

# ECE 385 – Digital Systems Laboratory

Lecture 12 – Exam review & Experiment 7 and the Nios II  
Zuofu Cheng

Fall 2017

[Link to Course Website](#)



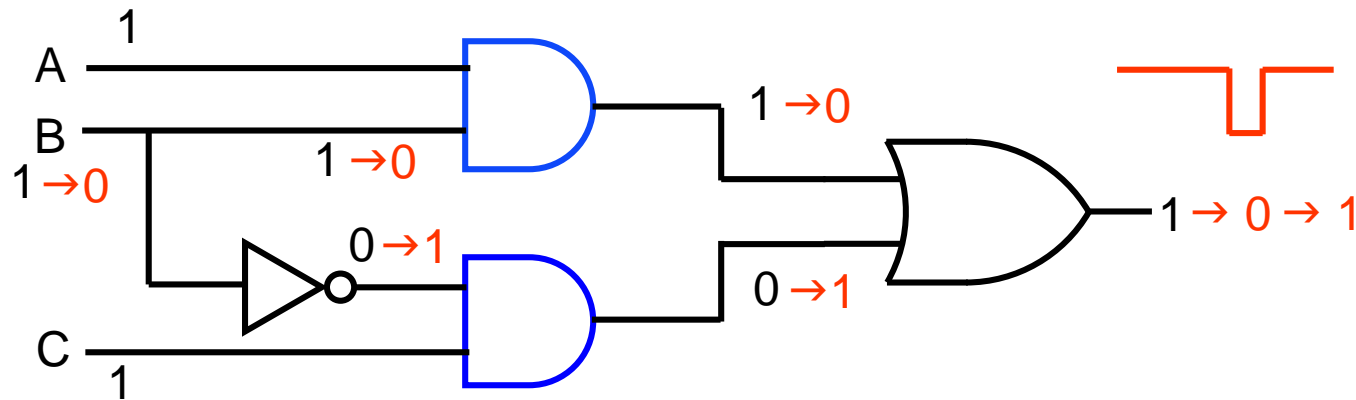
# Midterm 1 Logistics

- Midterm Exam on 11/11 in class (Wednesday)
- Rooms 1002, 2015, 2017 ECEB (by Last Name)
  - **2015** ECEB: **A – C**
  - **1002** ECEB: **D – T**
  - **2017** ECEB: **U – Z**
  - **Bring Student ID to Exam**
- 30 multiple choice questions, closed notes, 40 minutes, no calculator
- Review abstract for labs and block diagrams, review **your own notes** from lecture
- Bulk of questions on labs 1-6.1 (some on general debugging, lectures)

# General Studying Strategies

- Make sure you can sketch out the block diagrams for labs 1-5
  - Exception for lab 6, since block diagram is complex
  - This is the most important point
  - Exam will test to make sure you understand labs as an individual student
  - Also make sure you can write a short (couple sentence) description for each module
- In addition, make sure you understand how the “core algorithm” in each lab works
  - Adders in Lab 4
  - Multiplication in lab 5
  - If lab has a state machine, know generally what the states are and do

# Experiment 1: Static Hazards



		BC			
		00	01	11	10
A	0	0	1	0	0
	1	0	1	1	1

B'C (circled in blue)  
AC (circled in red)  
BA (circled in blue)

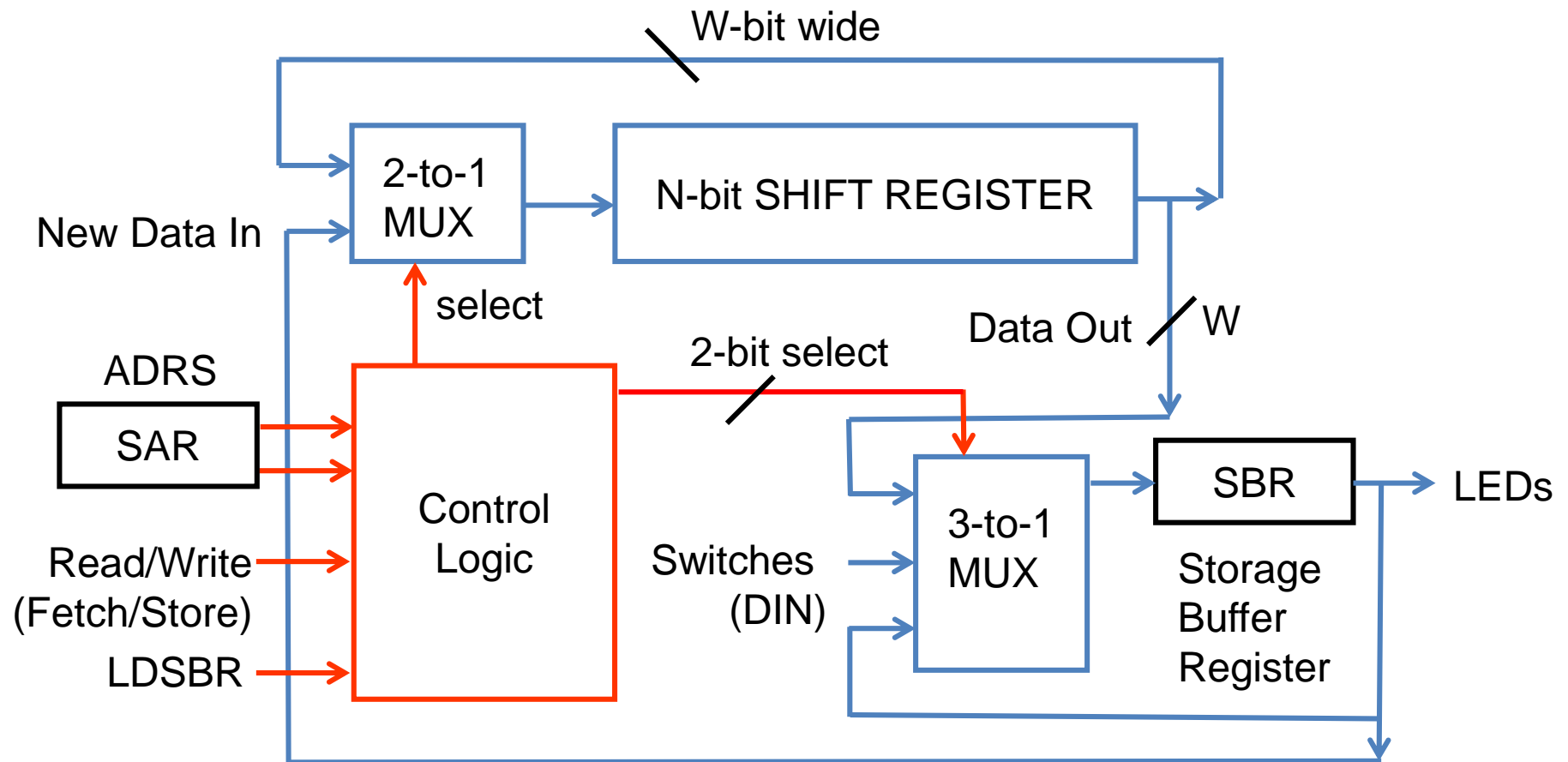
To avoid Static-1 hazard in an AND-OR (SOP) circuit, cover all adjacent min-terms in the K-map.

In this example, add the term  $AC$

- What is the final Boolean function for glitch-free circuit?
  - $Z = B'C + BA + AC$

# Experiment 2: Data Storage with TTL

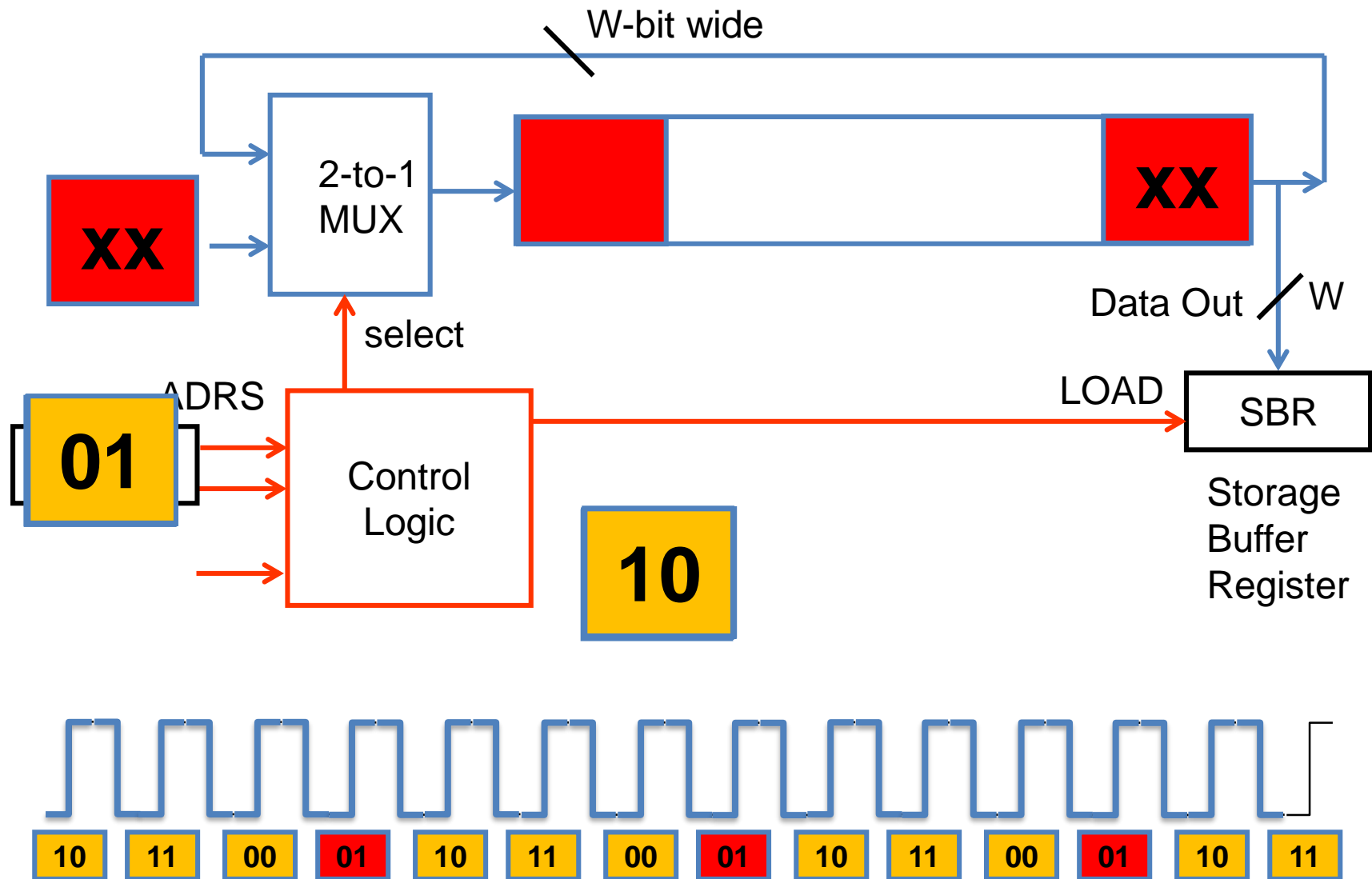
Use two 74LS194 shift-registers  
Serial Input and Output  
Circulating data



# Experiment 2: Inputs and Outputs

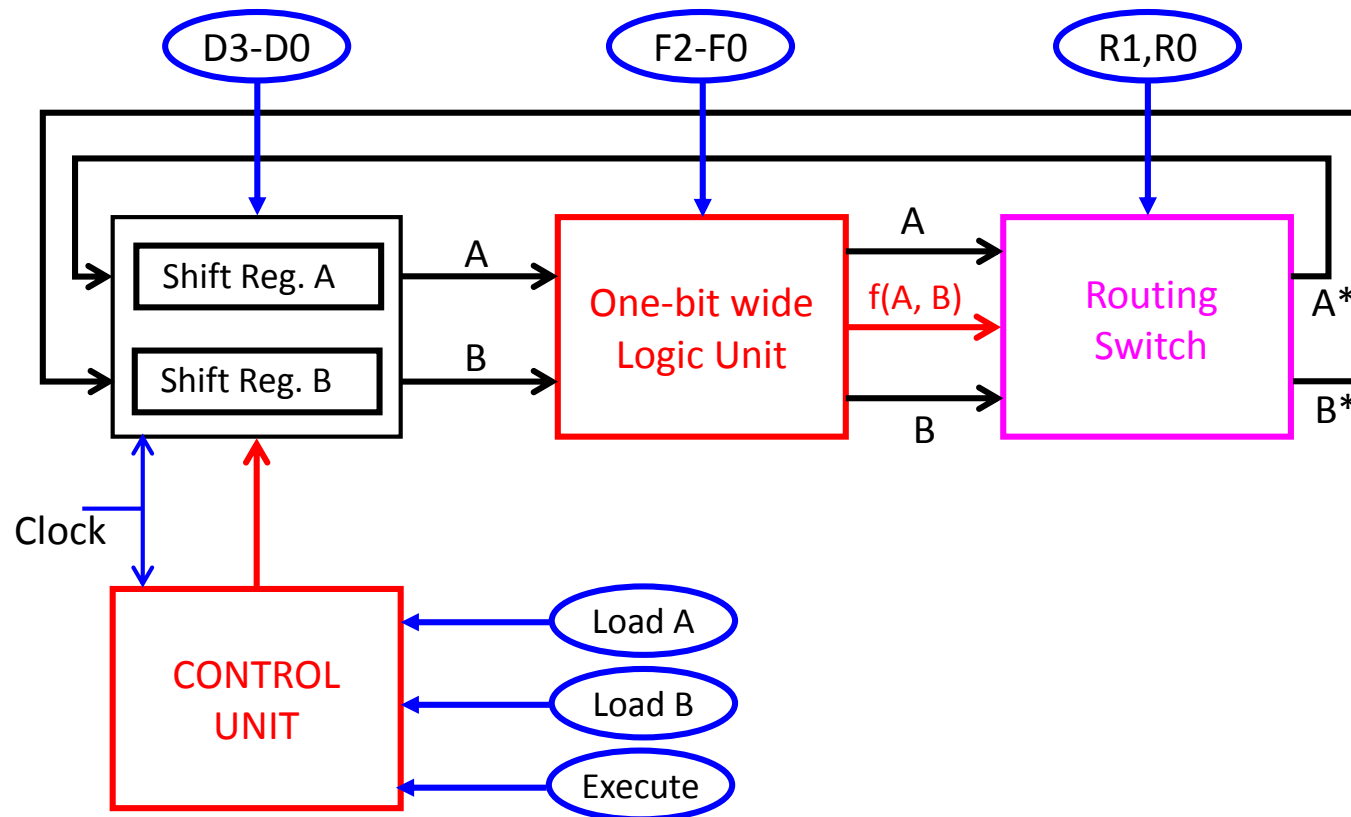
- **FETCH (Switch)**
  - When FETCH is high, the value in the data word specified by the SAR is read into the SBR.
- **STORE (Switch)**
  - When STORE is high, the value in the SBR is stored into the word specified by the SAR.
- **SBR1, SBR0 (Flip-flops)**
  - The data word in the SBR; either the most recently fetched data word or a data word loaded from switches
- **SAR1, SAR0 (Switches)**
  - The address, in the SAR, of a word in the storage
- **DIN1, DIN0 (Switches)**
  - Data word to be loaded into SBR for storing into storage
- **LDSBR (Switch)**
  - When LDSBR is high, the SBR is loaded with the data word DIN1, DIN0

## Experiment 2: Operation of Circuit



# Experiment 3: 4-bit Serial Processor

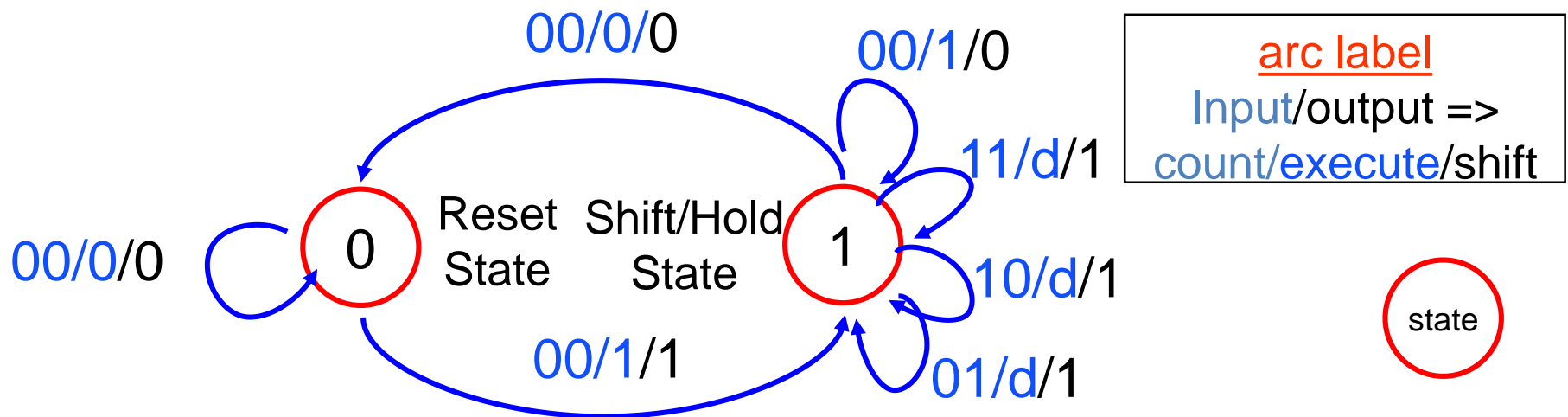
- A Bit-Serial Logic Processor
  - Two 4-bit registers A, B
  - A or B also serve as a destination register.
  - E.g., `AND A, B, A` /\* A AND B => A \*/





# Experiment 3: State Machine (Mealy)

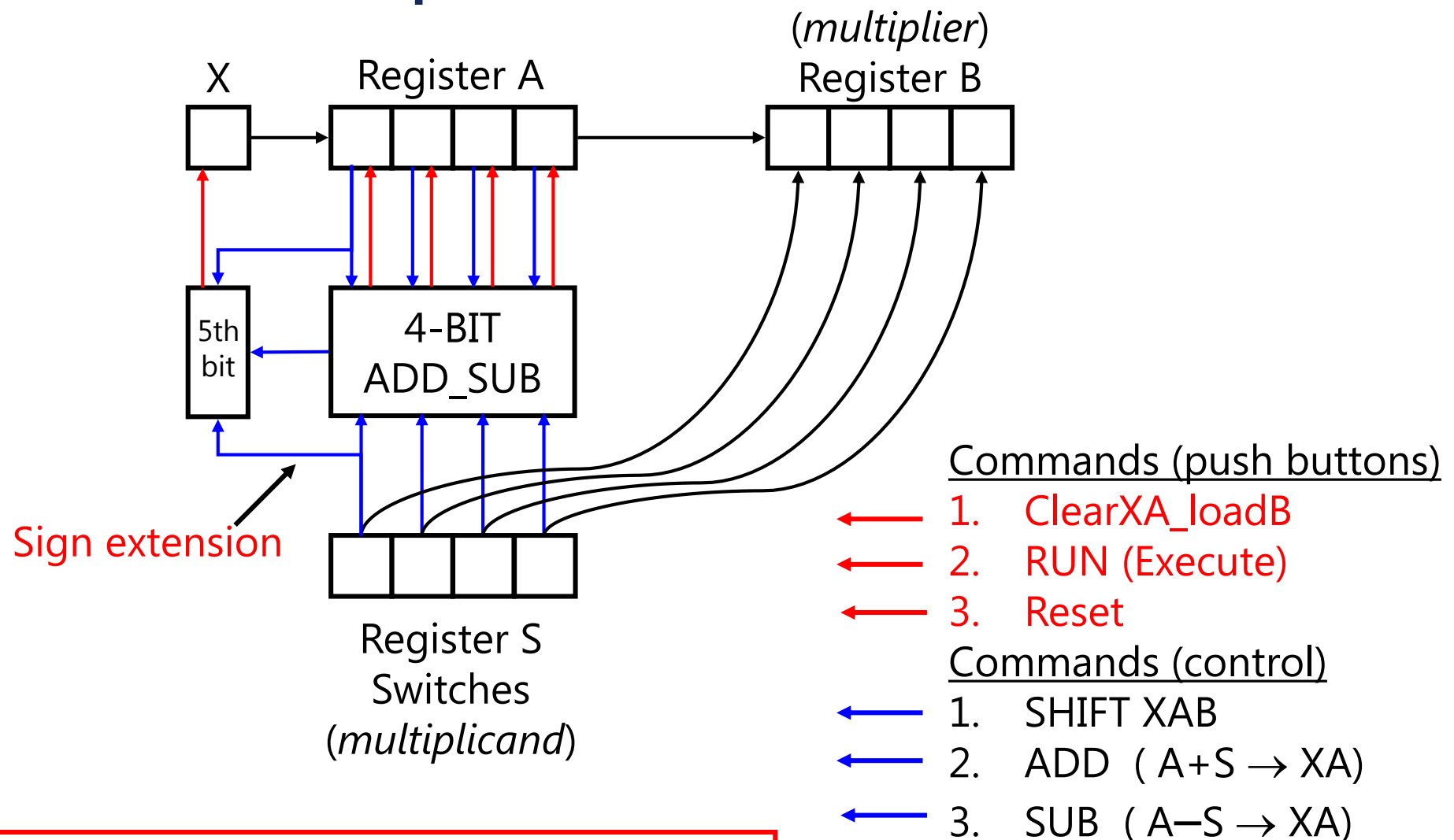
- Arcs are labeled with both inputs & outputs
- Fewer states than Moore machine (outputs depend on both state and inputs)



# Experiment 4: Adders in SystemVerilog

- Experiment 4 had 3 different types of adders
  - Carry ripple adder
  - Carry look-ahead adder
  - Carry select adder
- Know how they work and what are tradeoffs
  - Carry ripple adder – smallest and simplest
  - Carry look-ahead adder – generate P and G bits beforehand to predict carry
  - Carry select adder – pre-compute both carry cases and select correct one
- Know block diagrams for all 3 adders
- Understand how CLA and CSA speed up from CRA!

# Experiment 5: Multiplier in SV



SHIFT, ADD and SUB are derived signals from State Controller. They last only one clock cycle

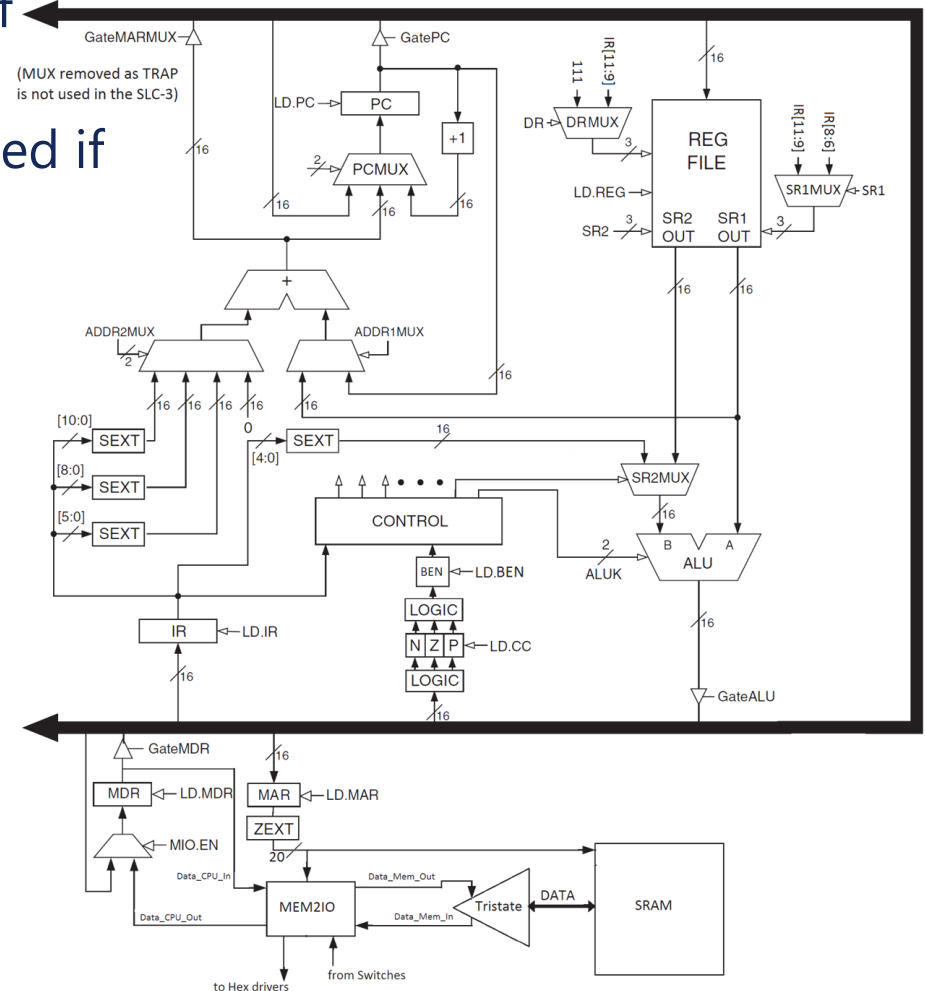
# State Machine: States (4 bit)

<u>state:</u>	<u>Action</u>	
		<start when RUN switch is ON>
S0:	$XA \leftarrow A + M \cdot S$	<if M=1 then Add S to the partial product>
S1:	SHIFT XAB	<arithmetic right shift >
S2:	$XA \leftarrow A + M \cdot S$	[fn<='0'; if M=1 then loadX <='1'; loadA <='1';]
S3:	SHIFT XAB	
S4:	$XA \leftarrow A + M \cdot S$	
S5:	SHIFT XAB	
S6:	$XA \leftarrow A - M \cdot S$	<if M=1 then fn<='1'; Subtract S from partial product>
S7:	SHIFT XAB	<final product in Register AB>
S8:	Wait Until RUN switch returns to 0	

# Experiment 6: SLC-3 CPU

- Because block diagram and state machine are complicated, they will be provided if required for any specific questions
- Instruction encoding will also be provided if necessary to do problem
- Make sure you understand additional components you wrote or used
  - Gate XYZ and MUXes
  - NZP, BEN
  - Sign extension
  - IR, register file, PC, ALU
  - Mem2IO

SLC-3 Updated Datapath for ECE 385



## Experiment 7 Goals

- Create a working Nios II/e based SoC which performs addition from switches into LEDs
- Behavior is similar to Lab 4, but using software (written in C) instead of hardware
- Program must execute from SDRAM and use PIO modules wired to LEDs
- May use provided pin-mapping file (DE2-115.QSF)
  - Simplifies pin-mapping for the (many) SDRAM signals
  - May need to reconcile names with HDL and QSYS
- Need to add an I/O constraint for SDRAM

# Motivations for System-on-Chip (SoC)

- So far, we've been designing in hardware (SystemVerilog)
- Good for tasks which require high performance (DSP, video processing, graphics)
- However – all systems need lots of low performance tasks (getting data in and out of system, formatting data, debugging, user interface)
- Want to use software for lower performance tasks

## Why use CPU and software?

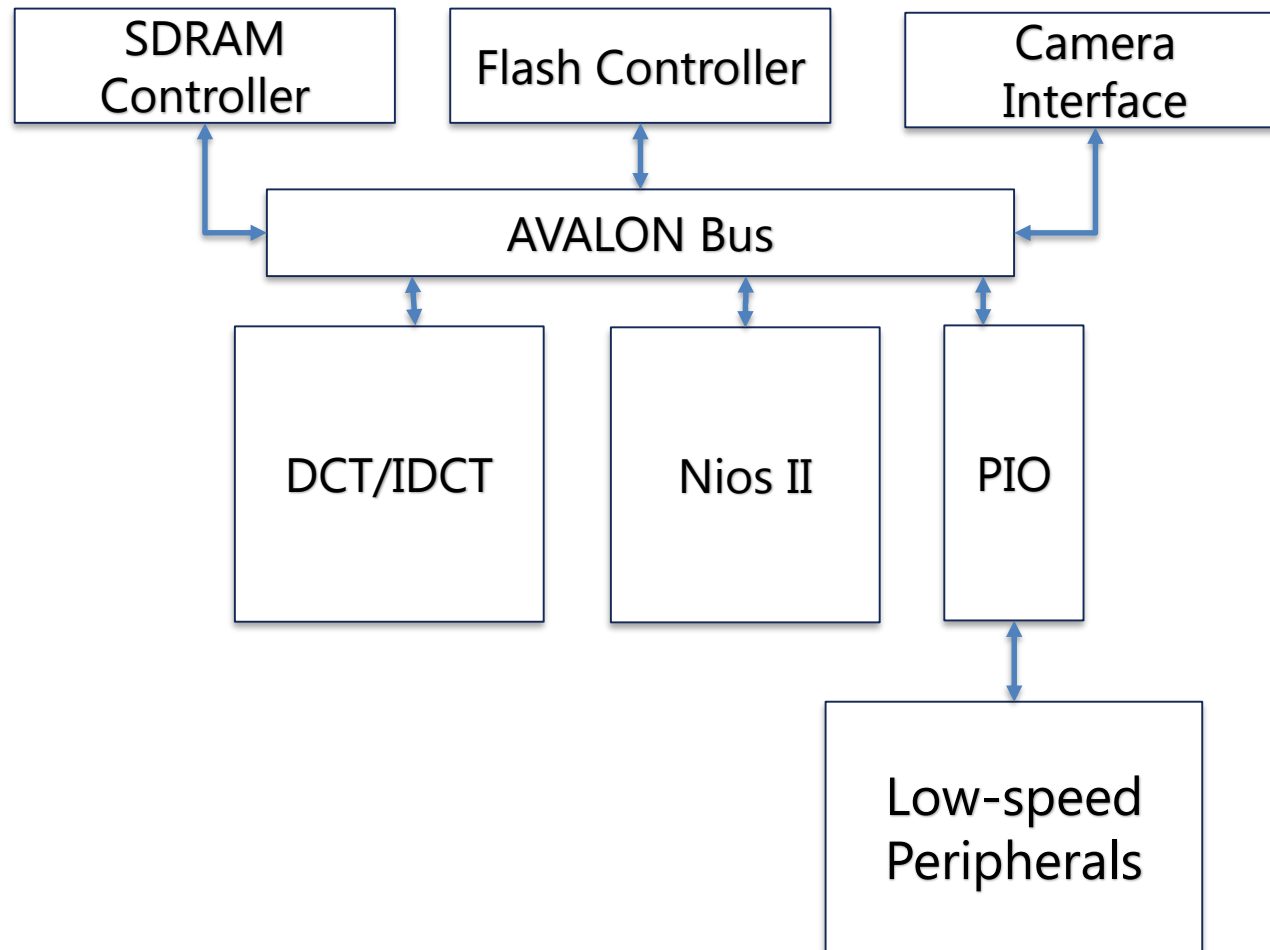
- Easier to engineer (sequential C/C++ code is much easier to write and debug than SystemVerilog)
- More efficient use of logic resources (CPU may be re-used for many tasks, dedicated hardware is application specific)
- Portable (C/C++ code can run on any platform, SV requires understanding specific FPGA platform)



# SoC Approach

- SoC = System on Chip
- Typically includes at least: CPU, memory, peripherals, accelerators
- We'll use NIOS IIe, this is a 32-bit CPU (soft-IP – which means that it is instantiated out of SV logic elements)
- Can still use rest of the FPGA for logic, in later labs, we'll create
  - Peripherals (for interfacing to other devices/standards)
  - Accelerators (for speeding up computationally intensive operations)

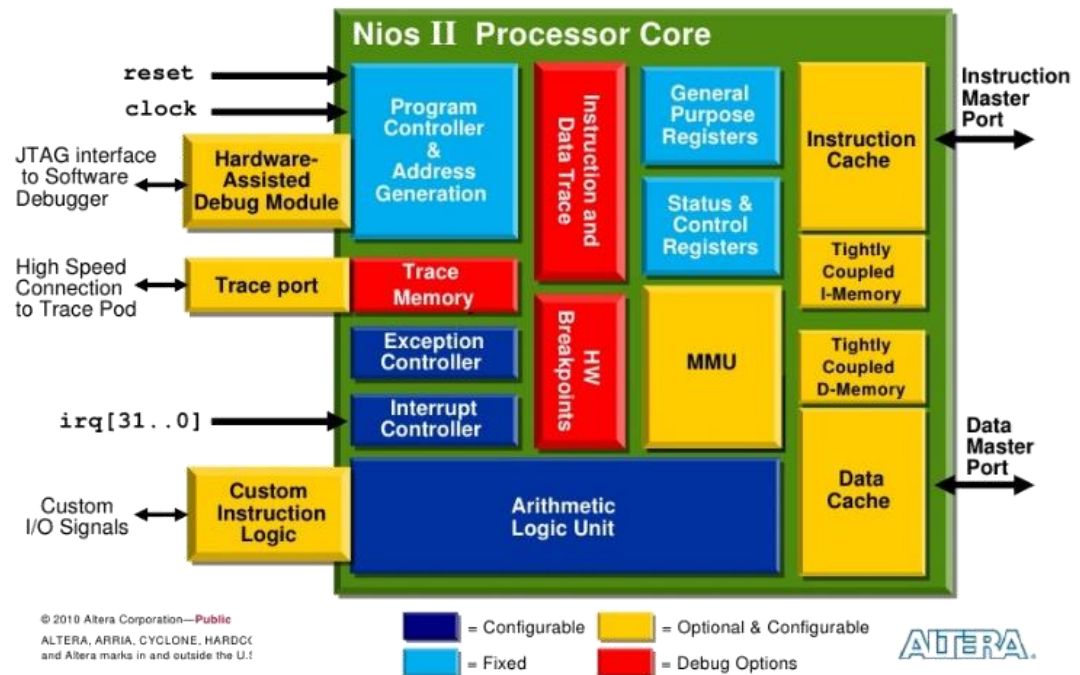
# Typical SoC System (Video Encoding)



# NIOS II ISA

- 32-bit modified Harvard RISC architecture
- User configurable tradeoffs (performance vs. LEs)
- We'll typically use "Ile" configuration (very small < 700 LEs)
- Used with Avalon bus
- Full C compiler included

## Nios II Processor Configuration



# SoC Approach

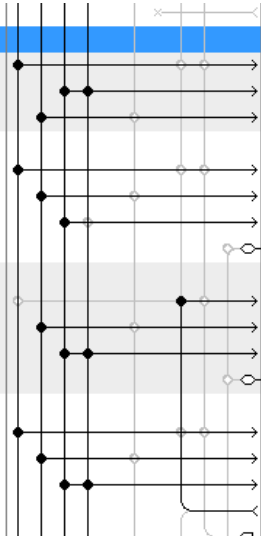
- In this class, we'll assume C/C++ compiler works (we don't need to worry too much about ISA, since we're not programming assembly)
- We won't be worried about NIOS II performance in general (since we'll only use it for low-performance tasks)
  - Performance is low compared to FPGA, only  $\sim .15$  MIPS/MHz
  - No cache memory, so largely dependent on RAM speed (slow with SDRAM, which is what we will mostly use)
- AVALON bus architecture is fairly complex, for labs 7 and 8, we'll simplify it by using an AVALON to PIO (Parallel IO) bridge
  - If we want to use more advanced features of AVALON bus (e.g. bus mastering, DMA controllers, we will need to make AVALON compliant peripherals/accelerators)
  - We'll do this approach for Lab 9

# I/O and Bus Architecture

- For Lab 7, we only need to use prepackaged IP (no new modules in System Verilog)
  - We'll instantiate a PIO module (AVALON to PIO bridge) to test our SoC
- For Lab 8, we will need to use a PIO module to interface to external USB chip and VGA peripheral
  - Software for USB host is provided, but need to write interface code to USB chipset
- For Lab 9, we'll need to design our own IP (to accelerate AES encryption)
- In all these cases, we must configure Qsys to design our SoC using the appropriate modules

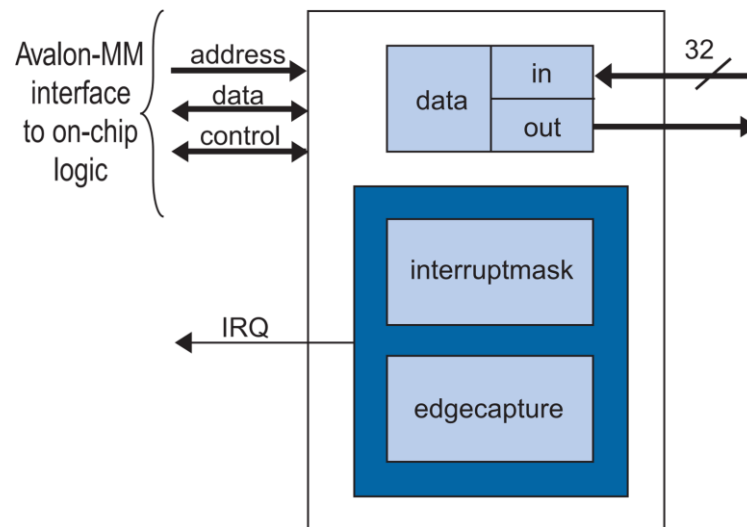
# AVALON Memory Mapped Bus

- AVALON MM bus is a 32-bit memory mapped interface
- Each device on the bus is assigned a block of addresses
  - Device could be RAM, ROM, peripheral, etc.
  - These assignments are reconfigurable in Qsys
- This is how the NIOS II interfaces to memory and I/O

<input checked="" type="checkbox"/>		custom_instruction_master	Custom Instruction Master	Double-click to export			
		<input checked="" type="checkbox"/> onchip_memory2_0	On-Chip Memory (RAM or ROM)	Double-click to export			
		clk1	Clock Input	Double-click to export	clk_0		
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0000_0000	0x0000_000f
reset1		Reset Input	Double-click to export	[clk1]			
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> led	PIO (Parallel I/O)	Double-click to export	clk_0		
		clk	Clock Input	Double-click to export	[clk]		
		reset	Reset Input	Double-click to export	[clk]		
		s1	Avalon Memory Mapped Slave	Double-click to export		0x0000_0020	0x0000_002f
external_connection		Conduit	led_wire				
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> sdram	SDRAM Controller	Double-click to export	sdram_pll_c0		
		clk	Clock Input	Double-click to export	[clk]		
		reset	Reset Input	Double-click to export	[clk]	0x1000_0000	0x17ff_ffff
		s1	Avalon Memory Mapped Slave	Double-click to export			
wire		Conduit	sdram_wire				
<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> sdram_pll	Avalon ALTPLL	Double-click to export	clk_0		
		indk_interface	Clock Input	Double-click to export	[indk_interface]		
		indk_interface_reset	Reset Input	Double-click to export	[indk_interface]	0x0000_0030	0x0000_003f
		pll_slave	Avalon Memory Mapped Slave	Double-click to export	sdram_pll_c0		
		c0	Clock Output	Double-click to export	sdram_pll_c0		
		s1	Clock Output	sdram_pll_c0			

# PIO (Parallel I/O) Module

- For Labs 7 and 8 we'll use the PIO module as a bridge from AVALON to FPGA logic
- For Lab 9, we'll actually make an Avalon module
- PIO modules may be input (to software), output (to FPGA fabric), or bidirectional
  - Note that restrictions about not having internal tristate buffers still apply
- Control registers memory mapped to addresses assigned by QSYS



# PIO Register Map

- Base address (offset 0) is assigned via Qsys
- Additional addresses are offset \* 4 addresses above base

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1) , (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set.				
5	outclear		W	Specifies which output bit to clear.				



## PIO Register Map (cont)

- Suppose PIO module for LEDs are assigned to 0x80
- PIO block is 32-bits wide, LED<sub>n</sub> is assigned to bit<sub>n</sub> on the port (we have 32 LEDs)
- How do I turn on LED 7?
- First – create pointer to PIO module (check register map)
- Data register is at offset 0, so that address is?  $0x80 + 0 = 0x80$
- `volatile unsigned int * myLEDs = (volatile unsigned int *) 0x80;`
- What is the cast for (unsigned int\*)? What is the volatile keyword for?

## PIO Register Map (cont)

- We could just do something like:
- `*myLEDs = 0x40;`
- This sets the value of the “data” register of the PIO module to binary: 0000 0000 0100 0000
- We don’t usually want to do this (why?)
- Want to preserve data already in register, (e.g. we might be using the LEDs for independent things, don’t want a single LED change to wipe away other LED states)
- What else can we do instead?

## Next time

- External clock constraints
- More NIOS II
- SDRAM and PLLs