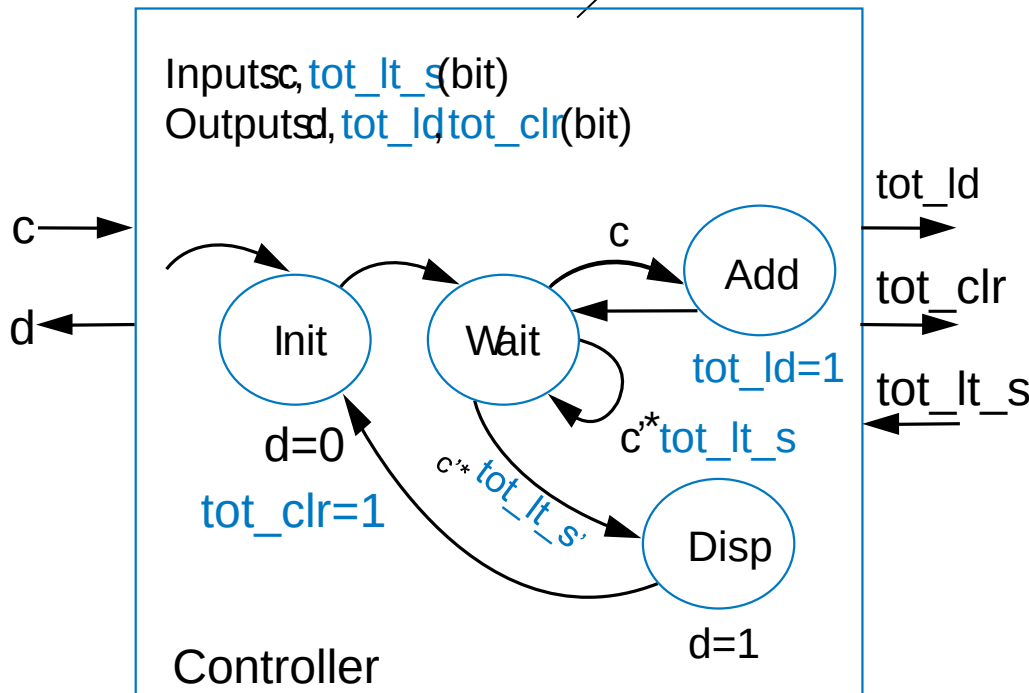
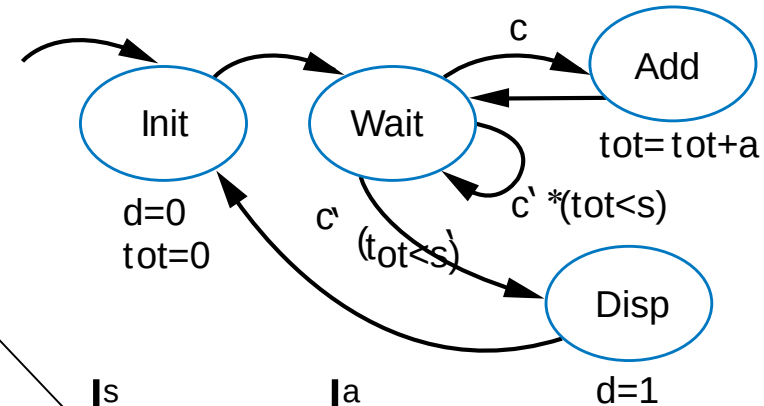
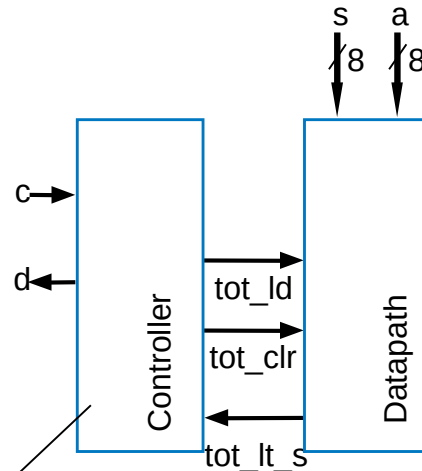
Remember the Soda HLSM?

Inputs: c (bit), a (8 bits), s (8 bits)

Outputs: d (bit)

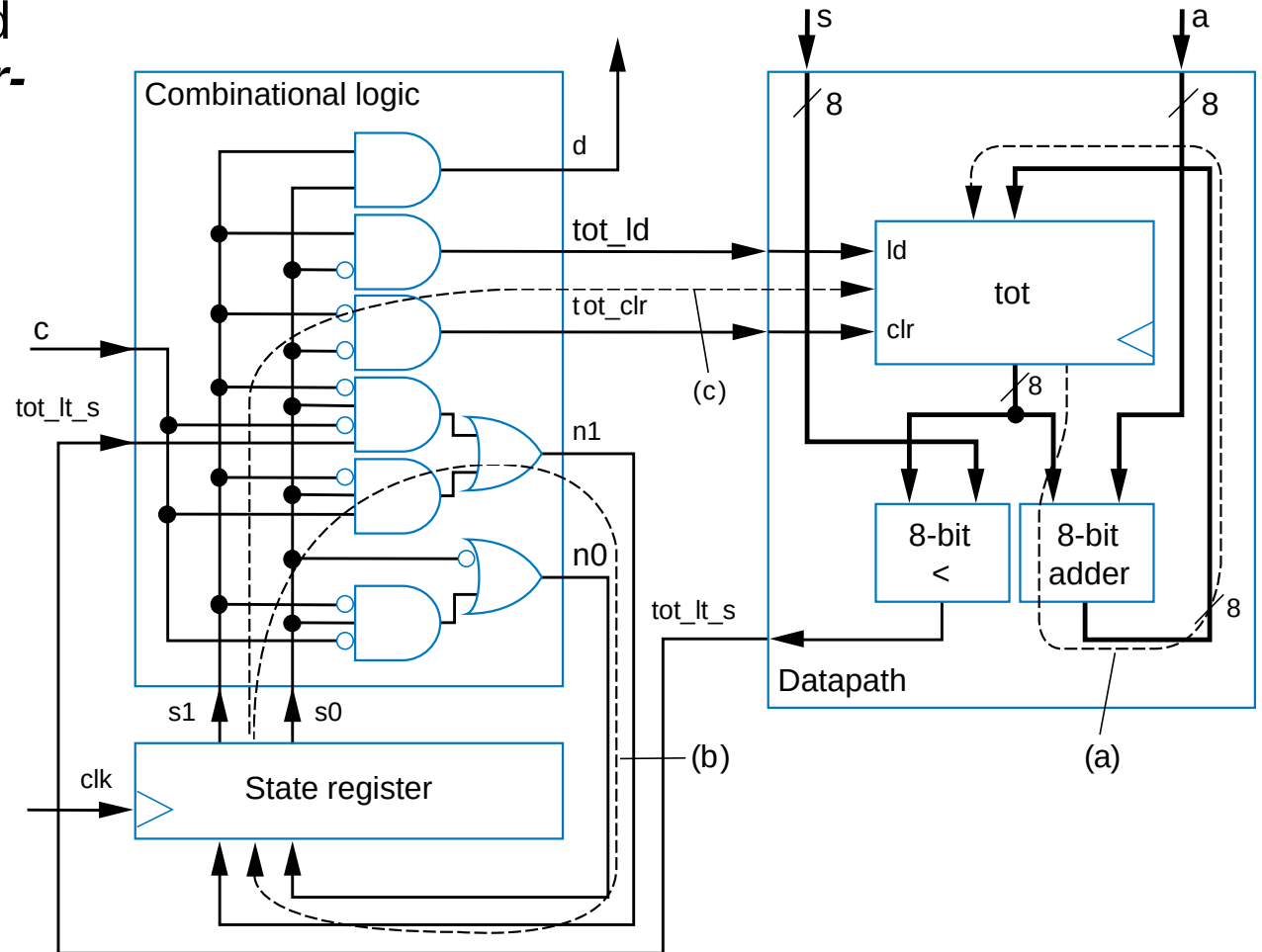
Local registers: tot (8 bits)

	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_clr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
...	...	...	...	...	...	...	...	...	...
Disp	1	1	0	0	0	0	1	0	0
...	...	...	...	...	...	...	...	...	...



RTL Delay: A Circuit May Have Numerous Paths

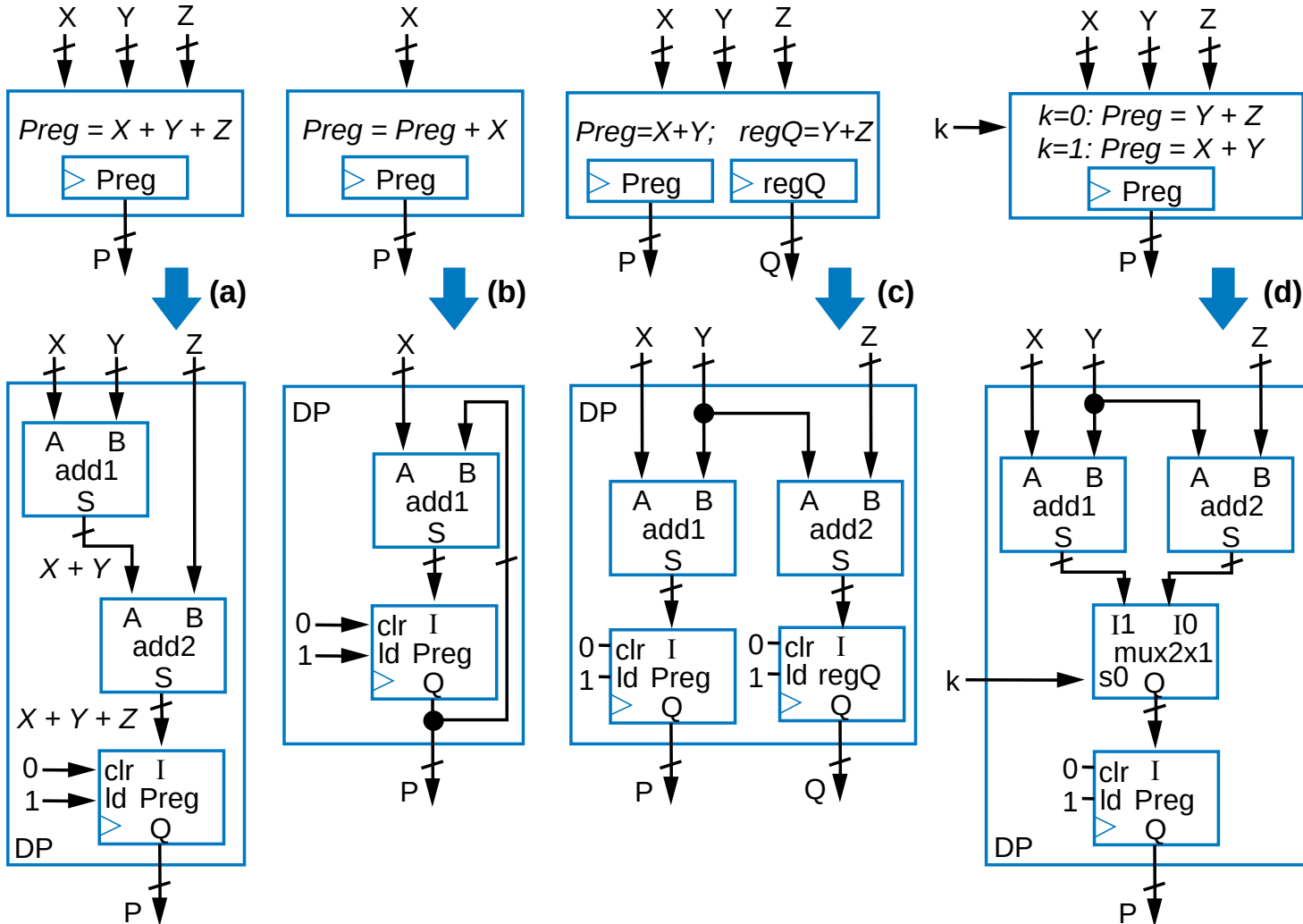
- Frequency is limited by **longest register-to-register delay**
 - **critical path**
- Paths can exist
 - In the datapath
 - In the controller
 - Between the controller and datapath
 - May be hundreds or thousands of paths



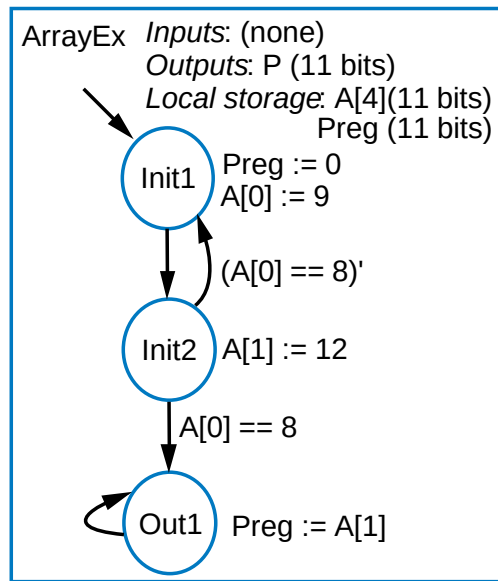
$$t_{ccq} + t_{cd} > t_{hold} + t_{skew}$$

$$T_c \geq t_{pcq} + t_{pd} + t_{setup} + t_{skew}$$

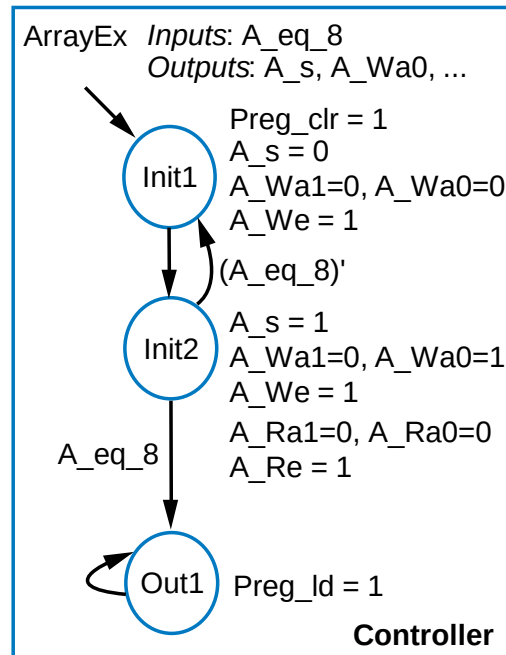
More Datapath Examples



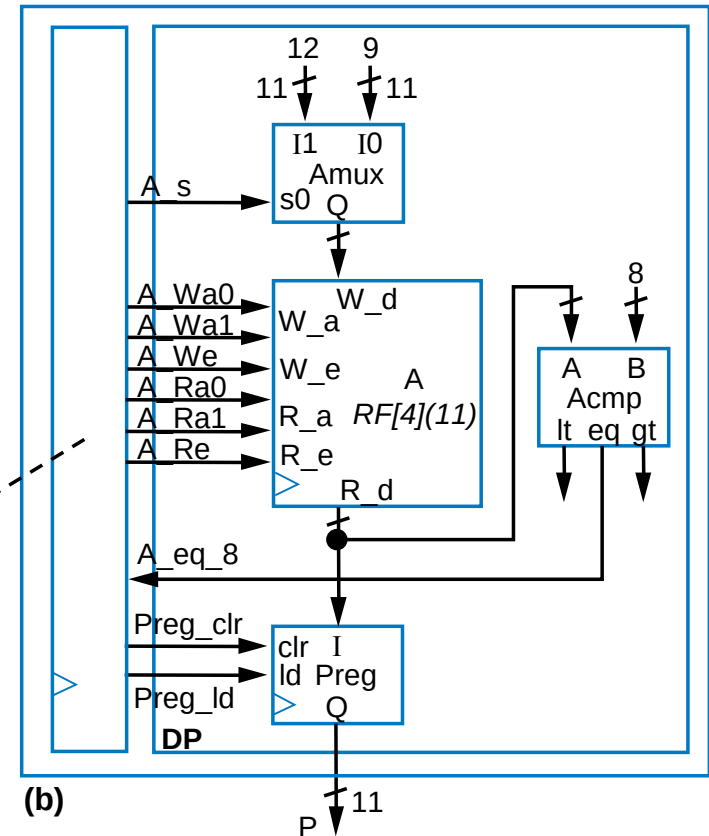
HLSM Array Example



(a)



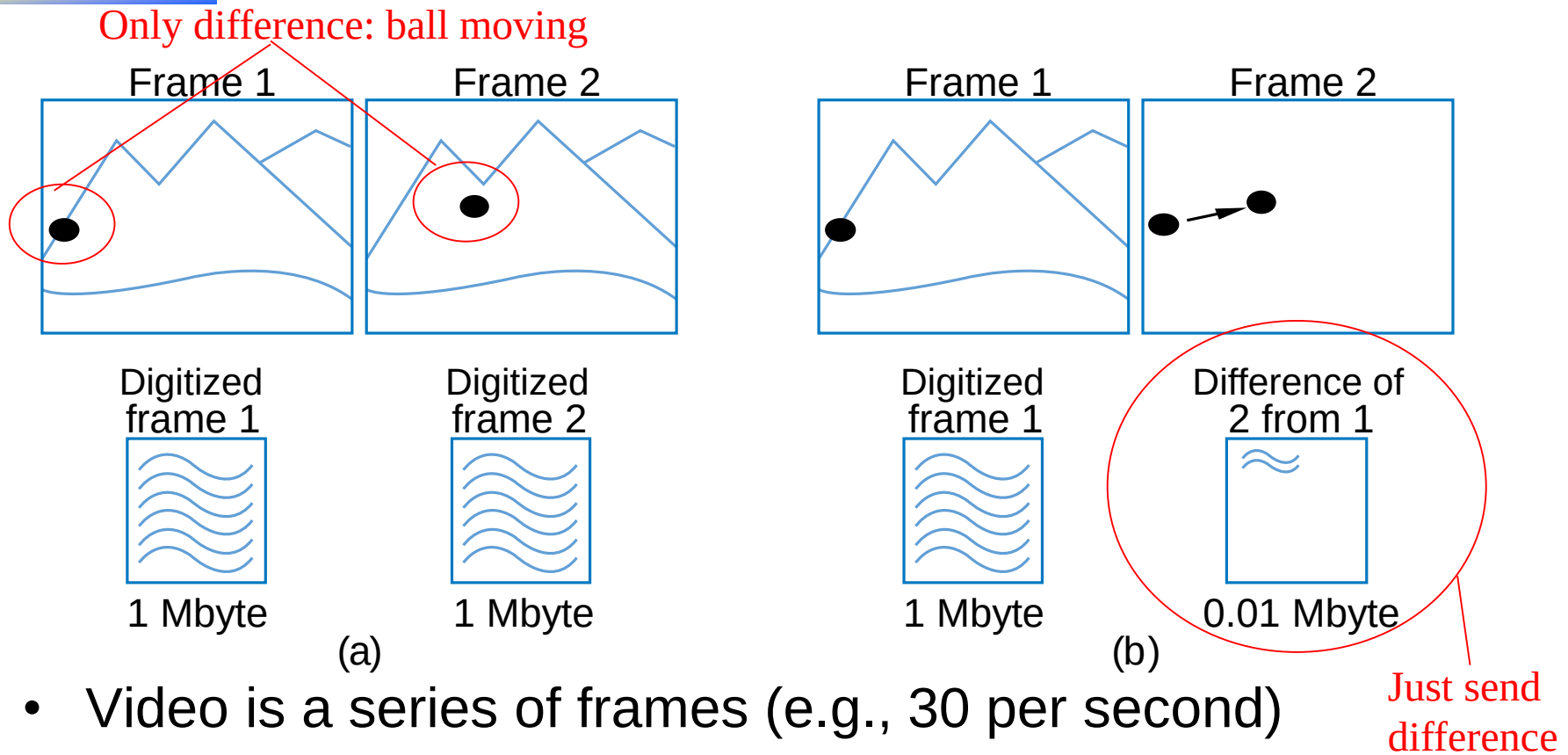
(c)





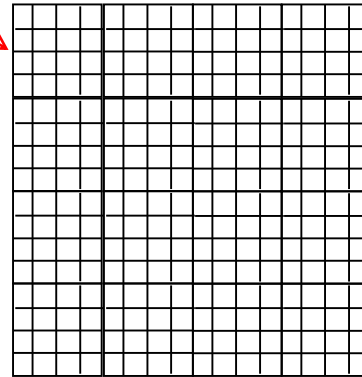
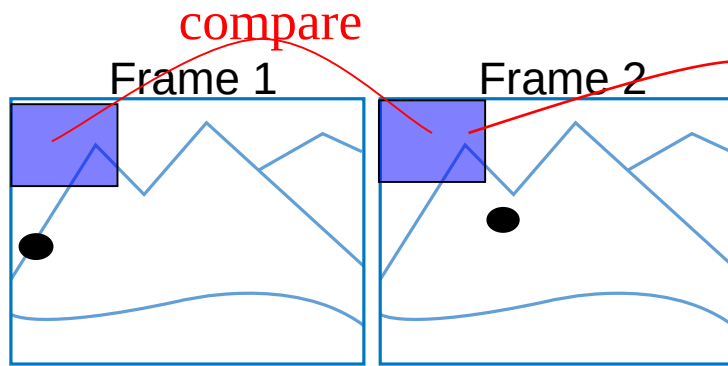
BREAK !

RTL Example: Video Compression



- Video is a series of frames (e.g., 30 per second)
- Most frames similar to previous frame
 - Compression idea: just send difference from previous frame

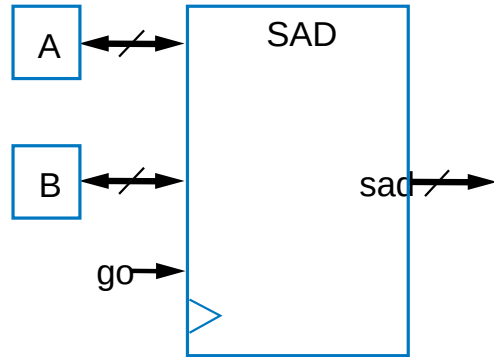
Video Compression – Sum of Absolute Differences



Each is a pixel, assume represented as 1 byte (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)

- If two frames are similar just send a difference instead
 - Compare corresponding 16x16 “blocks”
 - Treat 16x16 block as 256-byte array
 - Compute the absolute value of the difference of each array item
 - Sum the differences
 - If above a threshold, send a complete frame for second frame
 - Else send the difference

Sum-of-Absolute Differences: High-level FSM

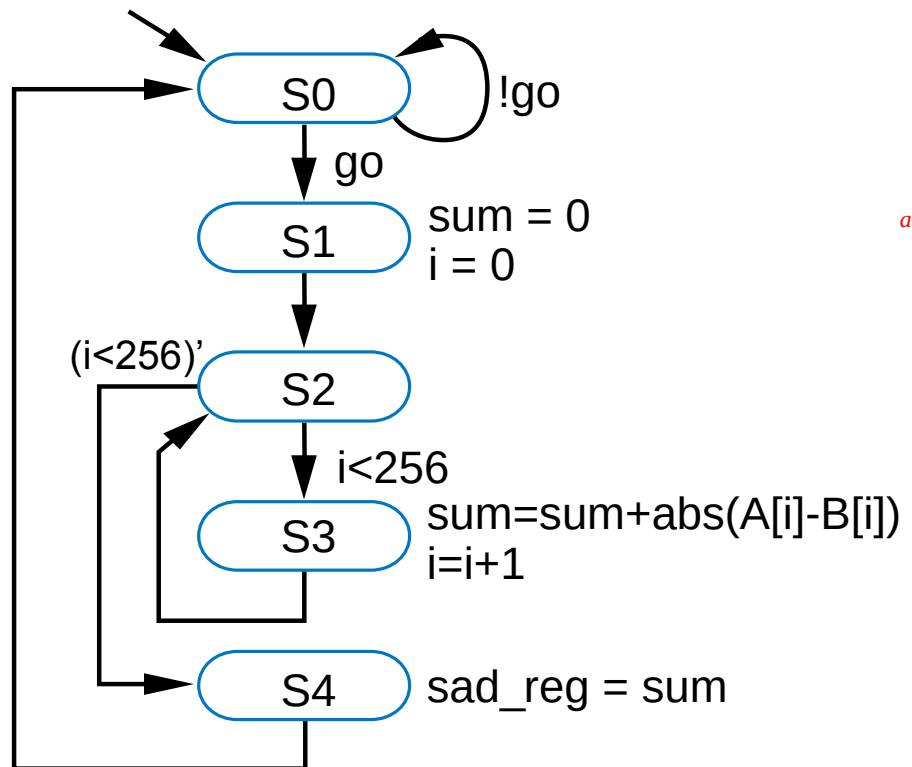


Inputs: A, B (256 byte memory); go (bit)

Outputs: sad (32 bits)

Local registers: sum, sad_reg (32 bits); i (9 bits)

- **S0**: wait for *go*
- **S1**: initialize *sum* and *index*
- **S2**: check if done ($i \geq 256$)
- **S3**: add difference to *sum*, increment index
- **S4**: done, write to output *sad_reg*

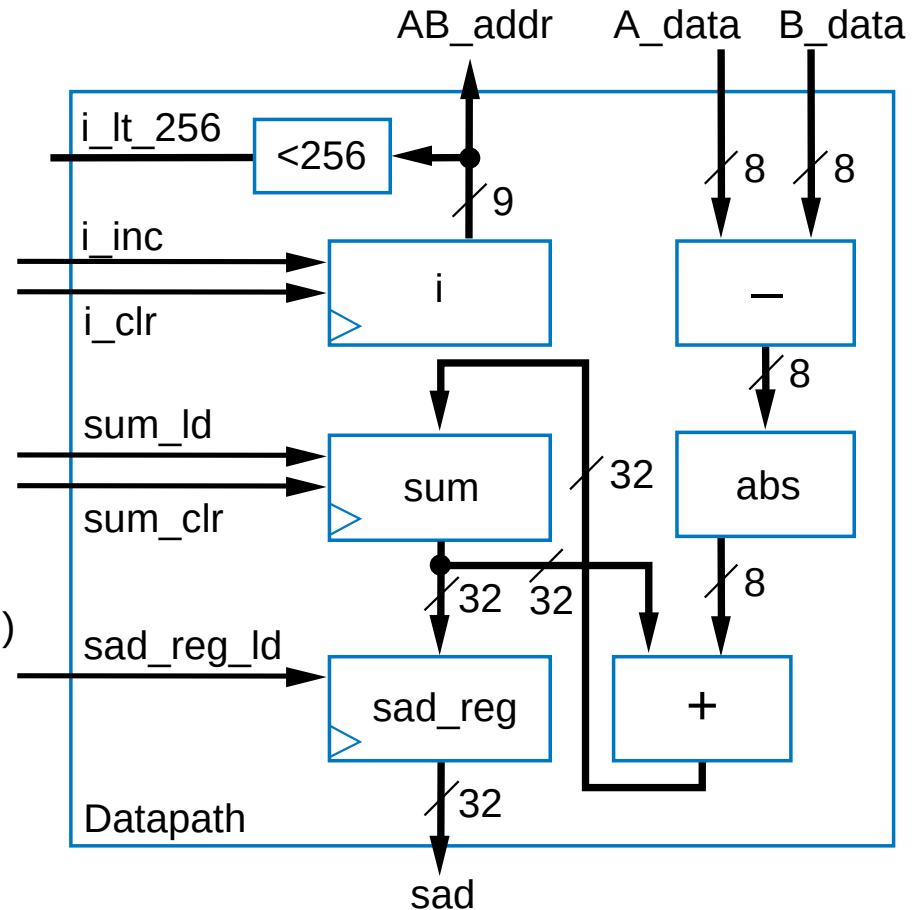
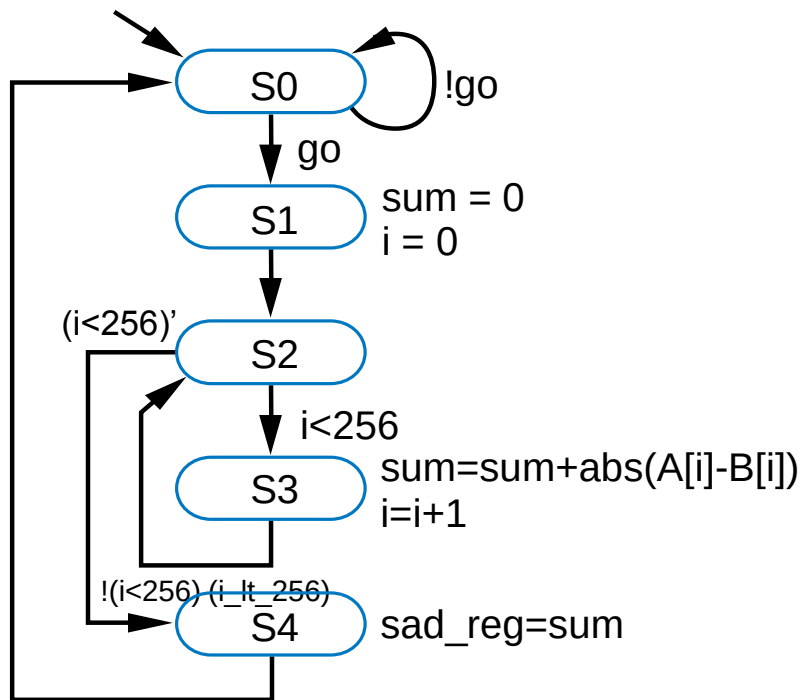


Sum-of-Absolute Differences: Datapath

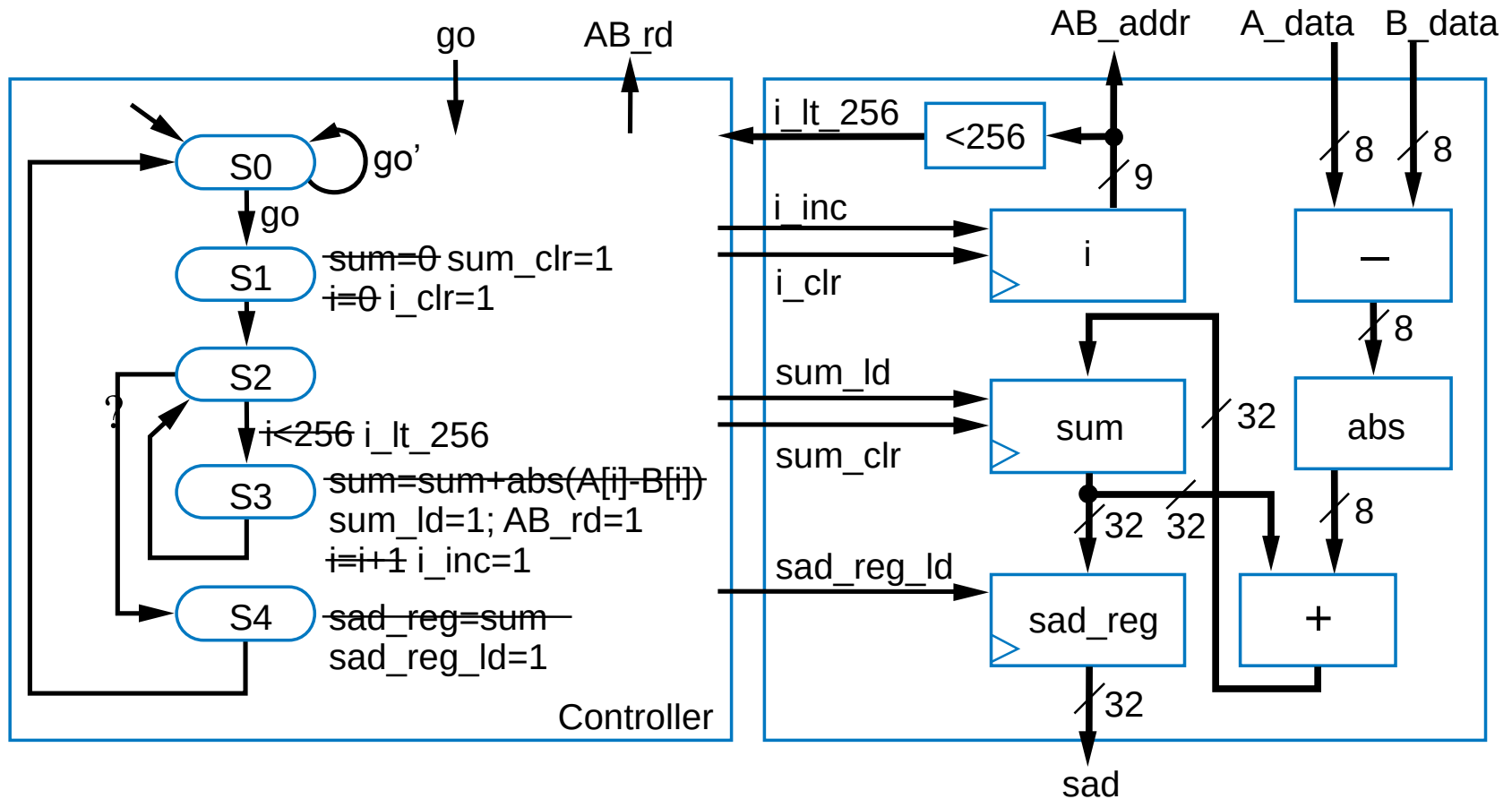
Inputs: A, B (256 byte memory); go (bit)

Outputs: sad (32 bits)

Local registers: sum, sad_reg (32 bits); i (9 bits)

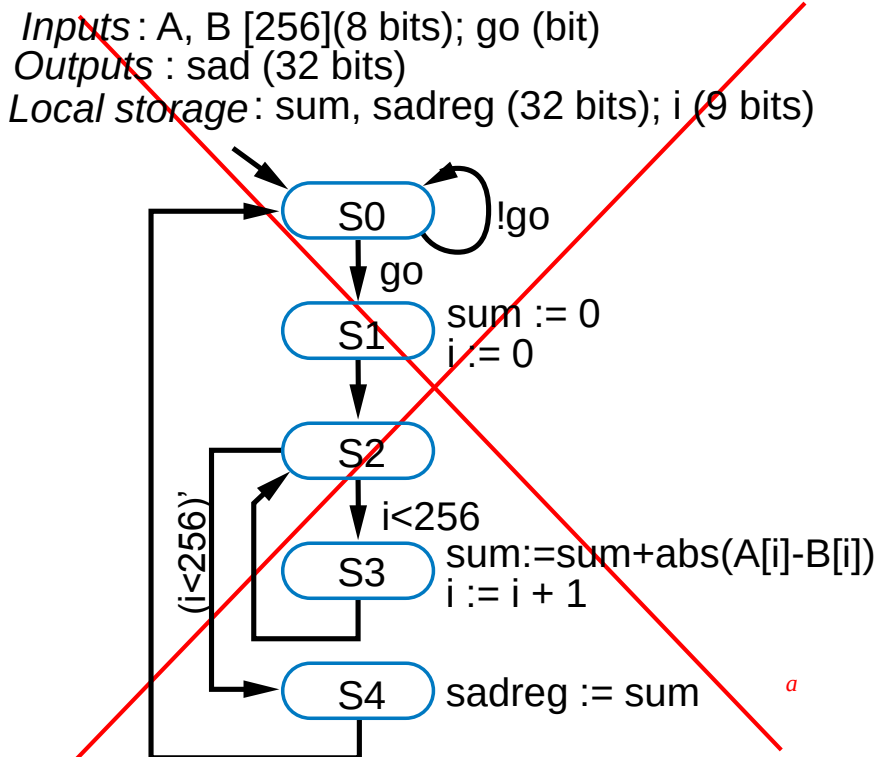


Sum-of-Absolute Differences: Connect datapath and controller & specify the controller FSM



Behavioral Level Design: C to Gates

- Earlier sum-of-absolute-differences example
 - Started with high-level state machine
 - C code is an even better starting point -- easier to understand

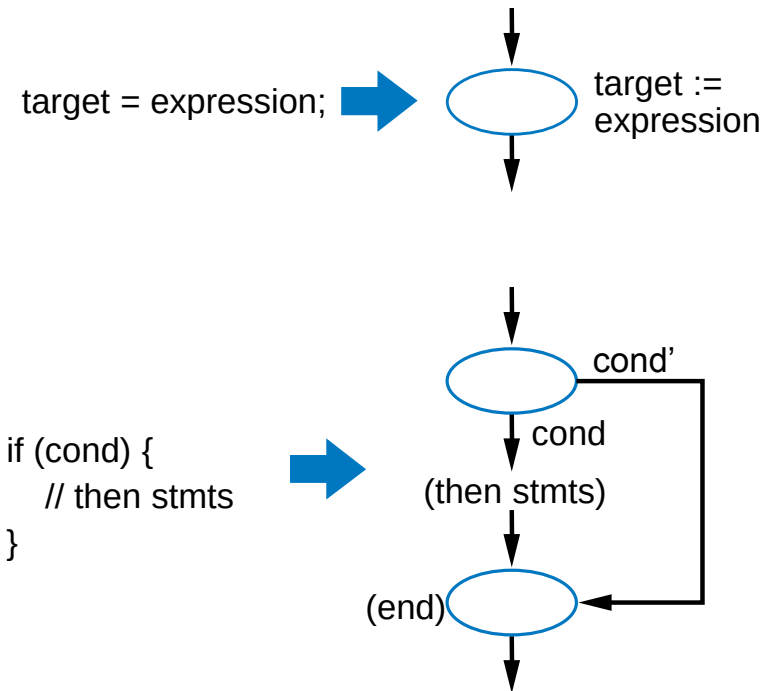


C code

```
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint I;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    return sum;
}
```

Converting from C to High-Level State Machine

- Convert each C construct to equivalent states and transitions
- *Assignment* statement
 - Becomes one state with assignment
- *If-then* statement
 - Becomes state with condition check, transitioning to “then” statements if condition true, otherwise to ending state
 - “then” statements would also be converted to states

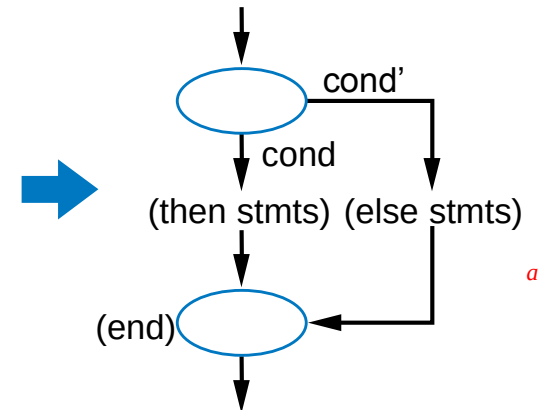


Converting from C to High-Level State Machine

- *If-then-else*

- Becomes state with condition check, transitioning to “then” statements if condition true, or to “else” statements if condition false

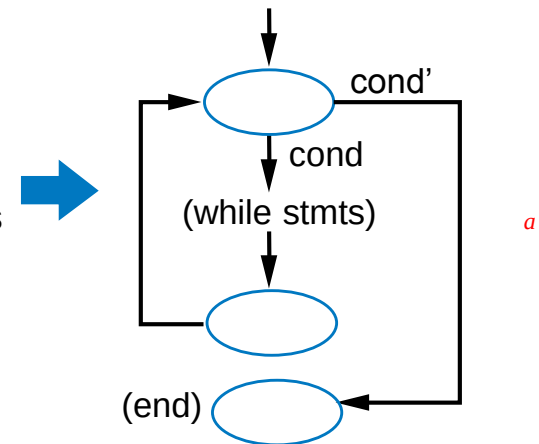
```
if (cond) {  
    // then stmts  
}  
else {  
    // else stmts  
}
```



- *While loop* statement

- Becomes state with condition check, transitioning to while loop's statements if true, then transitioning back to condition check

```
while (cond) {  
    // while stmts  
}
```



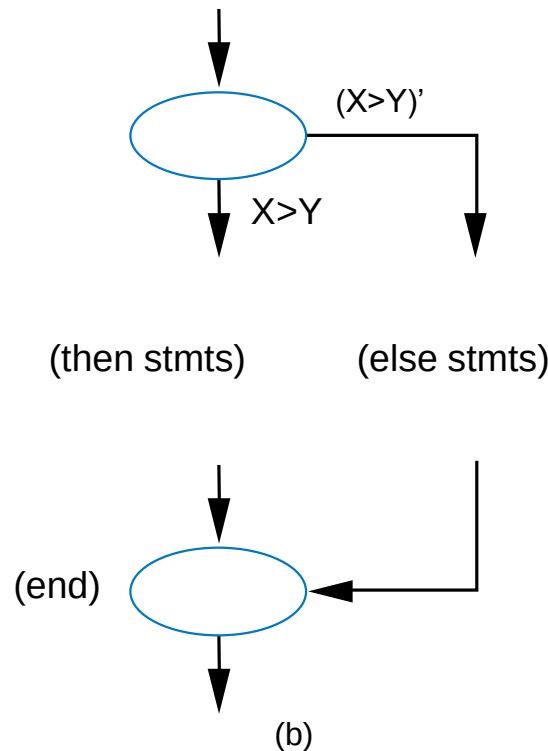
Converting from C to HLSM: Example

- Simple example: computing the maximum of two numbers
 - Convert if-then-else statement to states (b)
 - Then convert assignment statements to states (c)

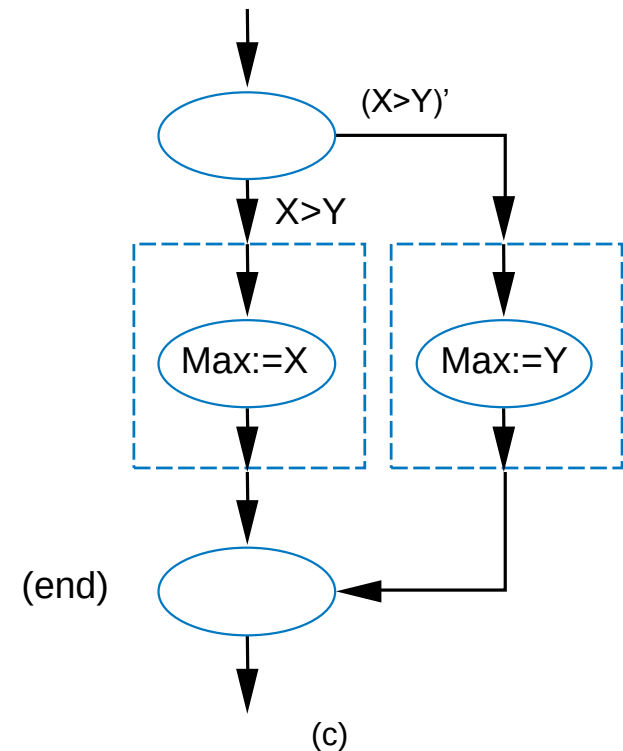
Inputs: uint X, Y
Outputs: uint Max

```
if (X > Y) {  
    Max = X;  
}  
else {  
    Max = Y;  
}
```

(a)



(b)



(c)

Example: SAD C code to HLSM

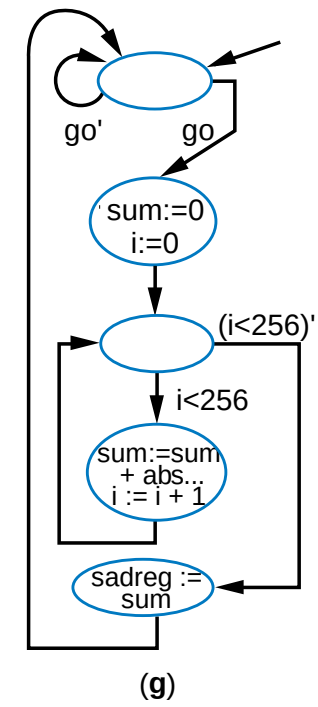
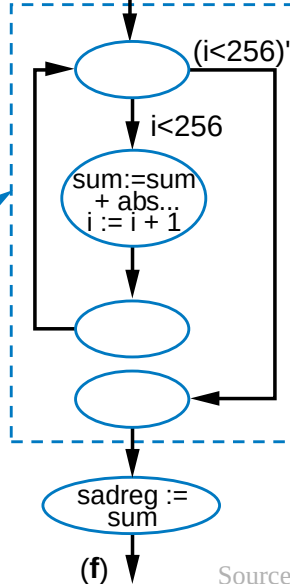
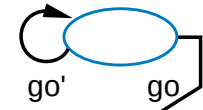
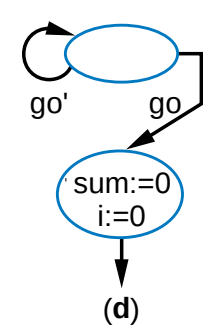
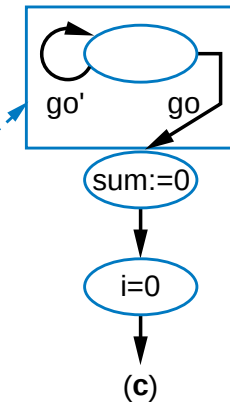
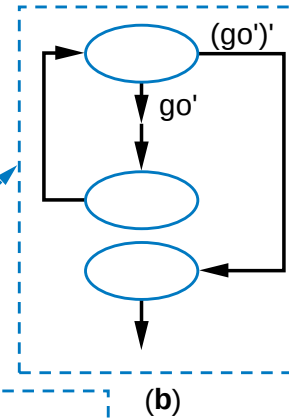
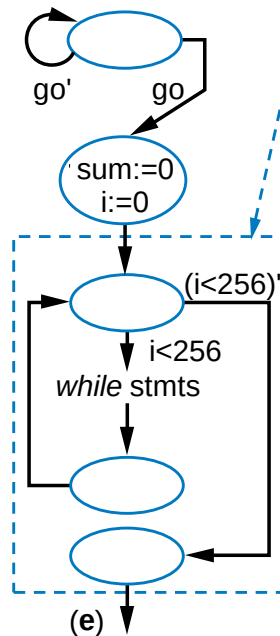
Inputs: byte A[256], B[256]
bit go;
Output: int sad
main()
{
 uint sum; short uint i;
 while (1) {

 while (!go);
 sum = 0;
 i = 0;

 while (i < 256) {
 sum = sum + abs(A[i] - B[i]);
 i = i + 1;
 }

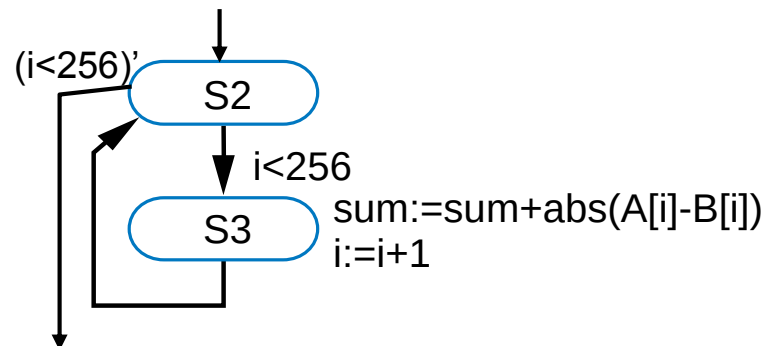
 sad = sum;
 }

} (a)



Circuit vs. Microprocessor

- **Circuit:**
 - Two states (**S2** & **S3**) for each i , 256 i 's $\rightarrow$ 512 clock cycles
- **Microprocessor:**
 - Loop (for $i = 1$ to 256), but for each i , must move memory to local registers, subtract, compute absolute value, add to sum, increment i
 - Each loop iteration is approx 6 cycles per array item $\rightarrow 256*6 = 1536$ cycles
- **Circuit is ~3 times (300%) faster**





BREAK !

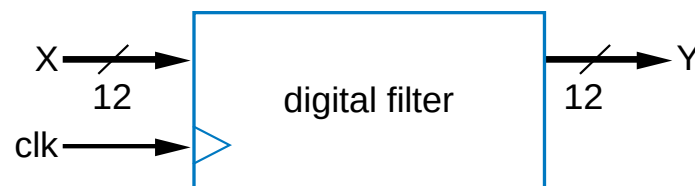
Data vs. Control Dominated RTL Design

- Data dominant design: extensive datapath, simple controller
- Control dominant design: complex controller, simple datapath

Example: Filter

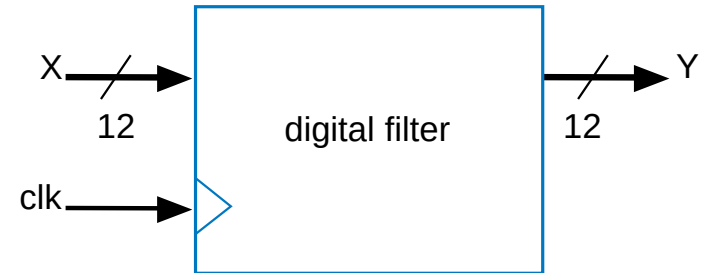
Converts digital input stream to new digital output stream

- e.g: remove noise
 - 180, 180, 181, 180, 240, 180, 181
 - 240 is probably noise, filter might replace by 181
- Simple filter: output average of the last N values
 - Small N : less filtering
 - Large N : more filtering, but less sharp output



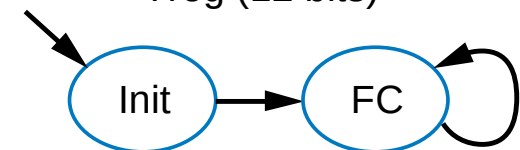
Data Dominated RTL Design Example: FIR Filter

- FIR filter
 - “Finite Impulse Response”
 - A configurable weighted sum of past input values
 - $y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$
 - Above known as “3 tap”
 - Tens of taps more common
 - Very general filter – User sets the constants (c0, c1, c2) to define a specific filter
- RTL design
 - Step 1: Create HLSM
 - Very simple states/transitions



$$y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$$

Inputs: X (12 bits) Outputs: Y (12 bits)
Local storage: xt0, xt1, xt2, c0, c1, c2 (12 bits);
Yreg (12 bits)



	Yreg := 0	Yreg :=
	xt0 := 0	c0*xt0 +
	xt1 := 0	c1*xt1 +
	xt2 := 0	c2*xt2
	c0 := 3	xt0 := X
	c1 := 2	xt1 := xt0
FIR filter	c2 := 2	xt2 := xt1

FC: FIR Compute

Assumes constants set to 3, 2, and 2

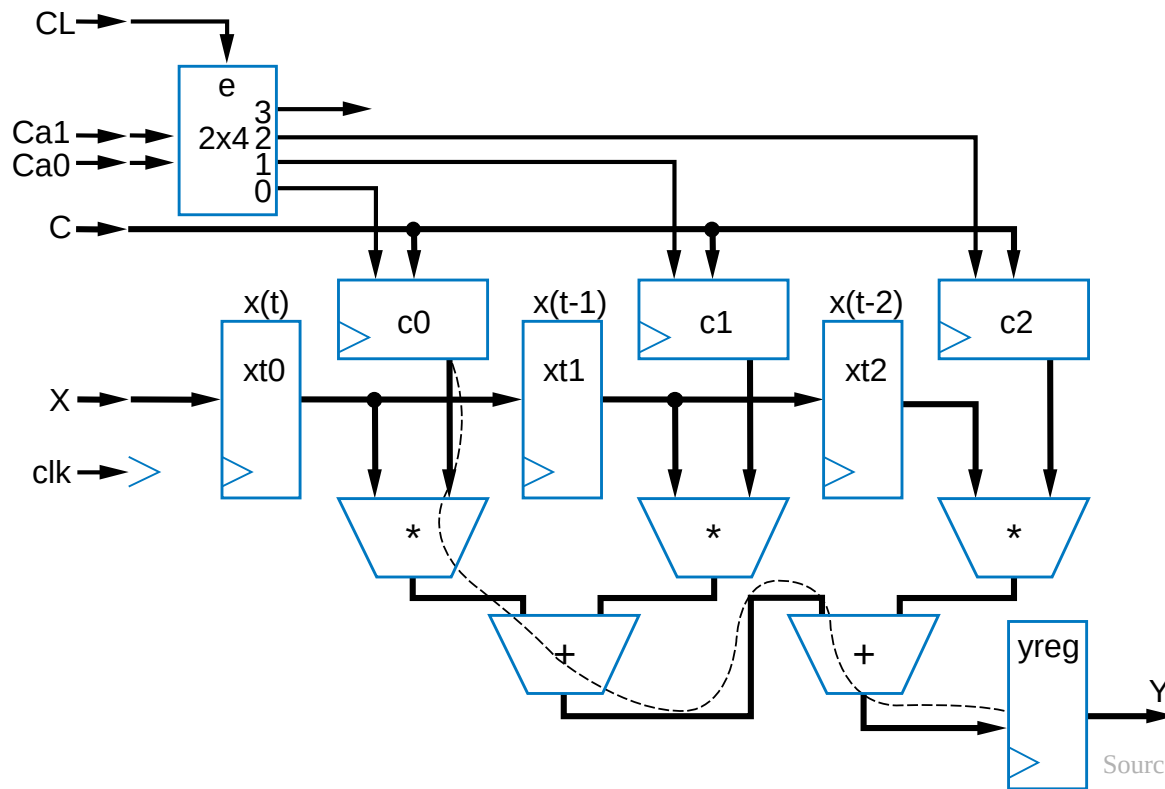
FIR Filter: Create datapath

Step 2: Create datapath

- Create a chain of xt registers to hold past values of x
- Instantiate registers for c0, c1, c2
- Instantiate multipliers to compute c*x values
- Instantiate adders
- Add circuitry to allow loading of c register



$$y(t) = c0 \cdot x(t) + c1 \cdot x(t-1) + c2 \cdot x(t-2)$$

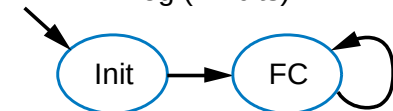


Steps 3 & 4:
Connect to
controller, &
create FSM:
**No controller
needed!**

FIR Filter: Design the Circuit

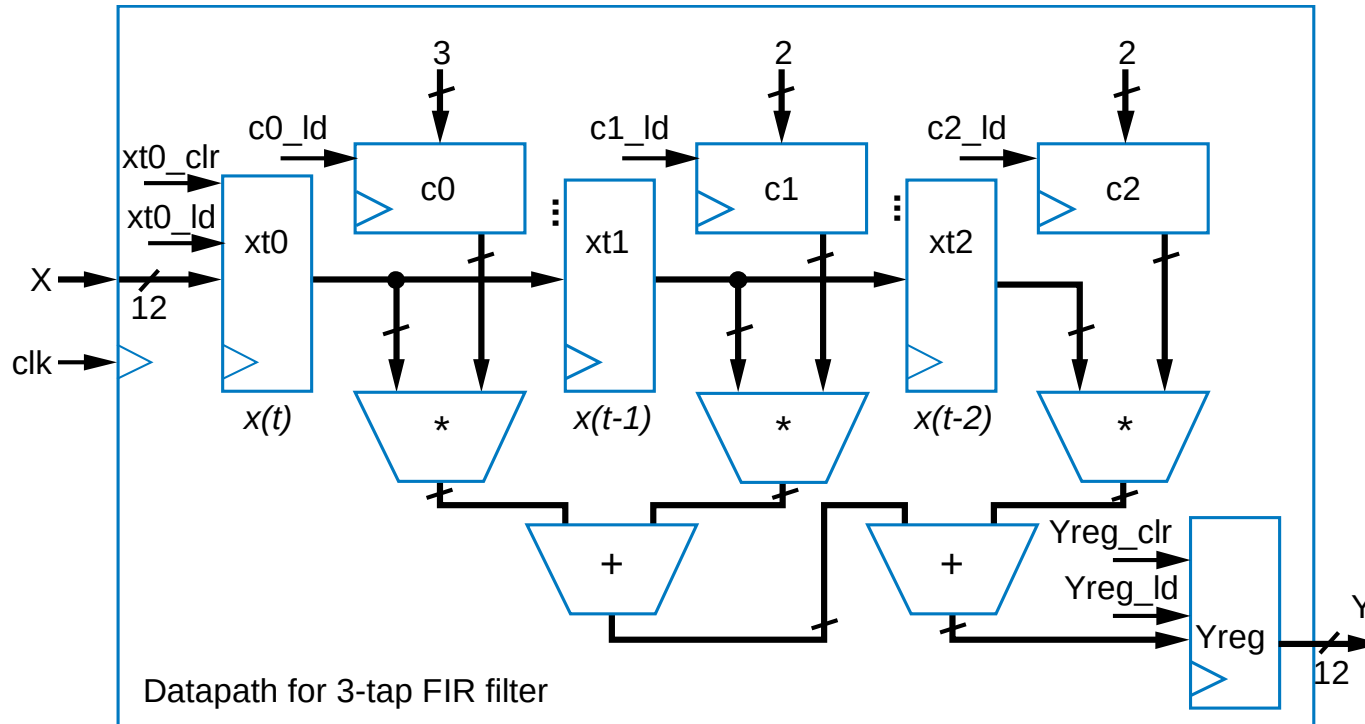
- Create datapath
- Connect control and datapath
- Derive FSM
 - Set clr and ld lines appropriately

Inputs: X (12 bits) Outputs: Y (12 bits)
 Local storage: xt0, xt1, xt2, c0, c1, c2 (12 bits);
 Yreg (12 bits)



Yreg := 0	Yreg :=
xt0 := 0	c0*xt0 +
xt1 := 0	c1*xt1 +
xt2 := 0	c2*xt2
c0 := 3	xt0 := X
c1 := 2	xt1 := xt0
c2 := 2	xt2 := xt1

FIR filter



Comparing the FIR circuit to a software implementation

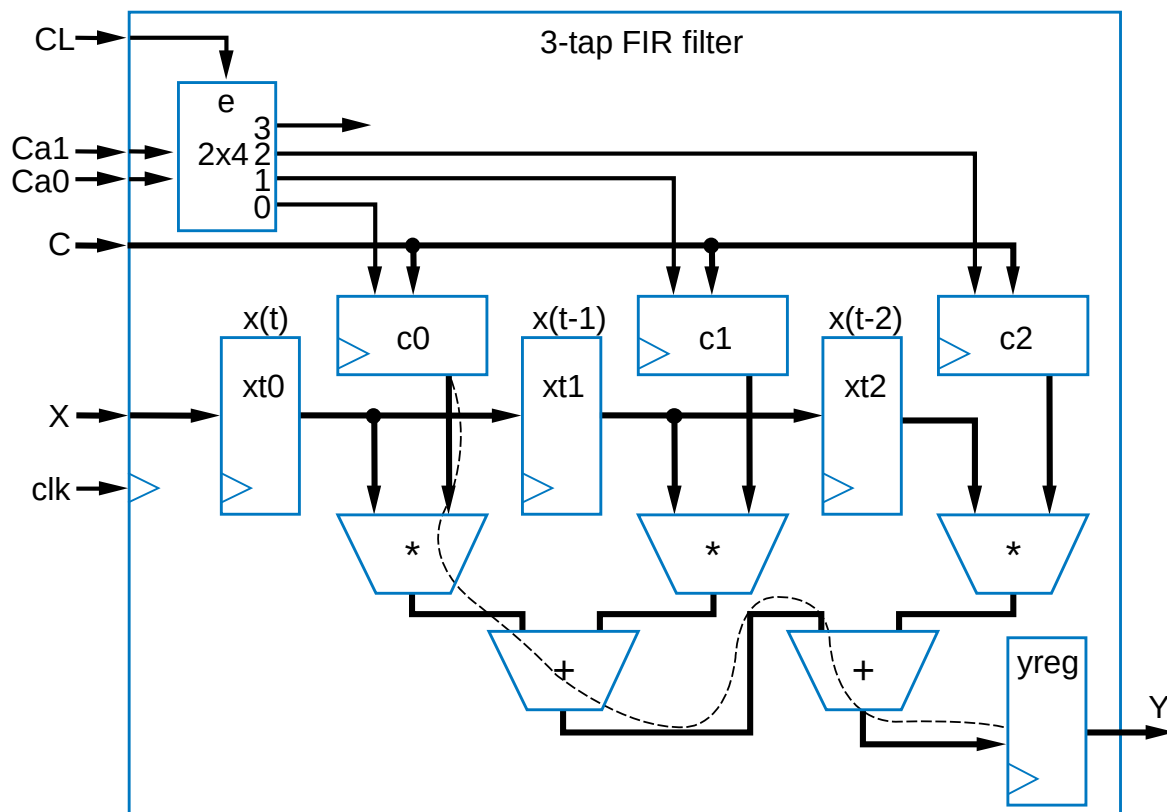
$$y(t) = c_0 \cdot x(t) + c_1 \cdot x(t-1) + c_2 \cdot x(t-2)$$

- Circuit

- Adder has 2-gate delay, multiplier has 20-gate delay
- Longest path goes through one multiplier and two adders
 - $20 + 2 + 2 = 24$ -gate delay
- **100-tap filter has 34-gate delay:** 1 multiplier & 7 adders on longest path

- Software

- 100-tap filter: 100 multiplications, 100 additions.
- If 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
- $(100 \cdot 2 + 100 \cdot 2) \cdot 10 =$
4000 gate delays



Summary

- RTL design:
 - HLSM diagram and differences w.r.t. FSM
 - Datapath and datapath components
 - Connect datapath to controller
 - Derive controller FSM
 - RTL circuit delays and clock
 - HLSM array and Register Files
 - HLSM and C-code
 - Data-dominated Vs Control-dominated RTL design