CS152 – Computer Architecture and Engineering

Lecture 8 – Logic Review

2003-09-18

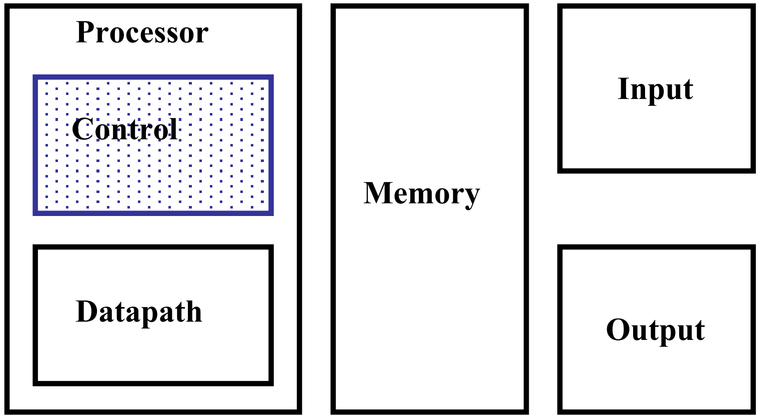
Dave Patterson

(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs152/

Review

- Single cycle datapath => CPI=1, CCT => long
- 5 steps to design a processor
  - 1. Analyze instruction set => datapath requirements
  - 2. Select set of datapath components & establish clock methodology
  - 3. Assemble datapath meeting the requirements
  - 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  - 5. Assemble the control logic
- Control is the hard part
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates

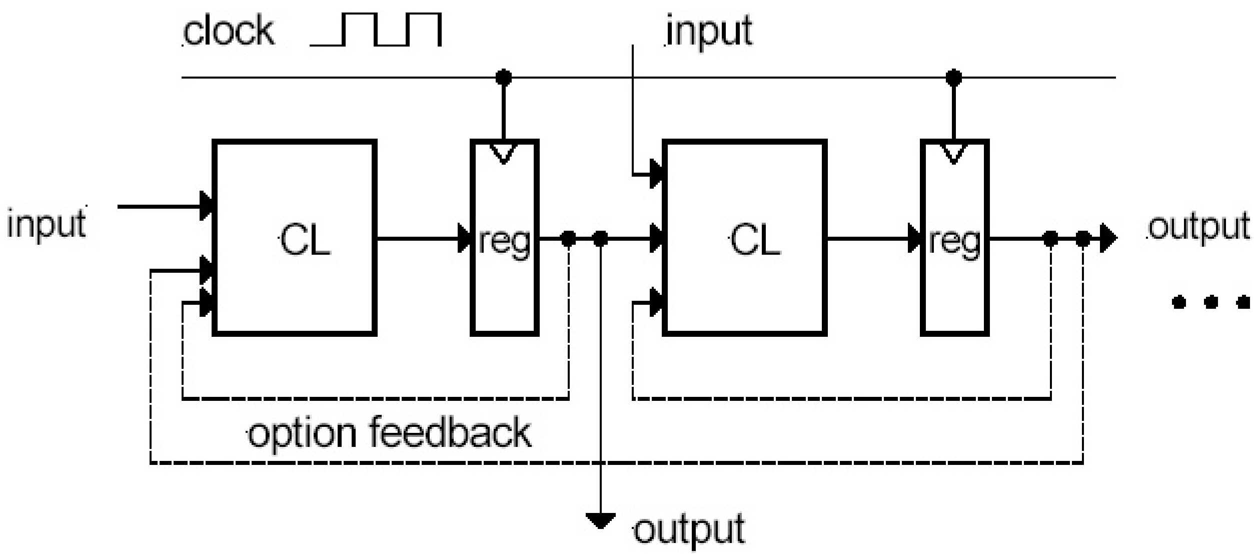


- On-line Design Notebook
  - Open a window and keep an editor running while you work;cut&paste
  - Former CS 152 students (and TAs) say they use on-line notebook for programming as well as hardware design; one of most valuable skills
  - Refer to the handout as an example

Outline

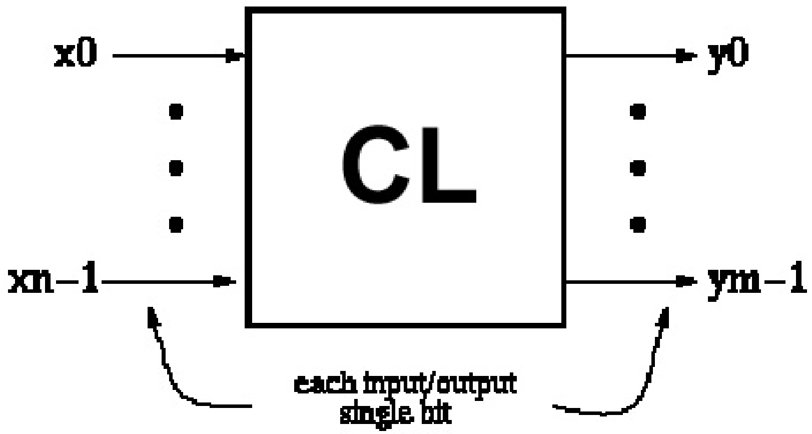
- Combinational Logic
  + Verilog Representation
- FlipFlops, Registers
  + Verilog Representation
- FSM + Verilog Representation
- Implementation of Control
  Representation of Single Cycle Datapath

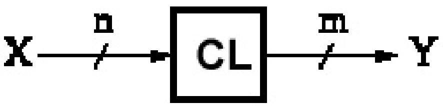
General Model of a Synchronous Circuit



- All wires, except clock, may be multiple bits wide.
- Registers (reg)
  - collections of flip-flops
- clock
  - distributed to all flip-flops
- Combinational Logic Blocks (CL)
  - no internal state
  - output only a function of inputs
- Particular inputs/outputs are optional
- Optional Feedback

Combinational Logic (CL) Defined





$y_i = f_i(x0 , ... , xn\text{-}1)$, where x, y are {0,1}.

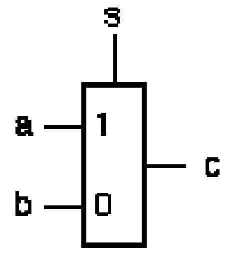
Y is a function of only X.

- If we change X, Y will change immediately (well almost!).
- There is an implementation dependent delay from X to Y.

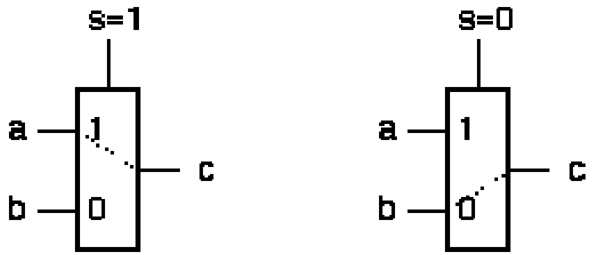
Some Combinational Logic Pieces

- OR gates
- AND gates
- NAND gates
- NOR gates
- XOR gates
- XNOR gates
- NOT gates
- Etc….

- Multiplexor



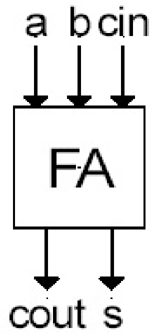
If s=1 then c=a else c=b



Verilog Representation

Cout = a · b + b · cin + a · cin

S = a xor b xor c



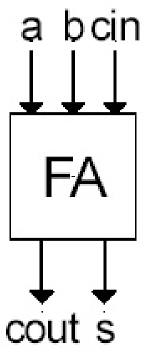

```
module FA(a, b, cin, cout, s)
    Input a, b, cin;
    Output cout, s;
    wire cout, s;
    assign cout = a & b + b & cin + a & cin;
    assign s = a ^ b ^ cin;
endmodule
```

- **Why are cout and s wires?**
- **Notice there is no clock in this code**
- **Why use "assign … =" ?**

Alternative Verilog Representation

$$Cout = a \cdot b + b \cdot cin + a \cdot cin$$

$$S = a \text{ xor } b \text{ xor } c$$




```
module FA(a, b, cin, cout, s)
Input a, b, cin;
output cout, s;
Reg cout, s;
always@(a or b or cin) begin
        if ( (a && b) || (b && cin) || (a && cin) )
                cout <= 1;
        else
                cout <=0;
        s <= a ^ b ^ cin;
    end
endmodule
```

- **What's the difference between && and &?**
- **Why are cout and s declared as reg now?**
- **Notice there is no clock in this code either**

Combinational Always Blocks

- Make sure all signals assigned in a combinational always block are explicitly assigned values *every* time that the always block executes. Otherwise latches will be generated to hold the last value for the signals not assigned values.

- Example:
  - Sel case value 2'd2 omitted.
  - Out is not updated when select line has 2'd2.
  - Latch is added by tool to hold the last value of out under this condition.

```
module mux4to1 (out, a, b, c, d, sel);
output out;
input a, b, c, d;
input [1:0] sel;
reg out;
always @(sel or a or b or c or d)
begin
  case (sel)
    2'd0: out = a;
    2'd1: out = b;
    2'd3: out = d;
  endcase
end
endmodule
```

Combinational Always Blocks

- To avoid synthesizing a latch in this case, add the missing select line:

```
2'd2: out = c;
```

- Or, in general, use the "default" case:

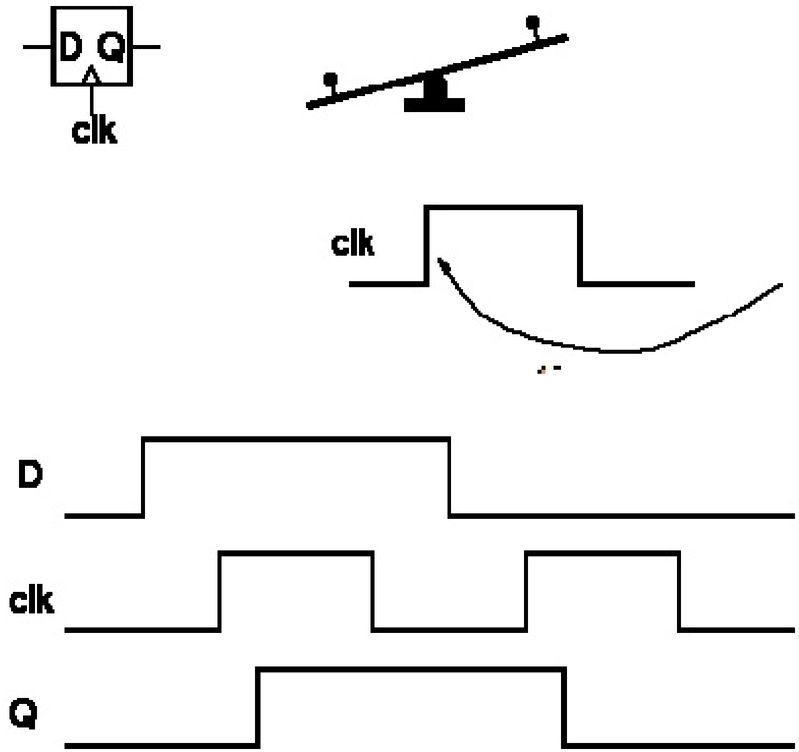
```
default:  out = foo;
```

- If you don't care about the assignment in a case (for instance you know that it will never come up) then assign the value "x" to the variable. Example:

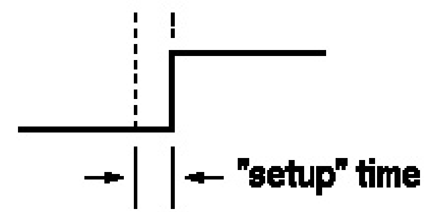
```
default:  out = 1'bx;
```

  The x is treated as a "don't care" for synthesis and will simplify the logic.

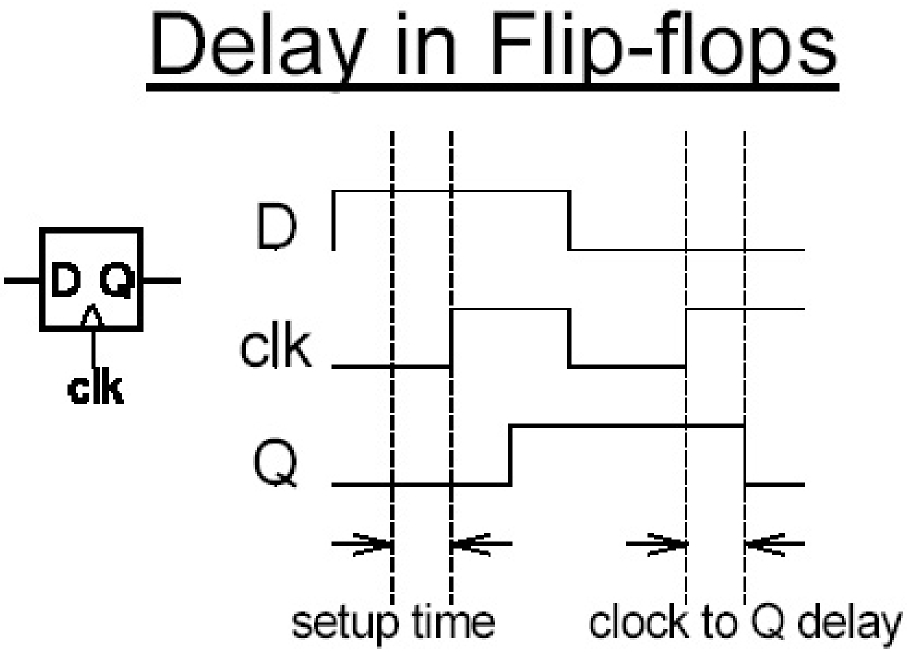
D-type edge-triggered flip-flop



- The edge of the clock is used to *sample* the "D" input & send it to "Q" (positive edge triggering).
  - At all other times the output Q is independent of the input D (just stores previously sampled value).
  - The input must be stable for a short time before the clock edge.

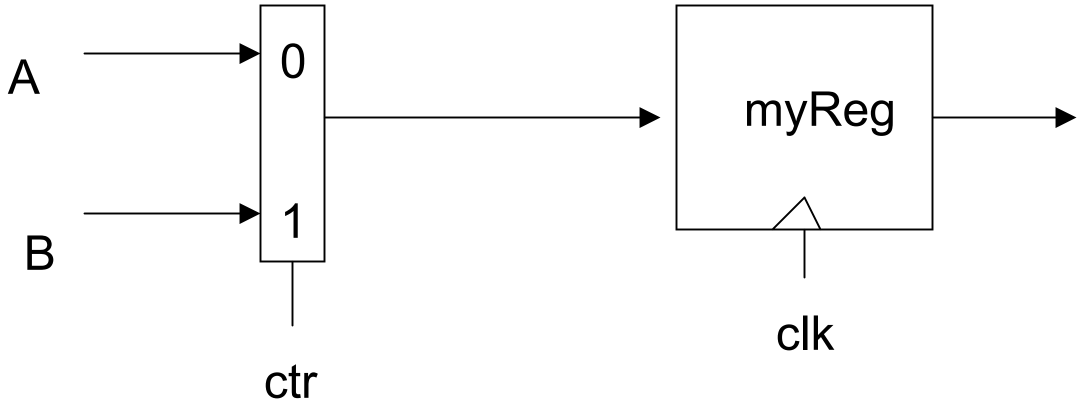


FF timing revisited



- On the posedge, the flip flop will latch in the value sitting on the D side of the flip flop, provided that the input has been stable for the “setup time”
- The actual output of the FF (the “Q”) doesn’t change until after the clock to Q delay.

Verilog representations of FFs




```
reg myReg;
always @(posedge clk) begin
        if (ctr == 1)
                myReg <= B;
        else
                myReg <= A;
end
endmodule
```

- **myReg is declared as a `reg`**
- **Notice that the only thing in the sensitivity list is "`posedge clk`". This creates a register**

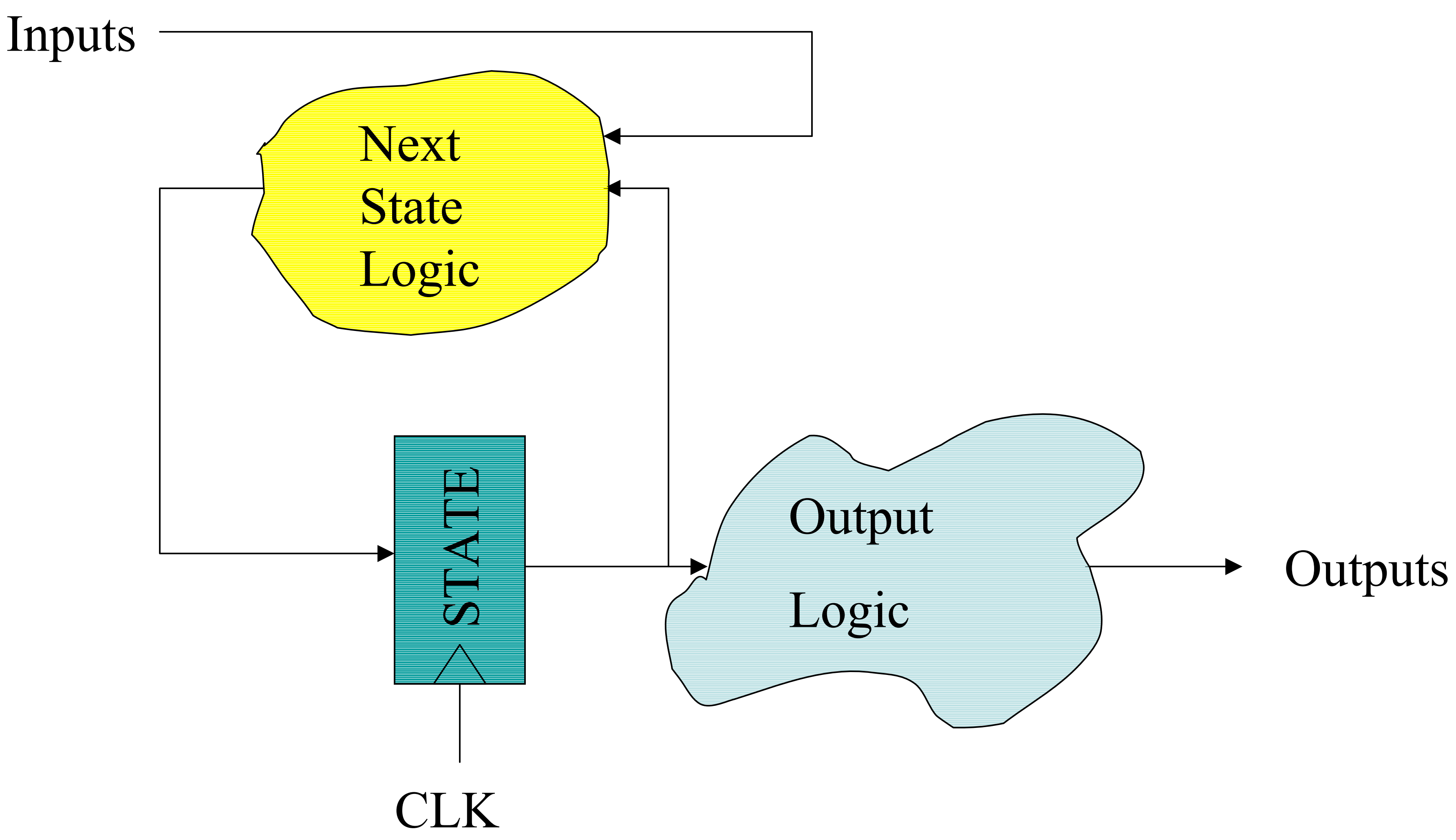
Administrivia

- Office hours in Lab
  - Mon 5 – 6:30 Jack, Tue 3:30-5 Kurt, Wed 3 – 4:30 John, Thu 3:30-5 Ben
- Dave’s office hours Tue 3:30 – 5
- Lab 3 up
- Reading: Sections 5.6, 5.8, 5.11, 5.12 in Beta ed.
- T.A. Review Design Document with groups in discussion section Friday
- Design Document: Email to TA today 9/17

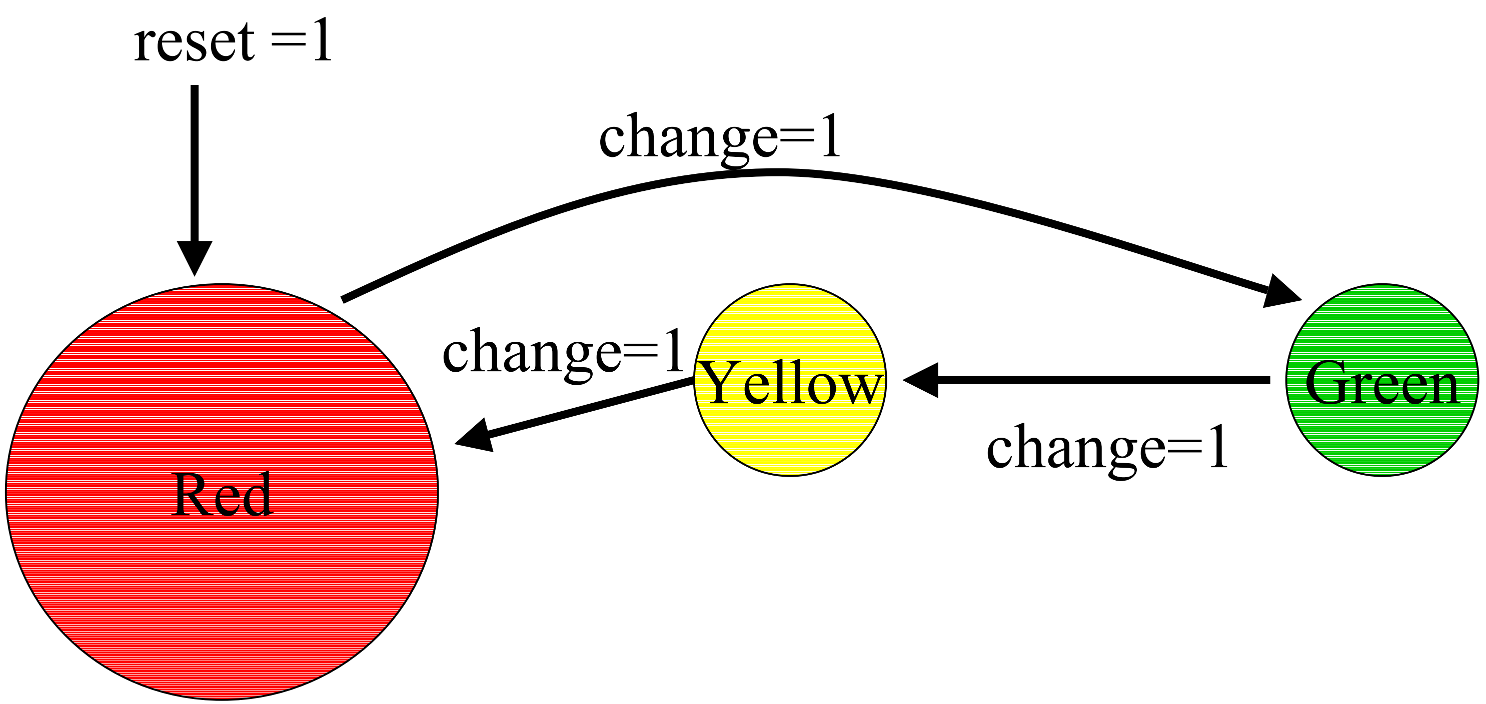
Administrivia: Design Document

- Design Document: 2-4 pages for Lab 3
- Should include:
  - Defining the Problem (makes sure you understand the problem)
  - Approach / Plan of Attack
  - Division of Labor
  - Timeline
  - Basic block diagrams of datapath pieces
  - Skeleton Verilog modules
  - Plan of testing

FSM structure



FSM description



Verilog Code – FSM State Register

```
module trafficLightFSM(clk,reset,change,red,yellow,green);
       input clk,reset,change;
       output red,yellow,green;
       reg red,yellow,green;
       reg [1:0] curState,nextState;
       parameter showRed = 2'b00, showYellow = 2'b01;
       parameter showGreen = 2'b10;
```

```
// state register
always @(posedge clk)
       if (reset==1'b1) curState <= 2'b0;
       else curState <= nextState;
```

FSM Next State Logic

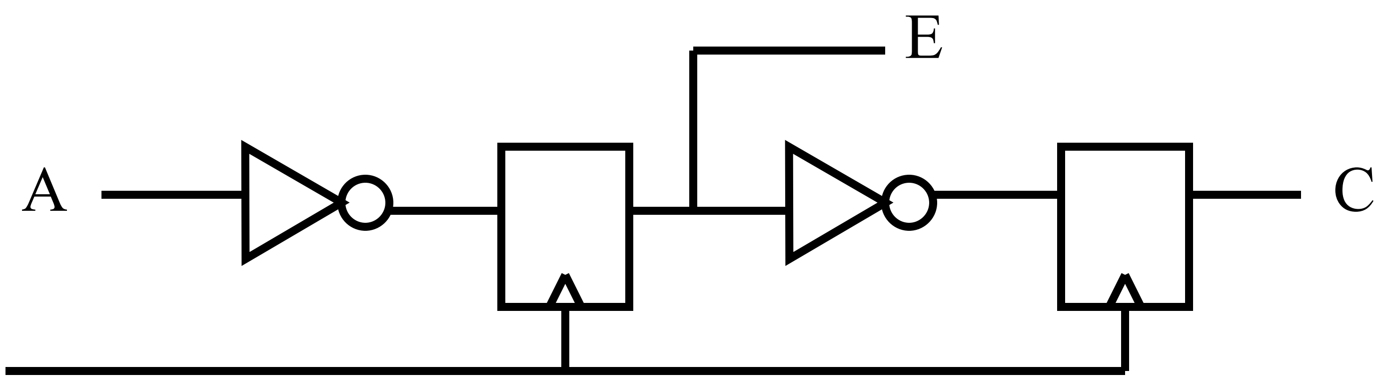
```
// next state logic
// dependent only on the current state and input
always @(curState or change)
begin
      nextState = showRed; // default state
      case (curState)
            showRed: if (change) nextState=showGreen;
            showYellow: if (change) nextState=showRed;
                  else nextState=showYellow;
            showGreen: if (change) nextState=showYellow;
                  else nextState=showGreen;
      endcase
end
```

FSM Output Logic

```
// Output Logic: dependent ONLY on state
always @(curState) begin
    // *ALWAYS* put default output values
    red=1'b0; green=1'b0; yellow=1'b0;
    case (curState)
        showRed: red=1'b1;
        showYellow: yellow=1'b1;
        showGreen: green=1'b1;
    endcase
end
endmodule
```

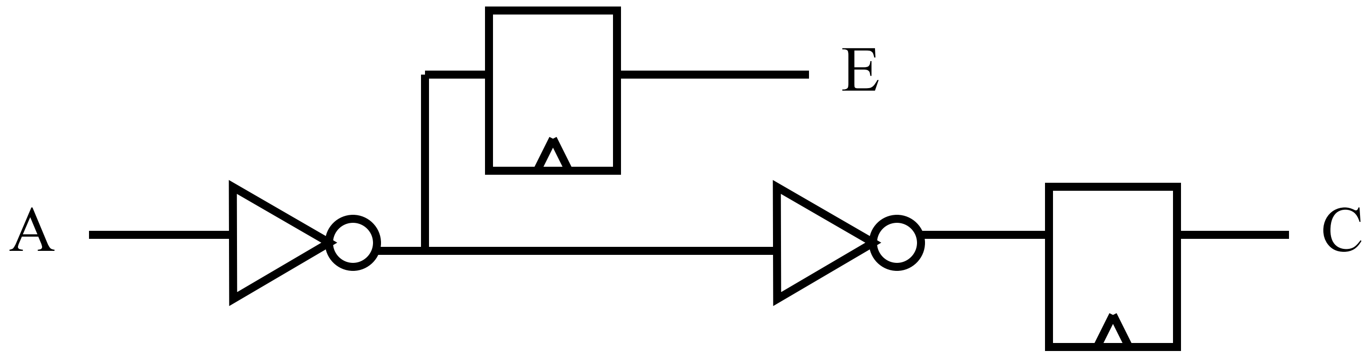
Verilog: Non-Blocking? Assignments

```
always @(posedge clk)
  begin
      E <= ~A;
      C <= ~E;
  end
```



Verilog: Blocking? Assignments

```
always @(posedge clk)
  begin
      E = ~A;
      C = ~E;
  end
```

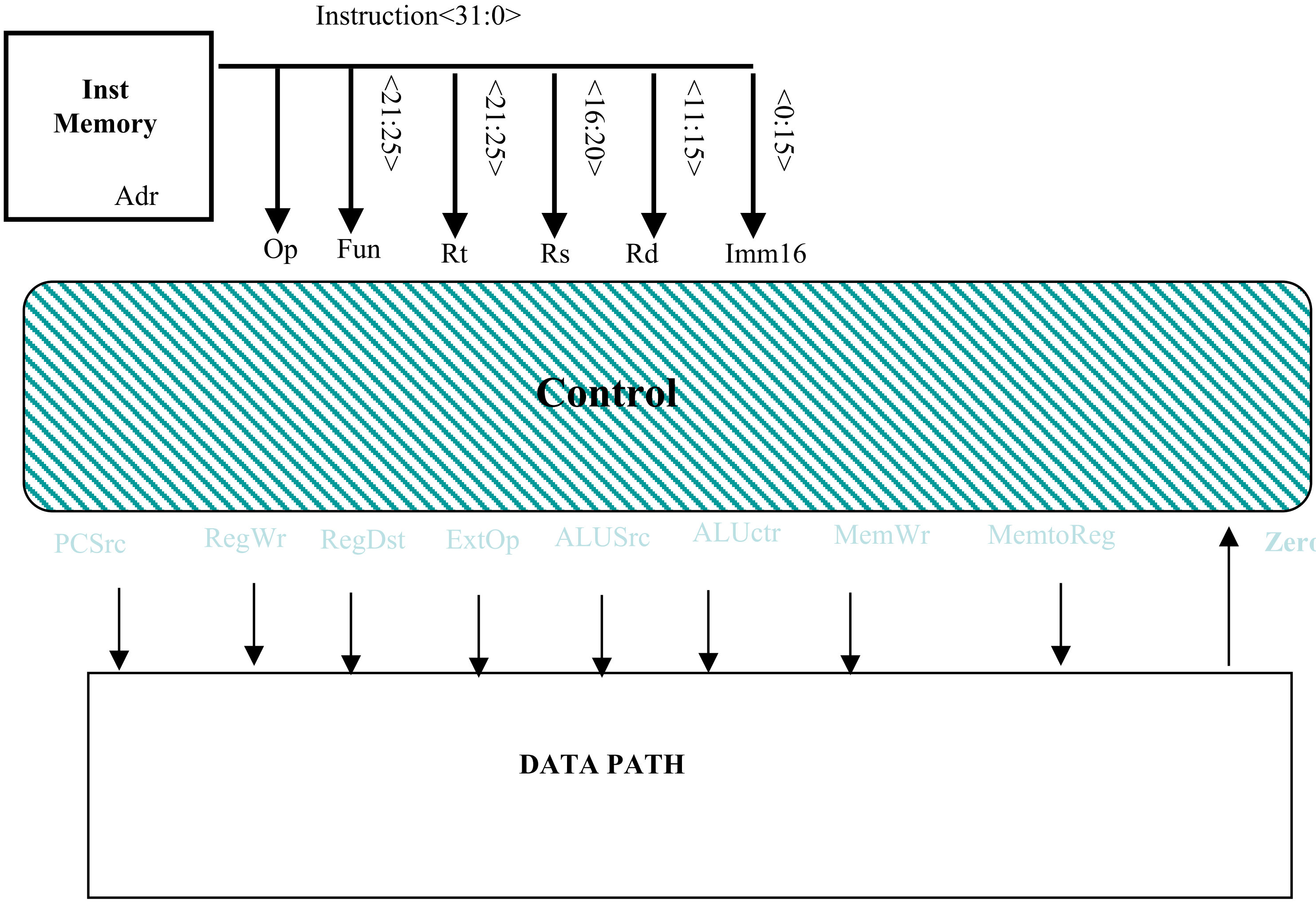


A final word on Verilog

- Verilog does not turn hardware design into writing programs!
  - Since Verilog looks similar to programming languages, some think that they can design hardware by writing programs. NOT SO.
  - Verilog is a hardware description language. The best way to use it is to first figure out the circuit you want, then figure out how to describe it in Verilog.
  - The behavioral construct hides a lot of the circuit details but you as the designer must still manage the structure, data-communication, parallelism, and timing of your design. Not doing so leads to very inefficient designs.

Peer Instruction

Step 4: Given Datapath: RTL -> Control



A Summary of the Control Signals

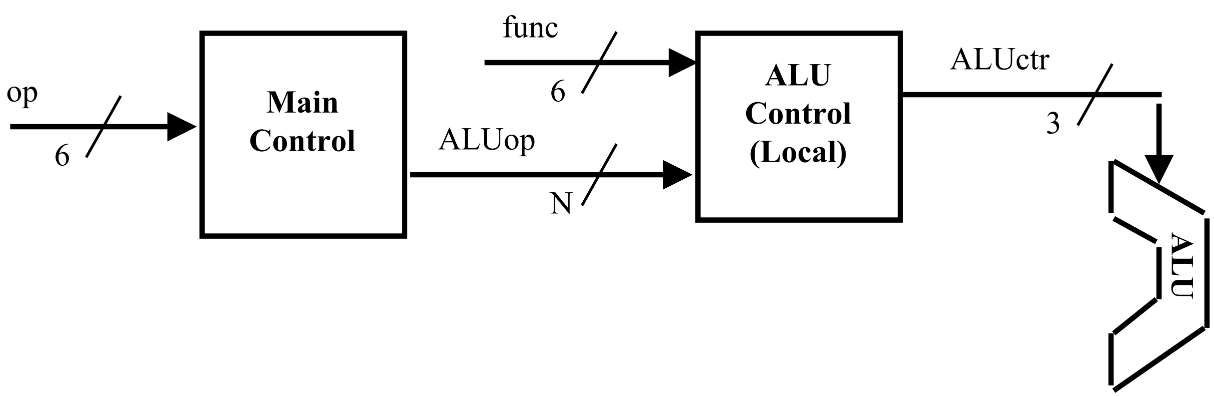
See Appendix A

	add	sub	ori	lw	sw	beq	jump
func	10 0000	10 0010	**We Don't Care :-)**				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
PCSrc	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<2:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

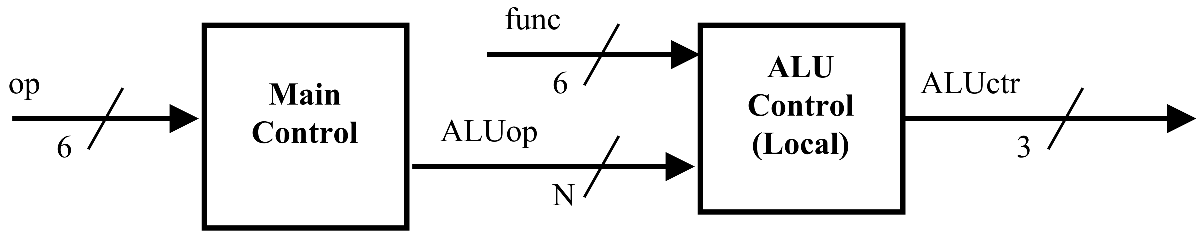
	31	26	21	16	11	6	0	
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**		**add, sub**
I-type	**op**	**rs**	**rt**	**immediate**				**ori, lw, sw, beq**
J-type	**op**	**target address**						**jump**

The Concept of Local Decoding

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



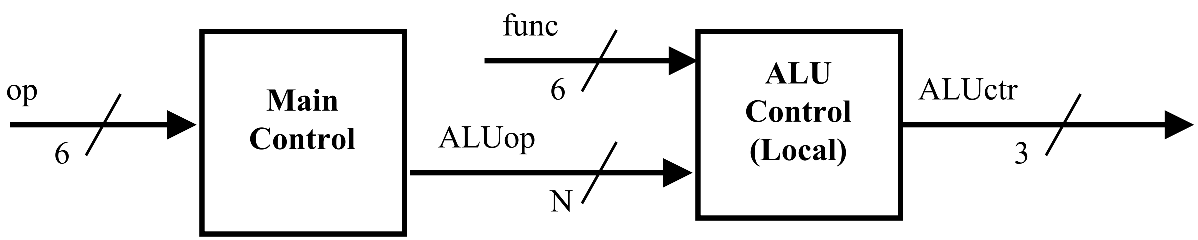
The Encoding of ALUop



- In this exercise, ALUop has to be 2 bits wide to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, and (4) Subtract
- To implement the more of MIPS ISA, ALUop has to be 3 bits to represent:
  - (1) "R-type" instructions
  - "I-type" instructions that require the ALU to perform:
    - (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

The Decoding of the “func” Field

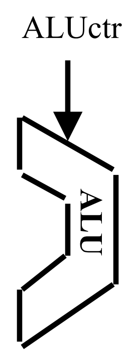


	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

	31	26	21	16	11	6	0
R-type	**op**	**rs**	**rt**	**rd**	**shamt**	**funct**	

P. 286 text:

funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than



ALUctr<2:0>	ALU Operation
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

The Truth Table for ALUctr

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	"R-type"	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

Turn Truth Table into Logic?

- Look at each output signal
- For each signal, subset truth table to include every row of when that signal is 1
- Create logic equations that cover each row, but try to minimize the logic

The Logic Equation for ALUctr<2>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

This makes func<3> a don't care

- ALUctr<2> = (! ALUop<2> & ALUop<0>) +
  (ALUop<2> & !func<2> & func<1> & !func<0>)

The Logic Equation for ALUctr<1>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

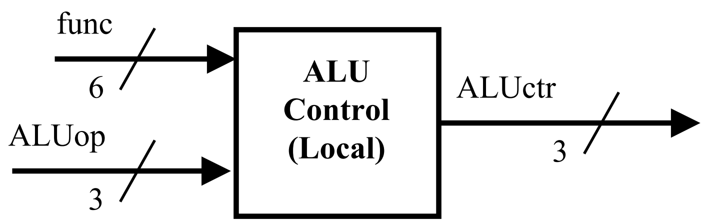
- ALUctr<1> = (!ALUop<2> & !ALUop<1>) +
  (ALUop<2> & !func<2> & !func<0>)

The Logic Equation for ALUctr<0>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

- ALUctr<0> = (!ALUop<2> & ALUop<1>) + (ALUop<2> & !func<3> & func<2> & !func<1> & func<0>) + (ALUop<2> & func<3> & !func<2> & func<1> & !func<0>)

The ALU Control Block



- ALUctr<2> = !ALUop<2> & ALUop<0> +
  ALUop<2> & !func<2> & func<1> & !func<0>
- ALUctr<1> = !ALUop<2> & !ALUop<1> +
  ALUop<2> & !func<2> & !func<0>
- ALUctr<0> = !ALUop<2> & ALUop<1>
  + ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
  + ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

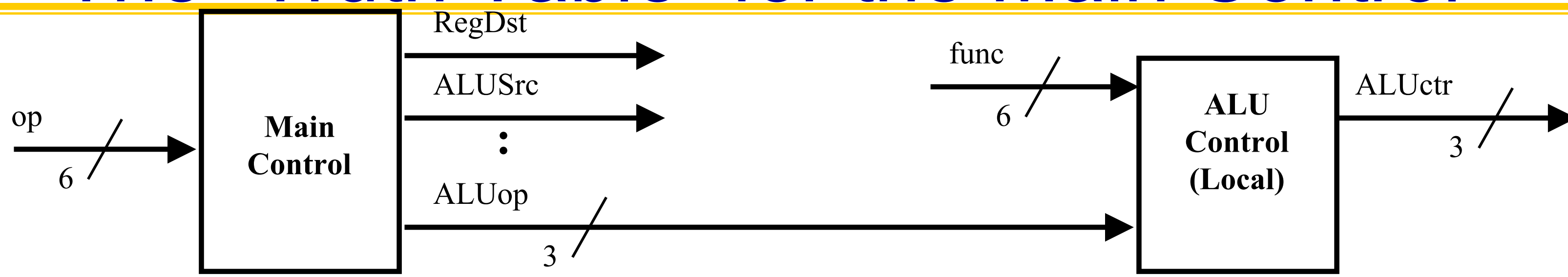
Step 5: Logic for each control signal

- PCSrc <= (OP == `BEQ) ? `Br : `plus4;
- ALUsrc <= (OP == `Rtype) ? `regB : `immed;
- ALUctr <= (OP == `Rtype`) ? **funct :**
  (OP == `ORi) ? `ORfunction :
  (OP == `BEQ) ? `SUBfunction : `ADDfunction;
- ExtOp <= ______________
- MemWr <= ______________
- MemtoReg <= ______________
- RegWr: <= ______________
- RegDst: <= ______________

Step 5: Logic for each control signal

- PCSrc <= (OP == `BEQ) ? `Br : `plus4;
- ALUsrc <= (OP == `Rtype) ? `regB : `immed;
- ALUctr <= (OP == `Rtype`) ? **funct :**
  (OP == `ORi) ? `ORfunction :
  (OP == `BEQ) ? `SUBfunction : `ADDfunction;
- ExtOp <= (OP == `ORi) : `ZEROextend : `SIGNextend;
- MemWr <= (OP == `Store) ? 1 : 0;
- MemtoReg <= (OP == `Load) ? 1 : 0;
- RegWr: <= ((OP == `Store) || (OP == `BEQ)) ? 0 : 1;
- RegDst: <= ((OP == `Load) || (OP == `ORi)) ? 0 : 1;

The "Truth Table" for the Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
PCSrc	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

The "Truth Table" for RegWrite

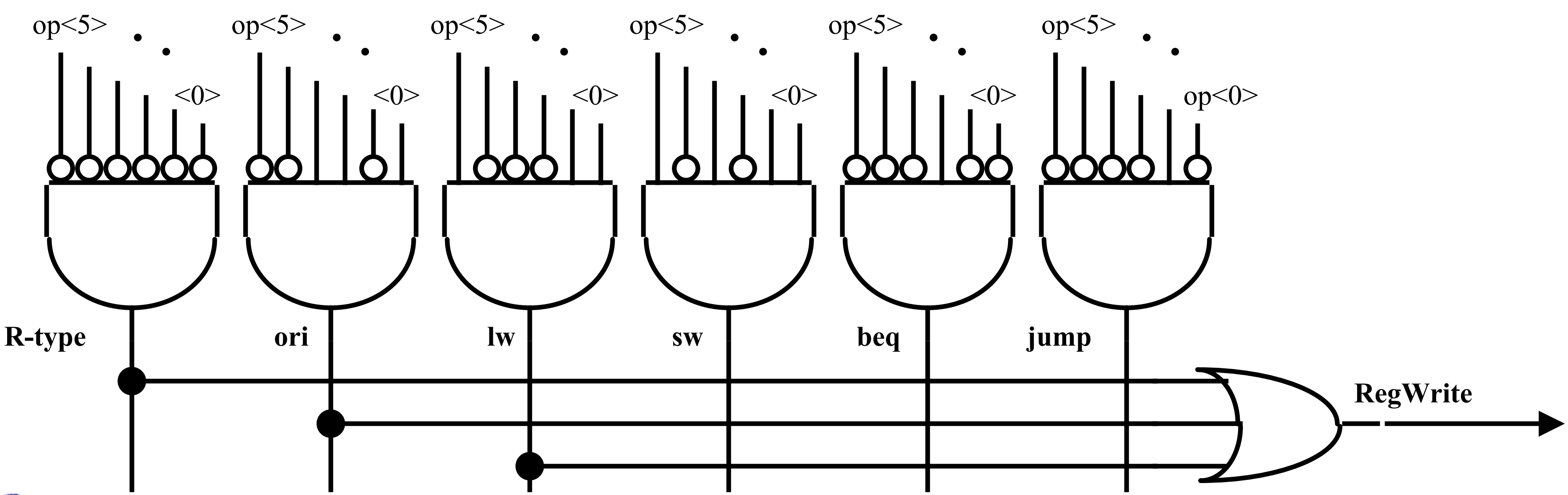
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

- RegWrite = R-type + ori + lw

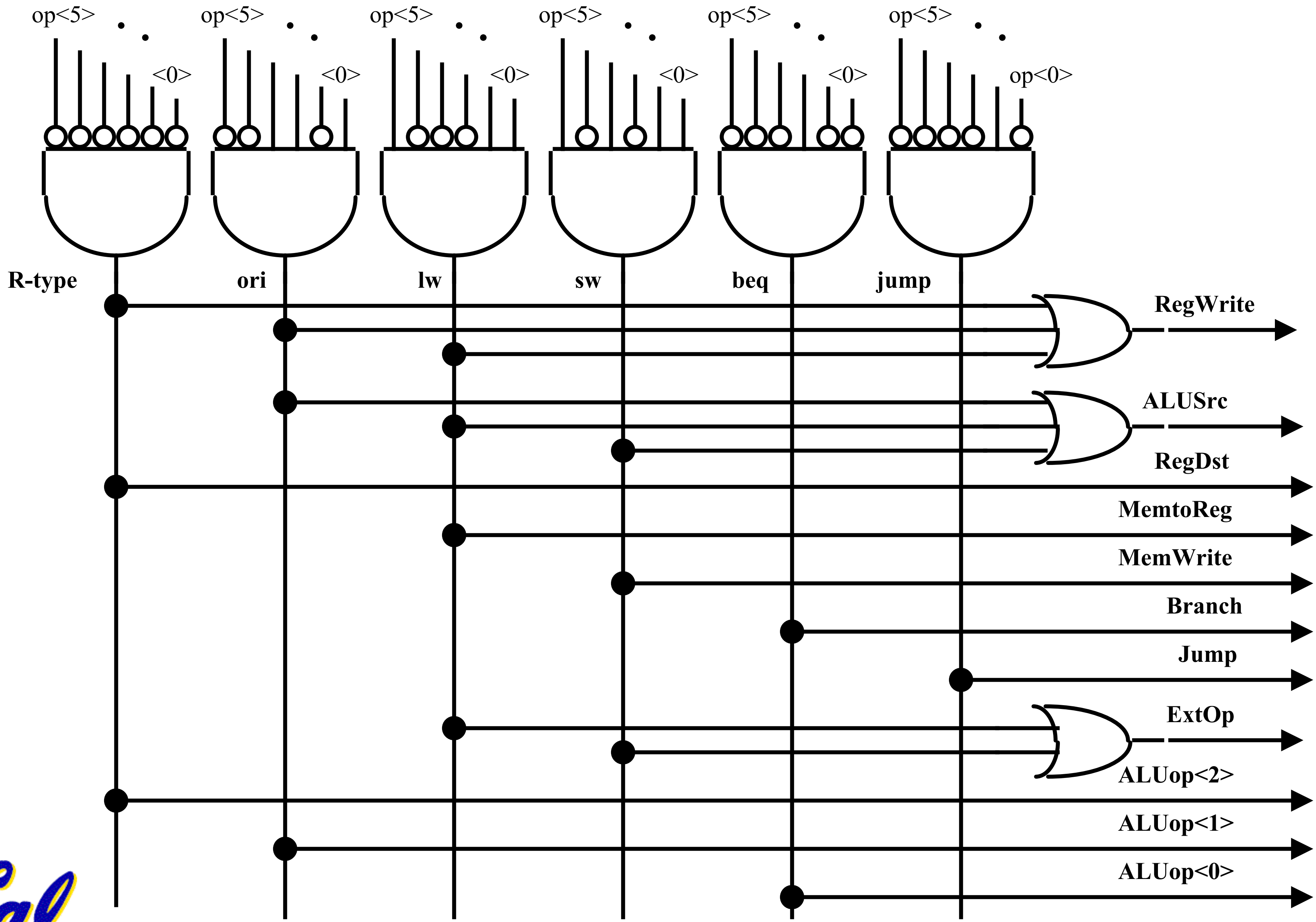
  = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

  \+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

  \+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



PLA Implementation of the Main Control



Summary

- Syncrhonous circuit: from clock edge to clock edge, just define what happens in between; Flip flop defined to handle conditions
- Combinational logic has no clock
- Always statements create latches if you don't specify all output for all conditions
- Verilog does not turn hardware design into writing programs; describe your HW design
- Control implementation: turn truth tables into logic equations