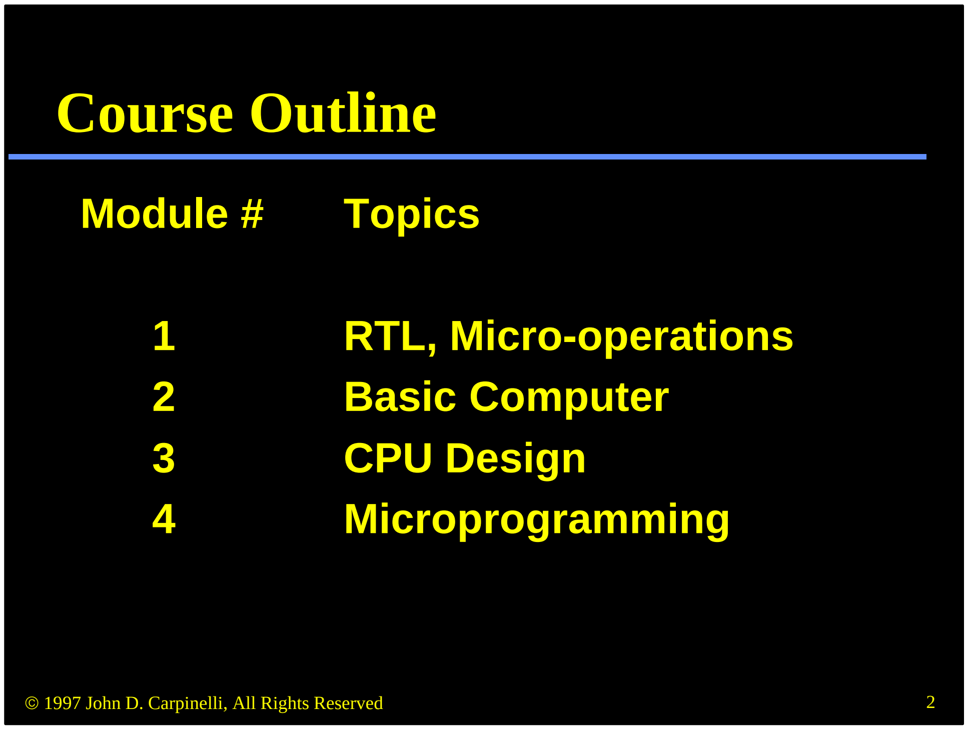
CoE 353:

Advanced Computer Architecture



Course Outline

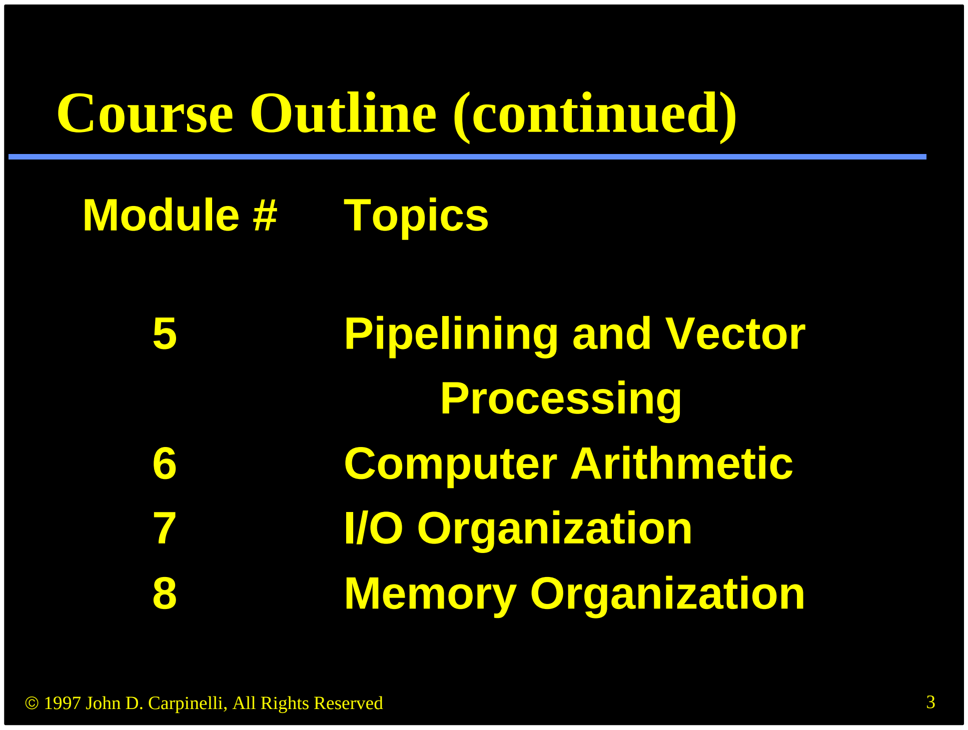
Module #	Topics
1	RTL, Micro-operations
2	Basic Computer
3	CPU Design
4	Microprogramming

RTL is Register Transfer Language. This is a hardware description language for digital system design. It allows you to specify the data transfers and data manipulation that can take place in a digital system, such as a computer, and the conditions under which these transfers take place. Each such conditional data transfer pair is called a micro-operation. By specifying the micro-operations properly, you can specify the entire digital system.

The Basic Computer was created by M. Morris Mano to illustrate the concepts of computer design. In this module we design a computer from the ground up. We start with an instruction set to be realized by the computer and the internal registers. Given that, we determine the micro-operations which must be performed by the computer to realize this instruction set. This includes both the micro-operations to fetch and to execute an instruction. We also determine the proper sequencing of these micro-operations to make the computer perform correctly.

In module 3 we explore CPU design in more detail. We will get more into the design of specific sections of the CPU, including the arithmetic/logic unit, or ALU. We also examine different instruction formats and addressing modes.

Microprogramming is one method of designing the control unit section of a CPU. In this method, control signals are stored in a separate memory unit rather than generated using combinatorial logic. The Basic Computer presented in module 2 used hardwired control. This module presents the other commonly used control unit design methodology.



Course Outline (continued)

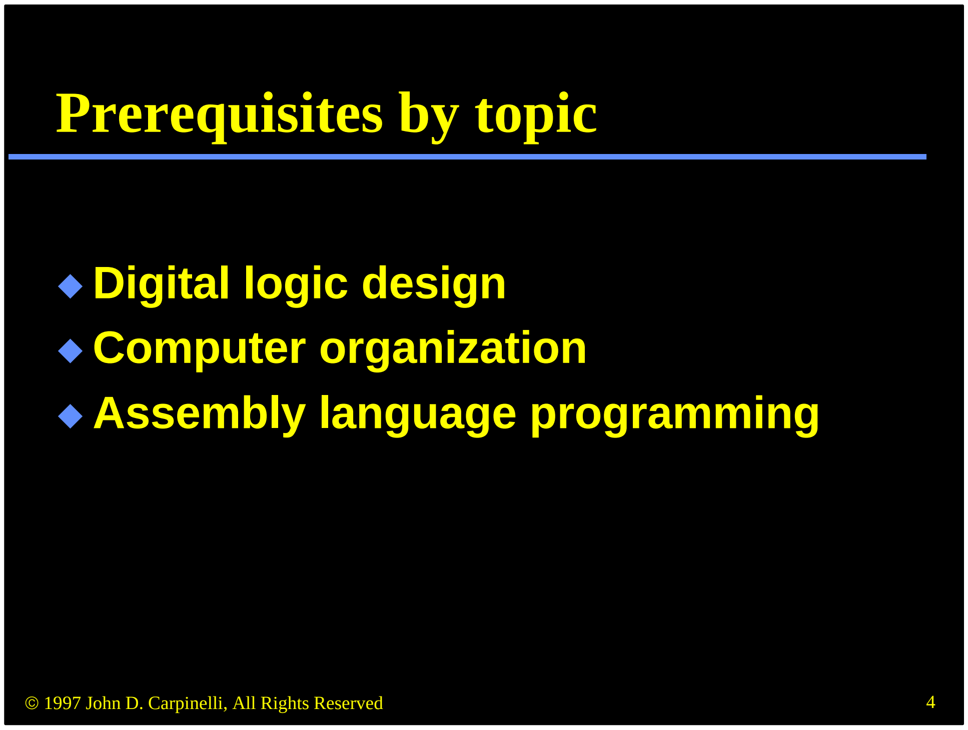
Module #	Topics
5	Pipelining and Vector Processing
6	Computer Arithmetic
7	I/O Organization
8	Memory Organization

Pipelining and vector processing are two methods used to speed up computer performance. A pipeline is very similar to an assembly line. Each stage of a pipeline performs its specific function on every piece of data that passes through it. By arranging the stages properly, the entire pipeline results in correctly processed data. Since different data is in each stage of a pipeline simultaneously, you get data out more frequently, even though each piece of data takes the same amount of time to process. Pipelines can process arithmetic data or instructions. Vector processing processes disjoint data in parallel. Although each operation takes the same amount of time, you achieve speedup by performing several operations simultaneously.

Computer arithmetic is an important part of CPU design. This module covers the design of arithmetic hardware for performing mathematical operations such as multiplication and division. This module covers different data formats, including signed-magnitude and floating point.

Input/output organization covers the design of I/O interfaces. It includes different types of serial and parallel data transfers, and interrupts.

The final module, memory organization, examines how to set up memory in a computer to maximize system performance. Topics covered in this module include cache memory, associateve memory and virtual memory and paging.



Prerequisites by topic

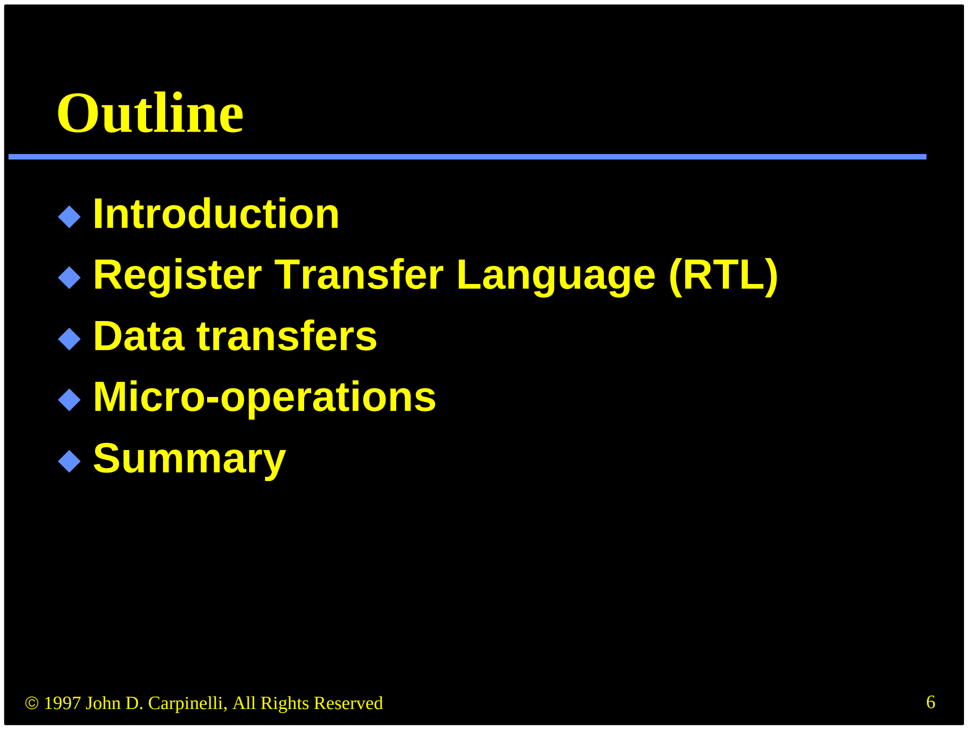
- Digital logic design
- Computer organization
- Assembly language programming

Digital logic design includes the topics typically taught in an introductory digital circuits course. These topics include combinatorial logic, sequential logic and Karnaugh maps. Students at NJIT learn this material in course EE 251, Digital Design.

Students also need to know the fundamentals of computer organization. This includes the various components of a computer and how they interact. The NJIT course which covers this material is CoE 252, Computer Architecture.

Finally, students should be familiar with at least one assembly language. The microprocessor used is not terribly important; the main concern is that each student understands how assembly languages work. NJIT course CoE 393, Assembly Language Laboratory, among others, covers this material.

Register Transfer Language and Micro-operations



Outline

- Introduction
- Register Transfer Language (RTL)
- Data transfers
- Micro-operations
- Summary

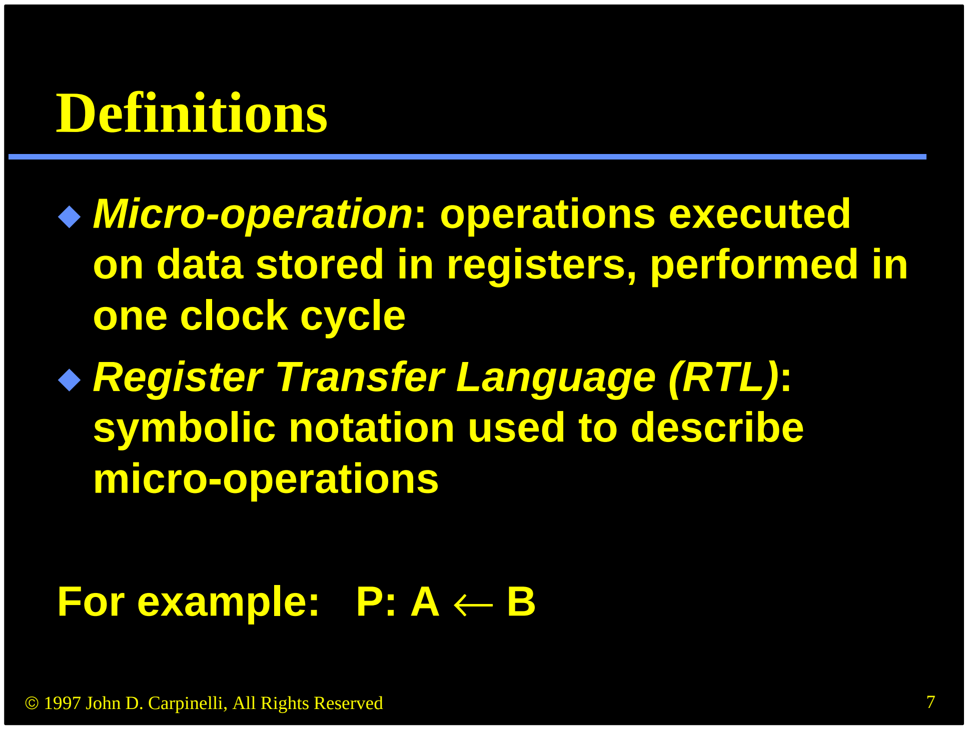
Introduction includes definitions and notation.

RTL shows various types of register transfers.

Data transfers expands on the previous topic by showing how to implement these transfers.

Micro-operations covers some of the data transfer and manipulation operations that can be performed in a CPU and the hardware to perform them.

Finally, concluding remarks are presented.

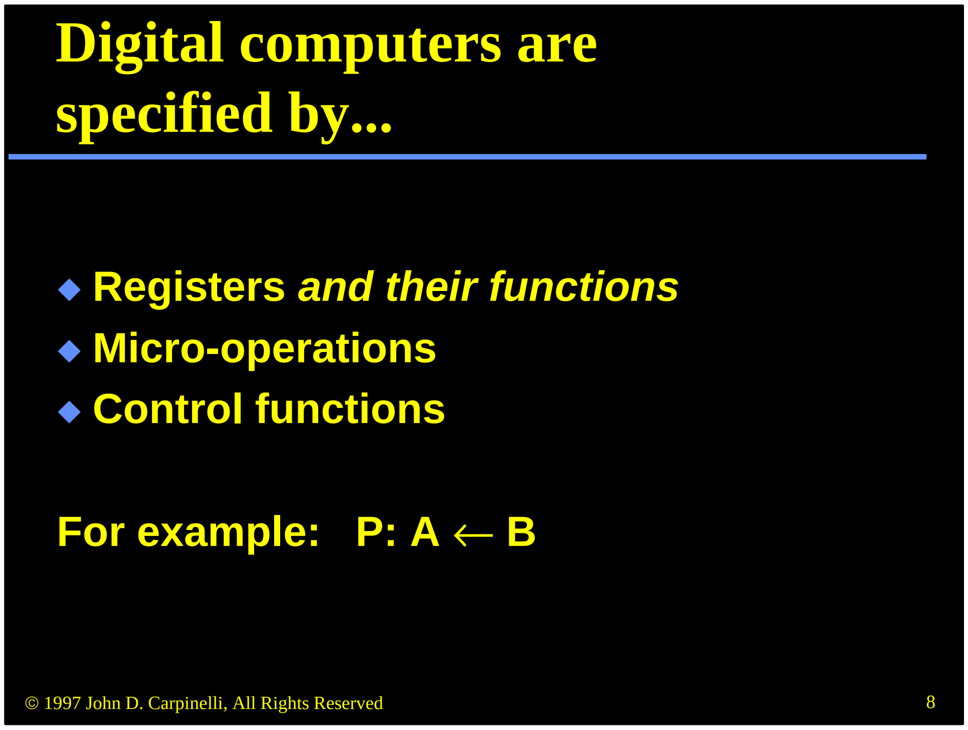


Definitions

- ***Micro-operation*: operations executed on data stored in registers, performed in one clock cycle**
- ***Register Transfer Language (RTL)*: symbolic notation used to describe micro-operations**

For example: P: A $\leftarrow$ B

Micro-operations are the basis for microprocessors. An instruction is fetched from memory, decoded and executed by performing a sequence of micro-operations. That is, a microprocessor performs the micro-operations in order to realize the instruction.



Digital computers are specified by...

- Registers *and their functions*
- Micro-operations
- Control functions

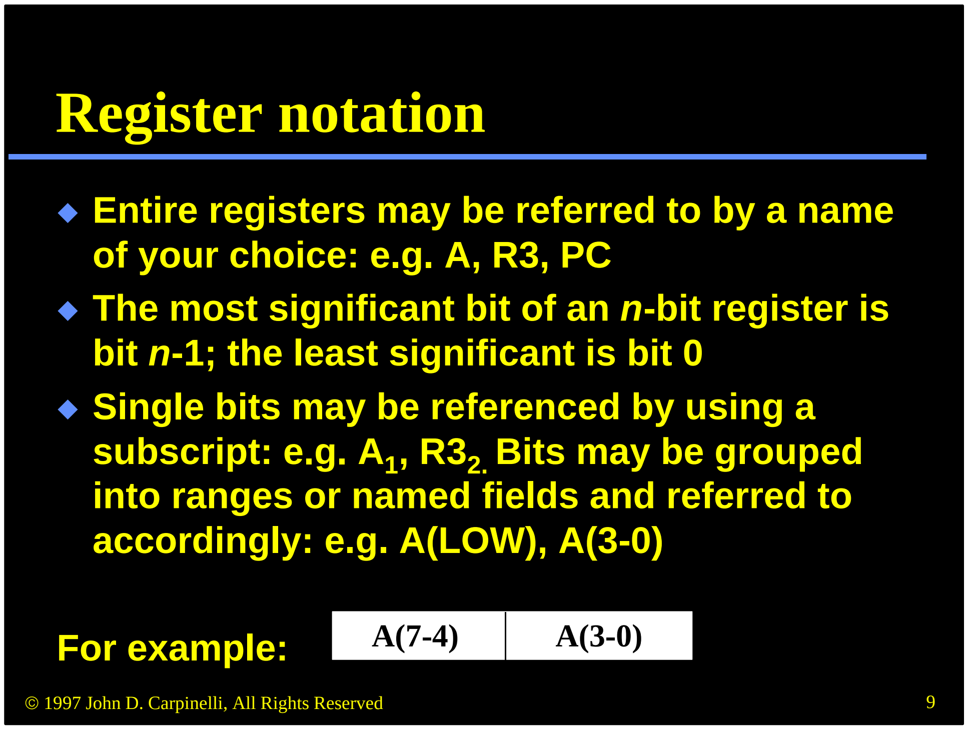
For example: P: $A \leftarrow B$

Specifying registers alone is not sufficient; it is also necessary to specify the functions which can be performed by each register. These may include such things as increment/decrement, parallel load, etc.

The micro-operations only specify which data transfers may occur; they don't specify when or how they may occur.

The control functions specify the functions which cause specific micro-operations to occur.

In the example above, P: A <-- B, P is the control function and A <-- B is the micro-operation.



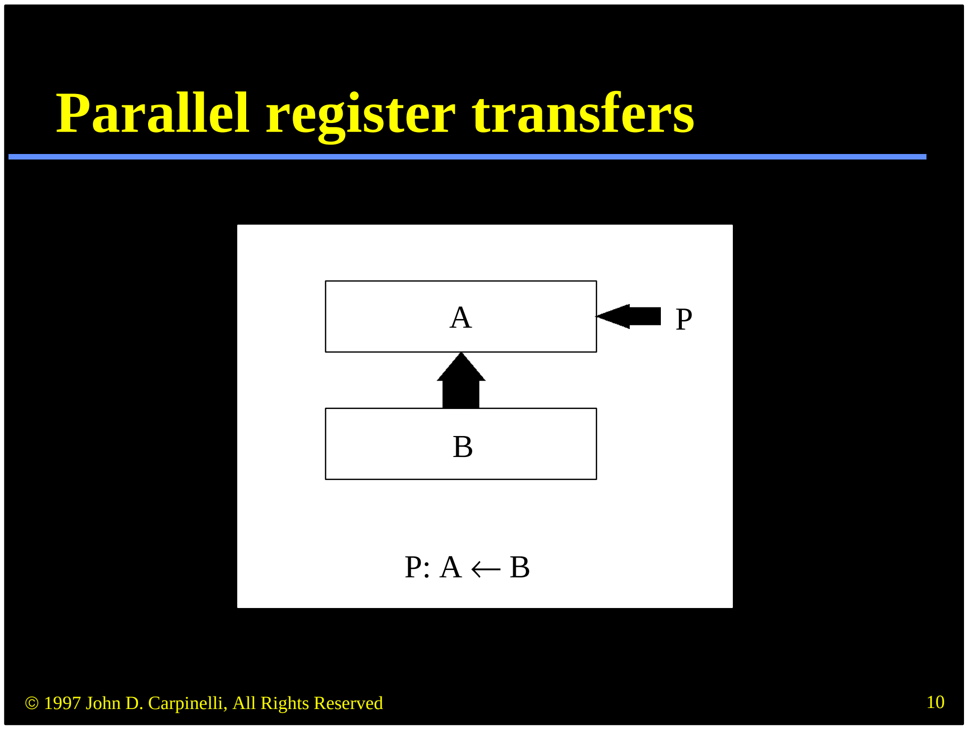
Register notation

- Entire registers may be referred to by a name of your choice: e.g. A, R3, PC
- The most significant bit of an *n*-bit register is bit *n*-1; the least significant is bit 0
- Single bits may be referenced by using a subscript: e.g. A_1, $R3_2$. Bits may be grouped into ranges or named fields and referred to accordingly: e.g. A(LOW), A(3-0)

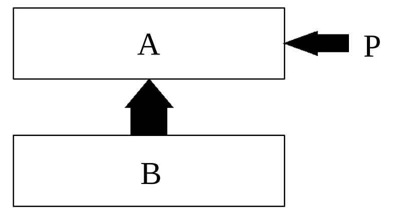
For example:

A(7-4)	**A(3-0)**

In the example, A(3-0) could have been named A(LOW).



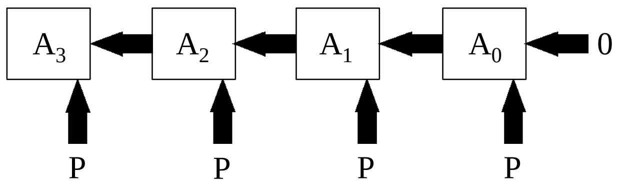
Parallel register transfers



$P: A \leftarrow B$

When condition P occurs, the contents of register B are copied into register A. Alternately, one could describe this as “if P then A gets B.” The word “transfer” is actually a misnomer, since it implies that data is moved from one location to another. In fact, the data is copied from one location to another, since it also still resides in register B.

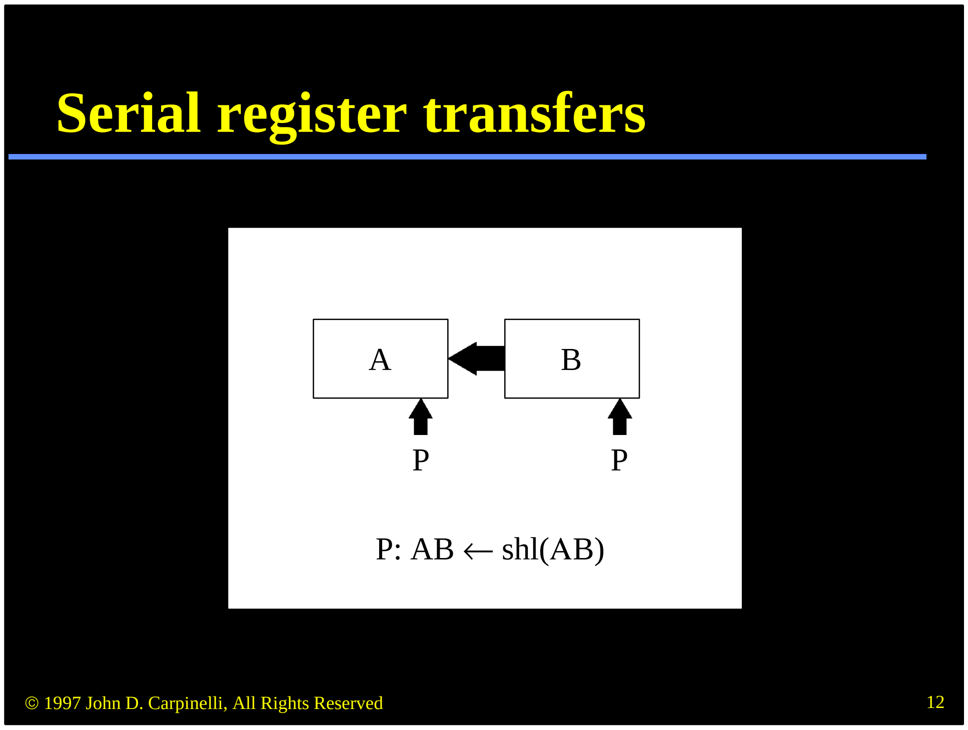
Serial register transfers



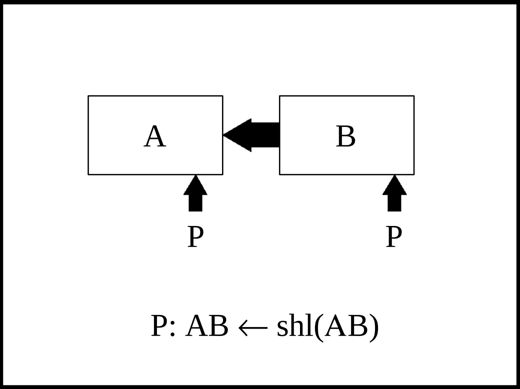
$P: A \leftarrow shl(A)$

In this example, each bit of A is shifted one position to the left. The value of 0 is shifted into the least significant bit of A. The value of the most significant bit of A is lost.

For example, if A = 1001, shl(A) = 0010.

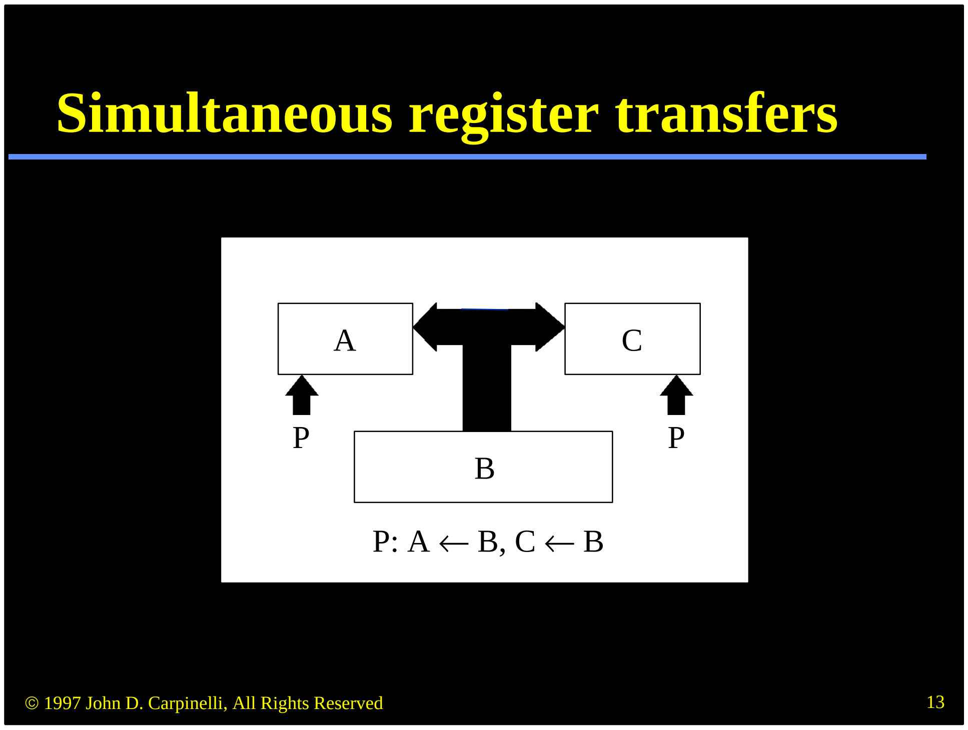


Serial register transfers

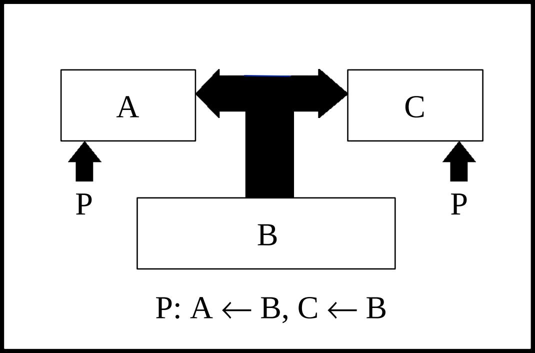


P: AB $\leftarrow$ shl(AB)

For example, if A = 1001 and B = 0110, shl(AB) = 0010 1100.

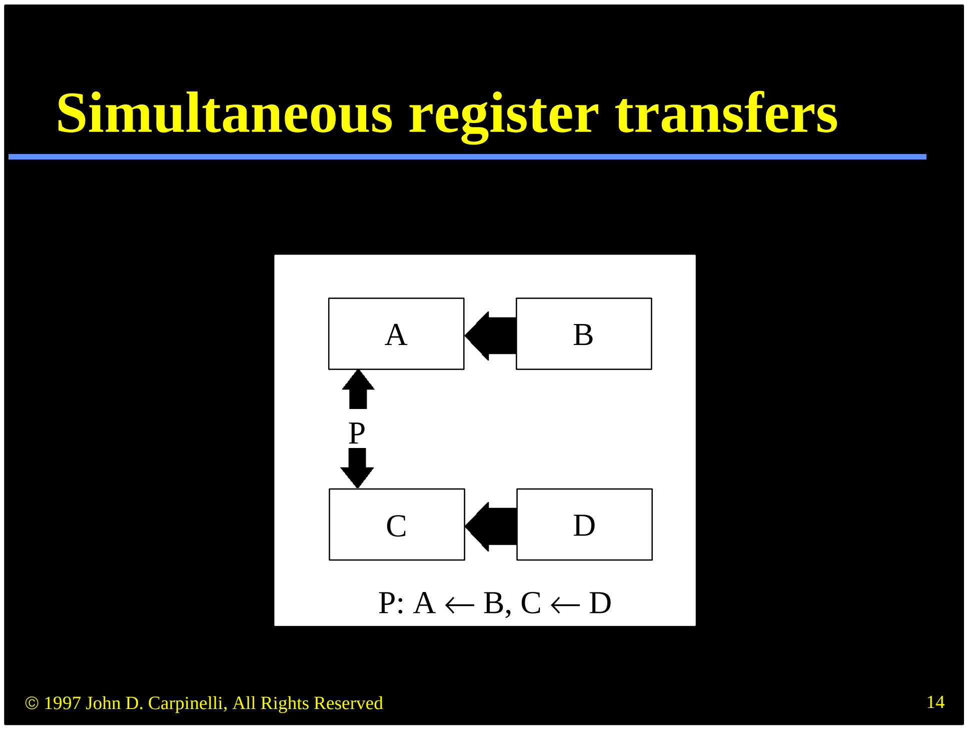


Simultaneous register transfers

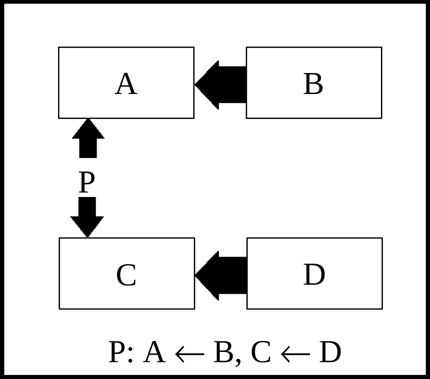


P: $A \leftarrow B$, $C \leftarrow B$

In this example, when signal P is asserted, the contents of B are copied into both registers A and C. Note that the order in which the statement is written does not matter. If the transfer from B to C was written first, the result would be the same. As long as the transfers to occur are separated by commas, they are simultaneous.

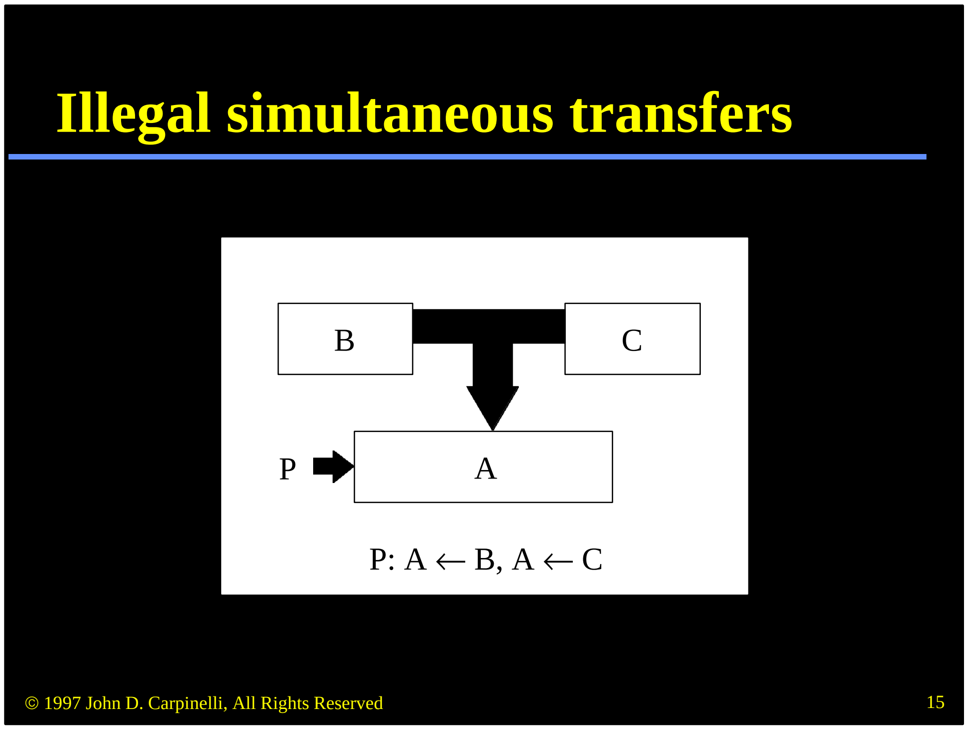


Simultaneous register transfers

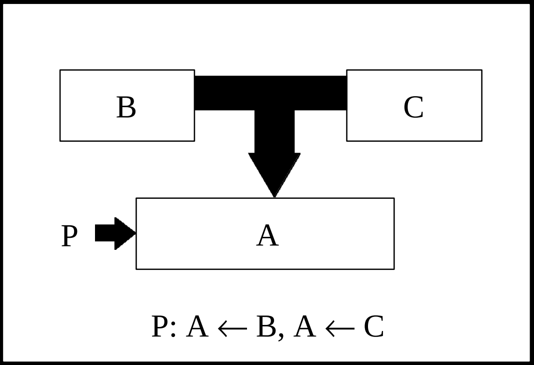


P: $A \leftarrow B, C \leftarrow D$

In this example, two registers are loaded simultaneously from different sources.

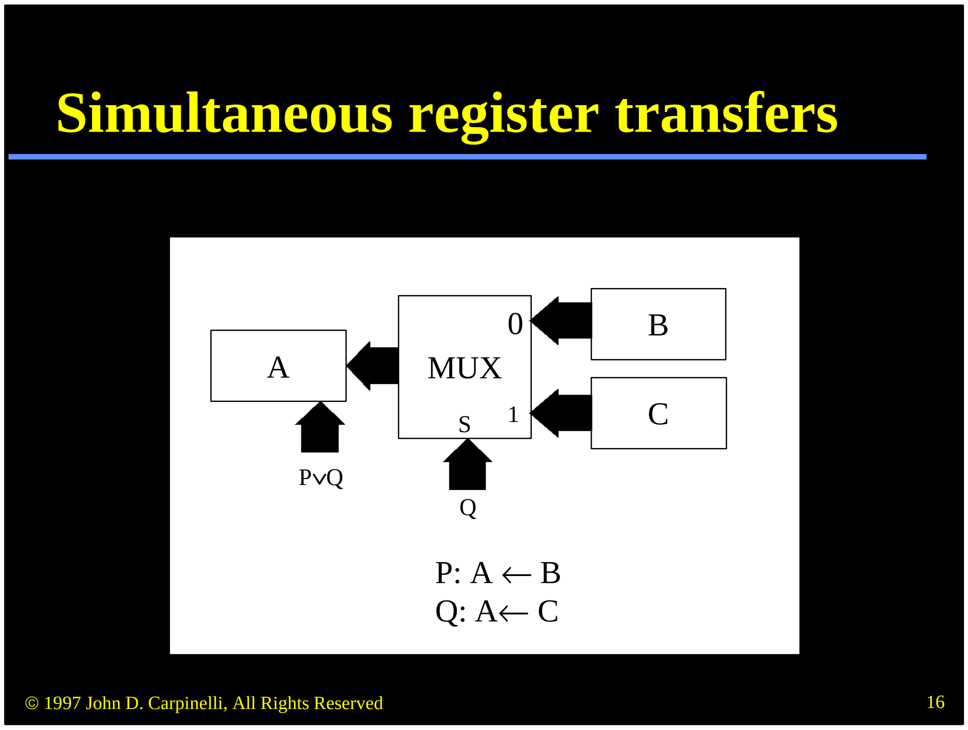


Illegal simultaneous transfers

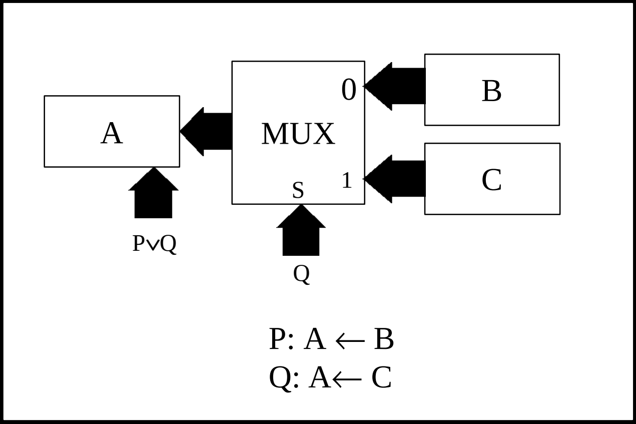


$P: A \leftarrow B, A \leftarrow C$

This is an example of an illegal operation, since A must be loaded with two different values simultaneously.



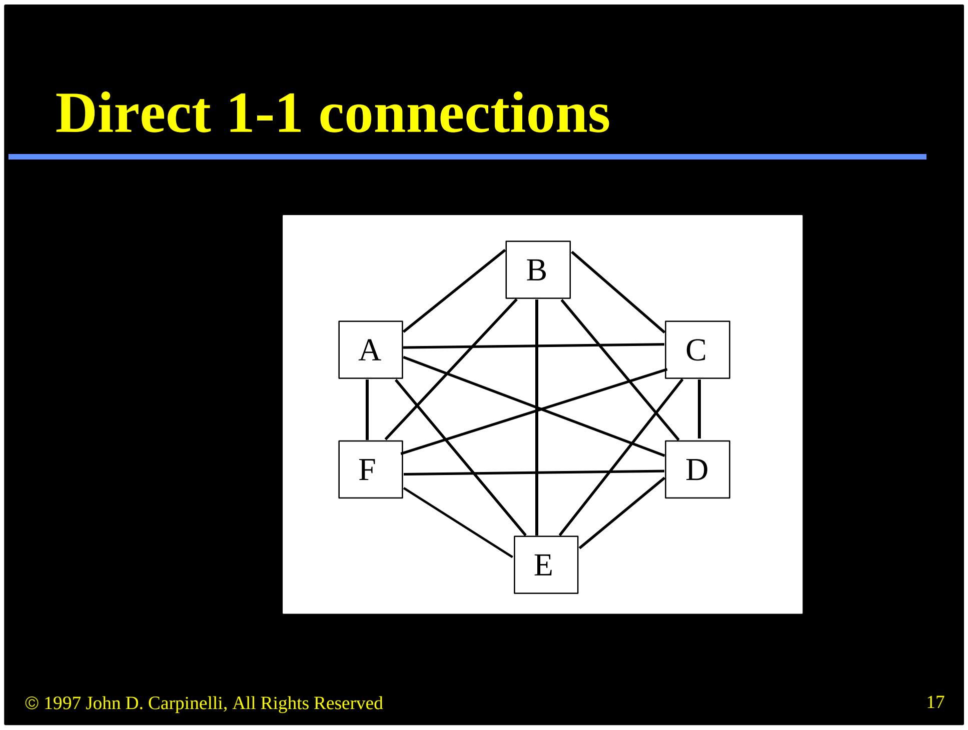
Simultaneous register transfers



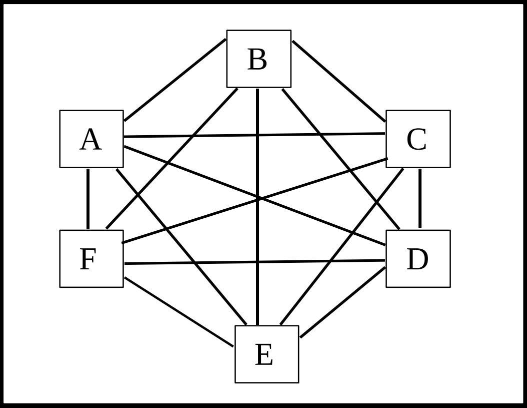
P: A ← B

Q: A← C

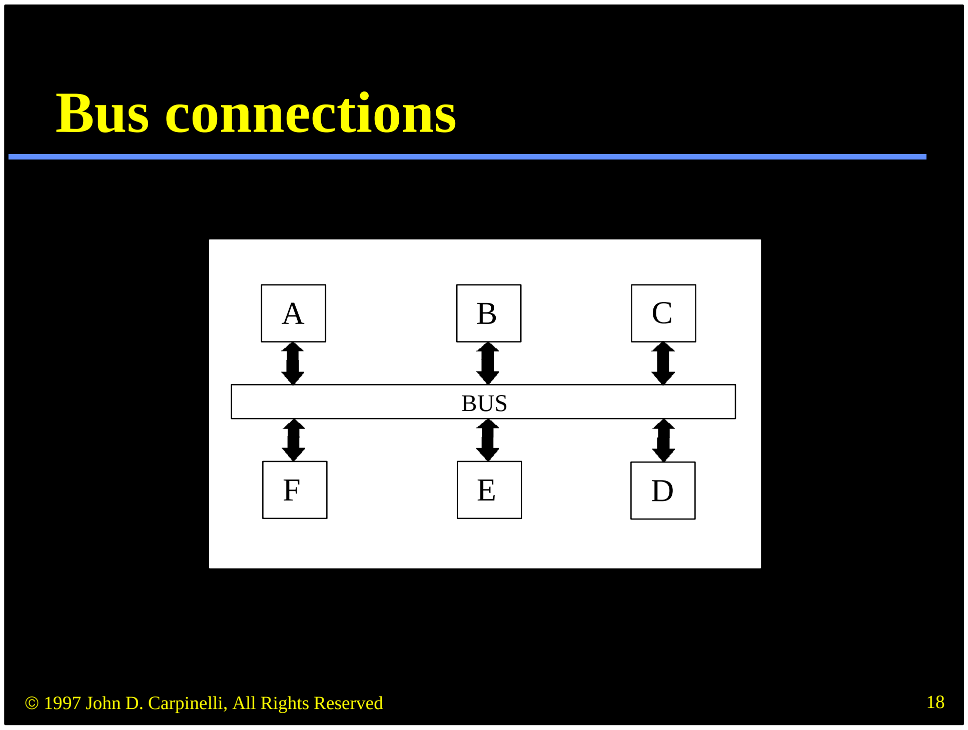
In this example, A can receive data from more than one source, depending on the control signal asserted. A multiplexer is used to determine the data to be loaded, although tri-state buffers could also have been used.



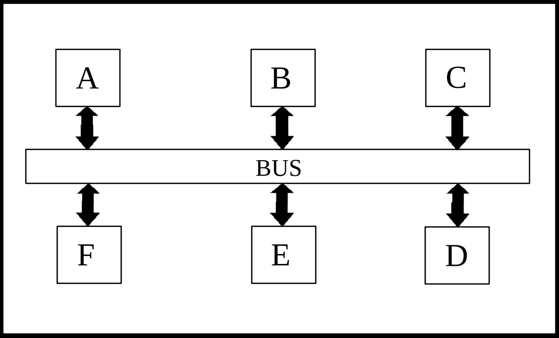
Direct 1-1 connections



To connect *n* items with direct connections, you need *n(n-1)/2* connections.



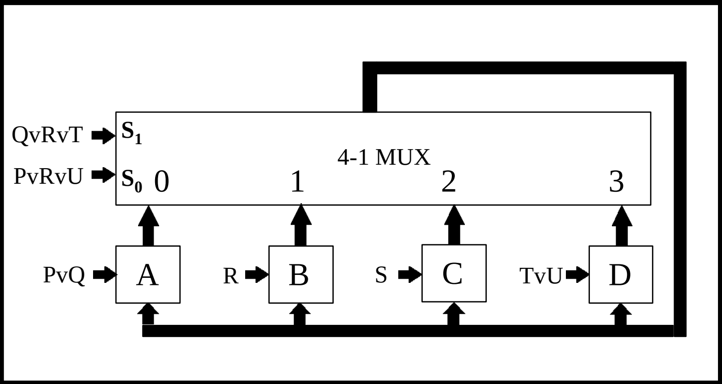
Bus connections



To connect *n* items with bus connections, you need only *n* connections.

Using a bus is similar to using a highway. The Garden State Parkway is about 172 miles long and contains about as many exits. If you built a separate road to connect each pair of exits, you would have about 15,000 different roads. There would hardly be any room left for the cities they connect! However, by constructing the Parkway and having each city connect to the Parkway, you have only one connection per exit, barely 1 % of the roadway. Also, since each road connects at its nearest point to the parkway, the connections are much shorter.

Connecting registers with multiplexers



This system implements the following micro-operations:

P: A <-- B

Q: A <-- C

R: B <-- D

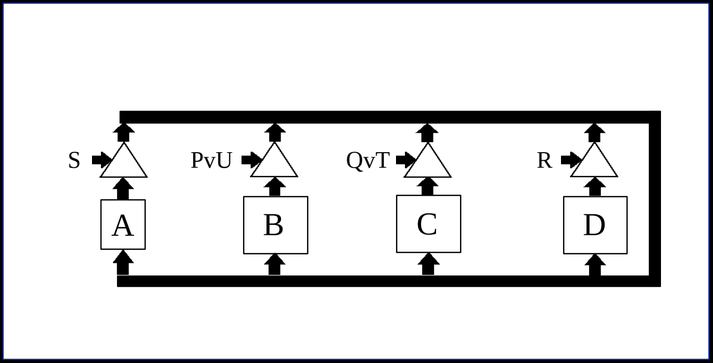
S: C <-- A

T: D <-- C

U: D <-- B

The multiplexer selects one of the four registers as the source register. Control lines for the mux are driven by external circuitry. The data is made available to all registers, but only one actually loads the data. Again, external hardware generates load signals for the four registers such that no more than one is active at any given time.

Connecting registers with tri-state buffers

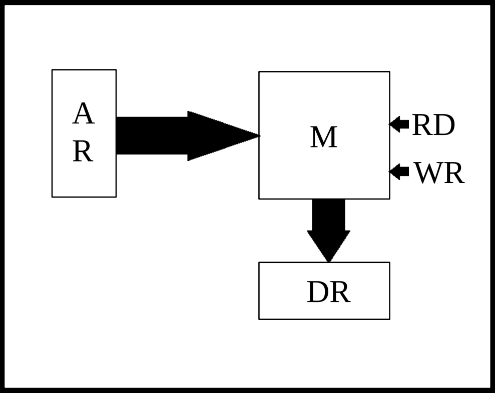


This system implements exactly the same micro-operations as in the previous example.

In this configuration, external circuitry enables at most one of the tri-state buffers to pass its data through to the inputs of the registers. Enable signals for the tri-state buffers are not shown. As before, the data is made available to all registers, but only one actually loads the data. The load signals for the four registers are exactly the same as in the previous example. This makes sense since the registers only know that they are to read in data from the bus. They don’t care how the data got onto the bus in the first place.

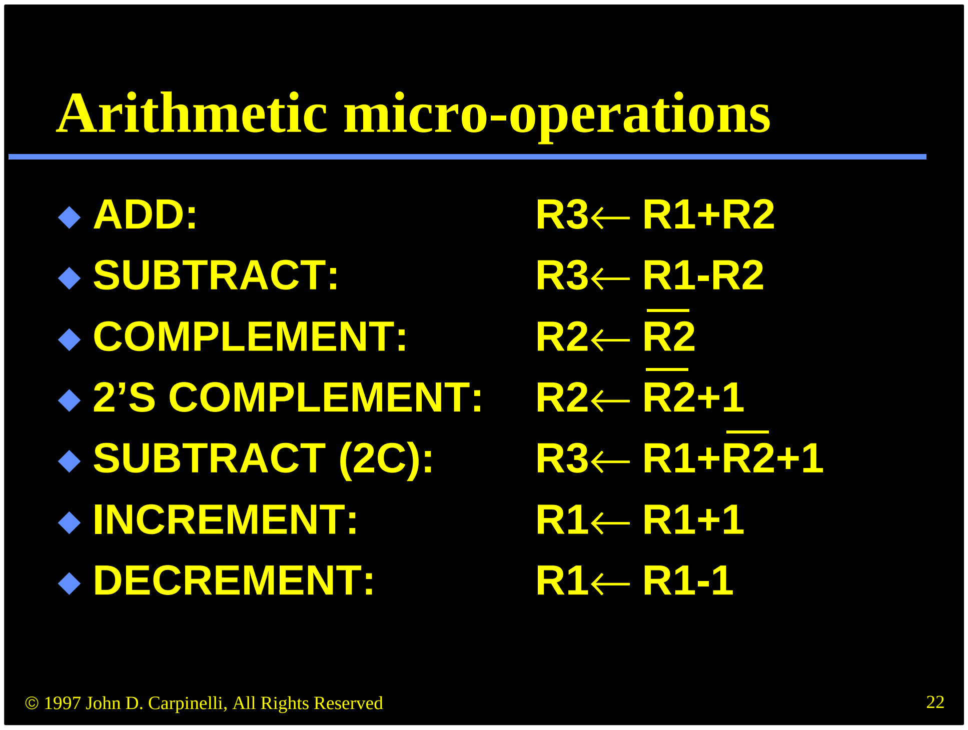
The primary advantage of using buffers rather than multiplexers is the reduced current load on the circuit.

Reading data from memory



In memory read operations, the address of the memory location to be read is supplied by the address register of the CPU. The memory module inputs this address and makes the data from that memory location available to the CPU, which reads it into either its data register (shown here) or its instruction register (not shown). The address register may or may not have tri-state buffers on its outputs. The memory module always has tri-state buffers on its outputs (not shown). The data register has a load signal that triggers it to load in the data.

In an actual CPU, the data register may receive data from sources other than the memory module, so some sort of input arbitration is necessary. This is usually similar to the tri-state buffers used in the example shown in the previous slide.



Arithmetic micro-operations

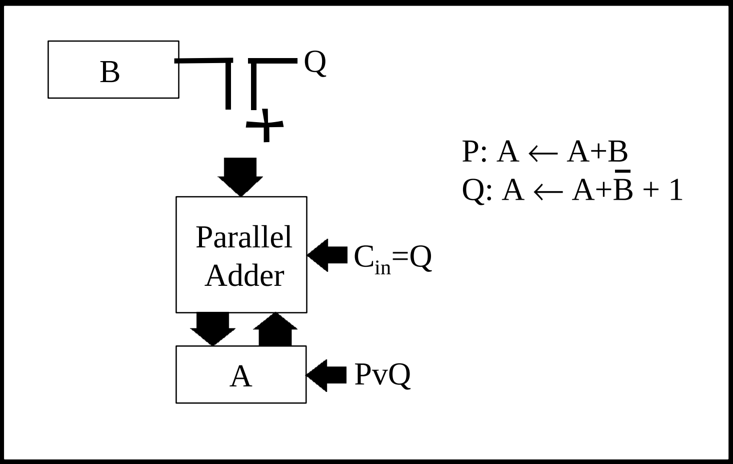
- ADD: $R3 \leftarrow R1+R2$
- SUBTRACT: $R3 \leftarrow R1-R2$
- COMPLEMENT: $R2 \leftarrow \overline{R2}$
- 2'S COMPLEMENT: $R2 \leftarrow \overline{R2}+1$
- SUBTRACT (2C): $R3 \leftarrow R1+\overline{R2}+1$
- INCREMENT: $R1 \leftarrow R1+1$
- DECREMENT: $R1 \leftarrow R1-1$

Describe operations individually. Subtract (2C) is subtraction performed by adding the two's complement of the number to be subtracted, i.e. x-y is implemented as x+(-y).

For R1 = 1001 1100 and R2 = 01010110, the results of the operations are:

ADD:	1111 0010
SUBTRACT:	0100 0110
COMP:	1010 1001
2'S COMP:	1010 1010
SUB (2C):	0100 0110
INCR:	1001 1101
DECR:	1001 1011

Hardware to implement A ± B



P: $A \leftarrow A+B$

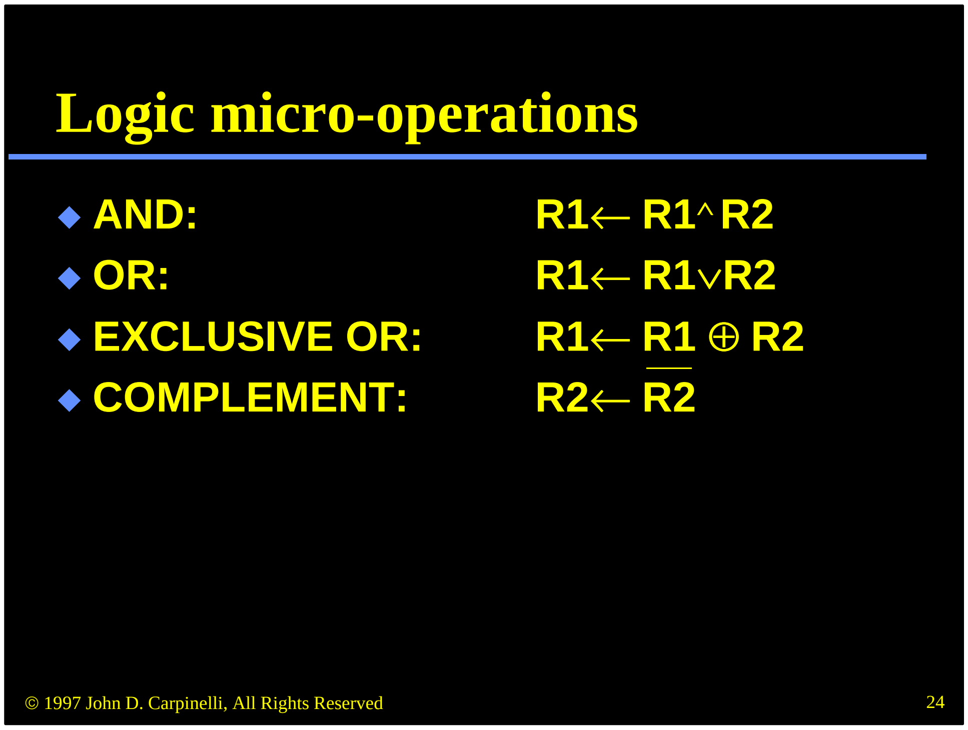
Q: $A \leftarrow A+\overline{B} + 1$

This circuit implemtns two micro-operations: P: A <-- A + B and

Q: A <-- A + B’ + 1, or A - B.   The parallel adder receives the inputs and generates the correct result. If P = 1, this result is A + B; if Q = 1 it is

A + B’ + 1. In either case, A must be one of the operands input to the parallel adder. The second operand will be either B, if P = 1, or B’, if Q = 1. We implement this by using a bank of exclusive-or gates, each of which has a single bit of B as one input and Q as the other input. If Q = 0,

B (XOR) Q = B (XOR) 0 = B; if Q = 1, B (XOR) Q = B (XOR) 1 = B’. Finally, by connecting Q to C_{in} of the adder, we take care of the +1 part of the Q micro-operation.



Logic micro-operations

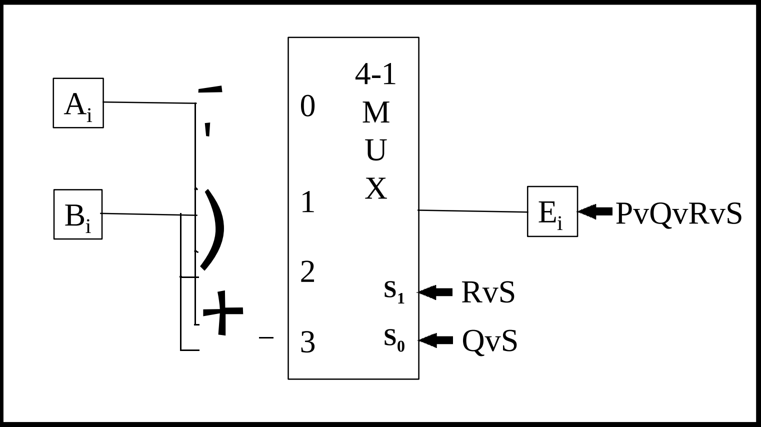
- AND: $R1 \leftarrow R1 \wedge R2$
- OR: $R1 \leftarrow R1 \vee R2$
- EXCLUSIVE OR: $R1 \leftarrow R1 \oplus R2$
- COMPLEMENT: $R2 \leftarrow \overline{R2}$

Other logic operations, such as nor, nand, etc., can be generated using the logic micro-operations listed above. Depending on the CPU under design, some of these additional logic micro-operations may be implemented explicitly in hardware.

For R1 = 1001 1100 and R2 = 01010110, the results of the operations are:

AND:	0001 0100
OR:	1101 1110
XOR:	1100 1010
COMP:	1010 1001

Logic hardware



This system implements one bit of the following micro-operations:

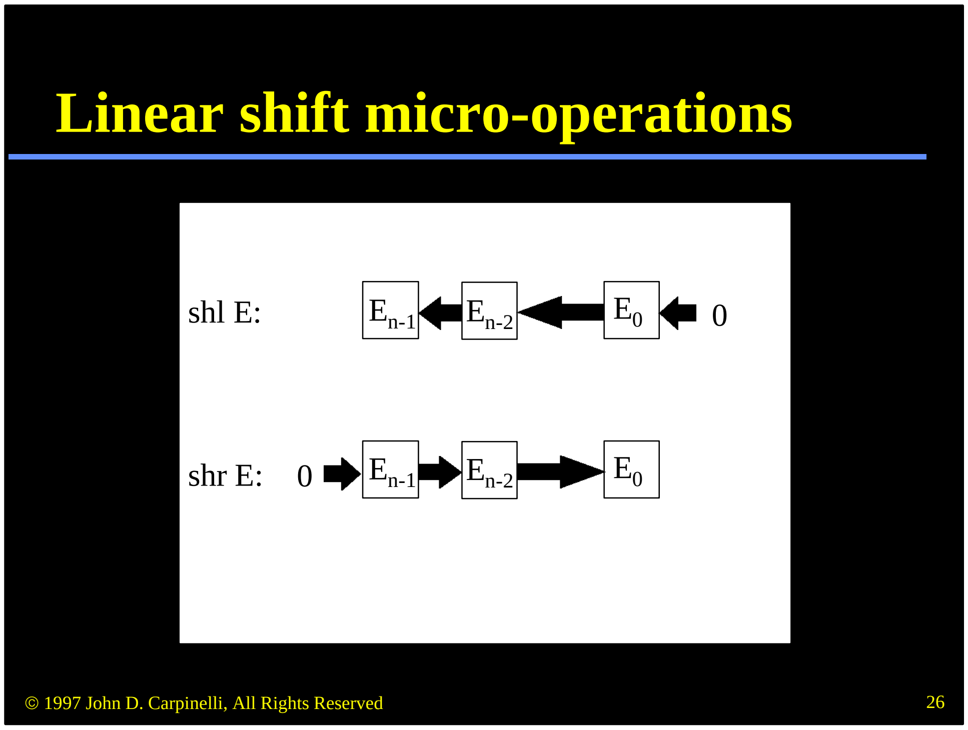
P: E <-- $\overline{A}$

Q: E <-- A ^ B

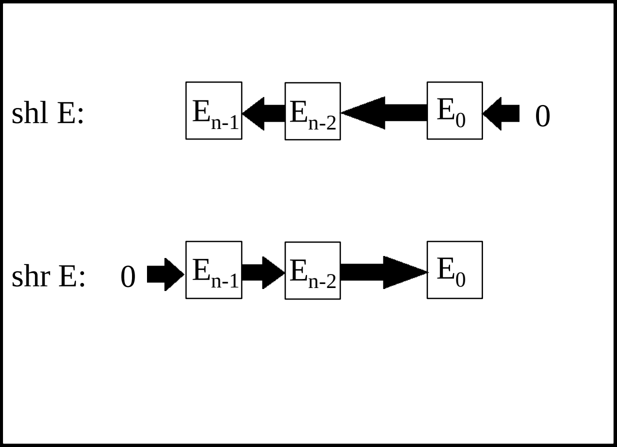
R: E <-- A v B

S: E <-- A (XOR) B

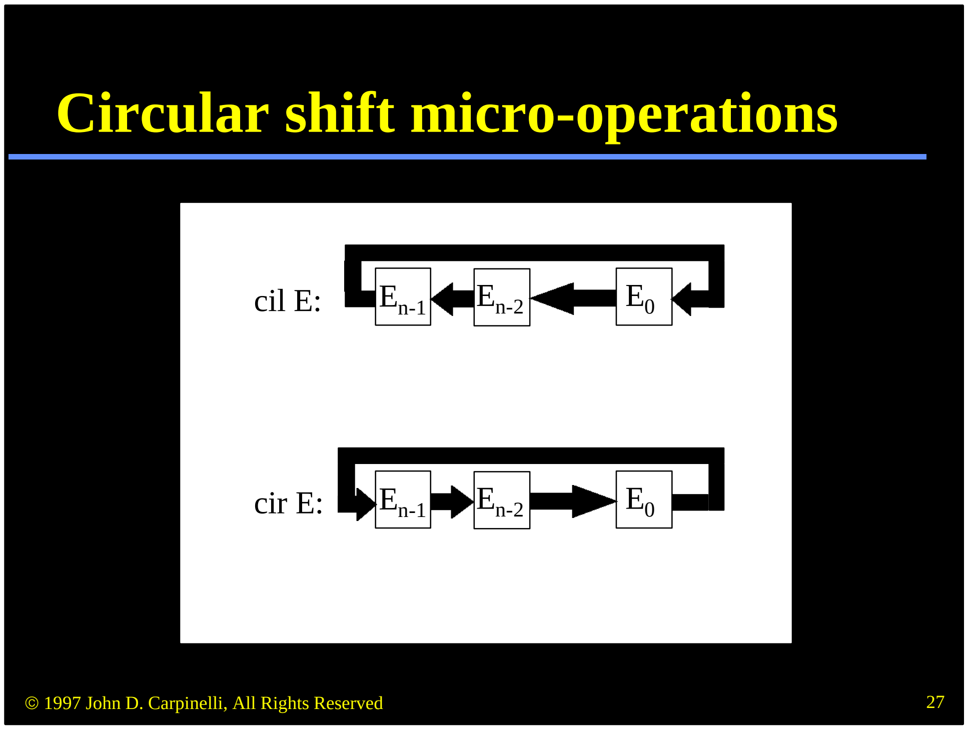
Two control signals select one of the four logic micro-operations to be passed through to E_i. If no logic micro-operation is to occur, one of the values is still passed though; additional hardware will make sure that this output is ignored by the rest of the CPU.



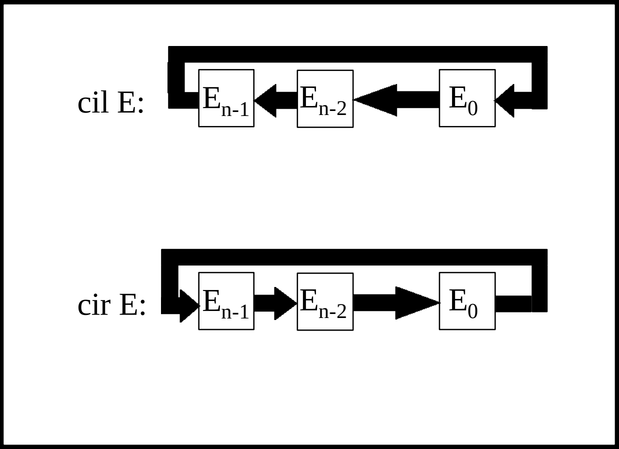
Linear shift micro-operations



For linear shifts, each bit is shifted one position to the left or right, as appropriate. A zero is shifted into the vacated bit. One bit's information is lost in the shift.

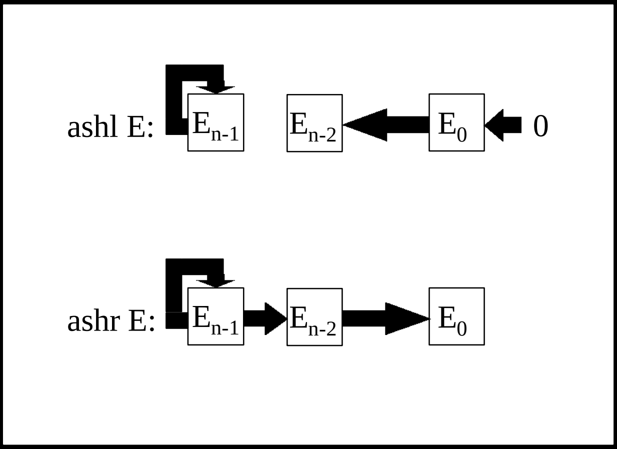


Circular shift micro-operations

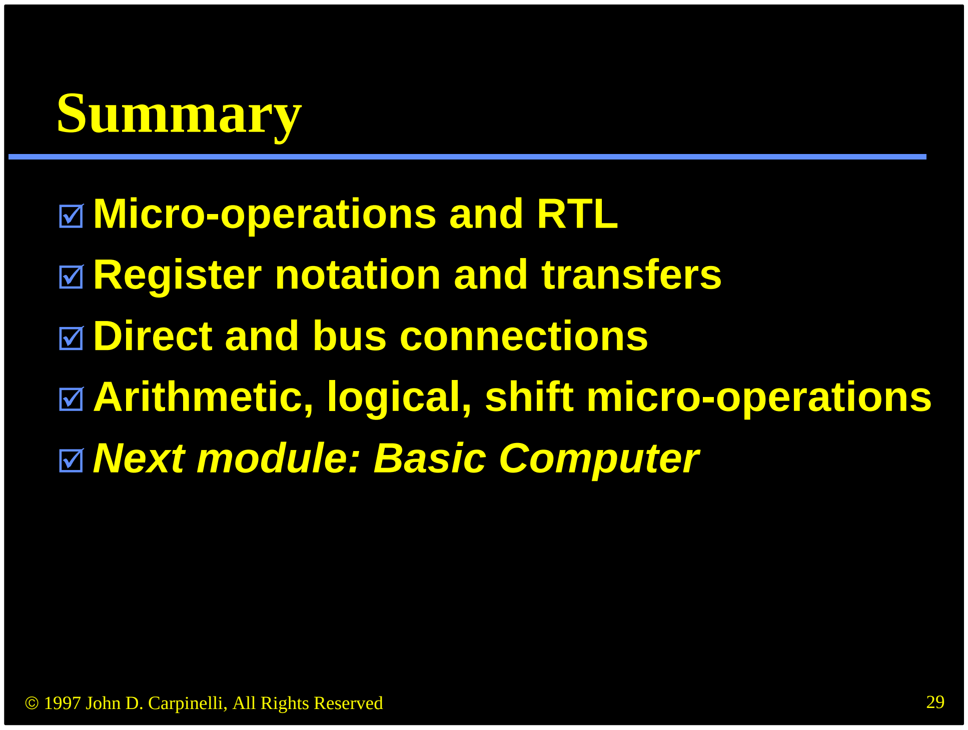


Circular shifts act exactly like linear shifts, except the bit formerly lost is circulated back into the vacated bit instead of a zero.

Arithmetic shift micro-operations



For arithmetic shifts, the most significant bit is treated as a sign bit that cannot be modified. Many arithmetic algorithms, as we will see later in this course, must use arithmetic shifts. The left arithmetic shift acts like the shl, except the most significant bit is unchanged and the next to most significant bit is lost. For the right arithmetic shift, the sign bit is not only retained, it is also shifted right as in the shr.



Summary

- ☑ Micro-operations and RTL
- ☑ Register notation and transfers
- ☑ Direct and bus connections
- ☑ Arithmetic, logical, shift micro-operations
- ☑ *Next module: Basic Computer*

This chapter has covered various types of register transfers. It also showed how to implement these transfers. Micro-operations which cover some of the data transfer and manipulation operations that can be performed in a CPU and the hardware to perform them were discussed.

In the next module, we will proceed to a full design of the Basic Computer and put some of these ideas into practice.