

MIT OpenCourseWare  
<http://ocw.mit.edu>

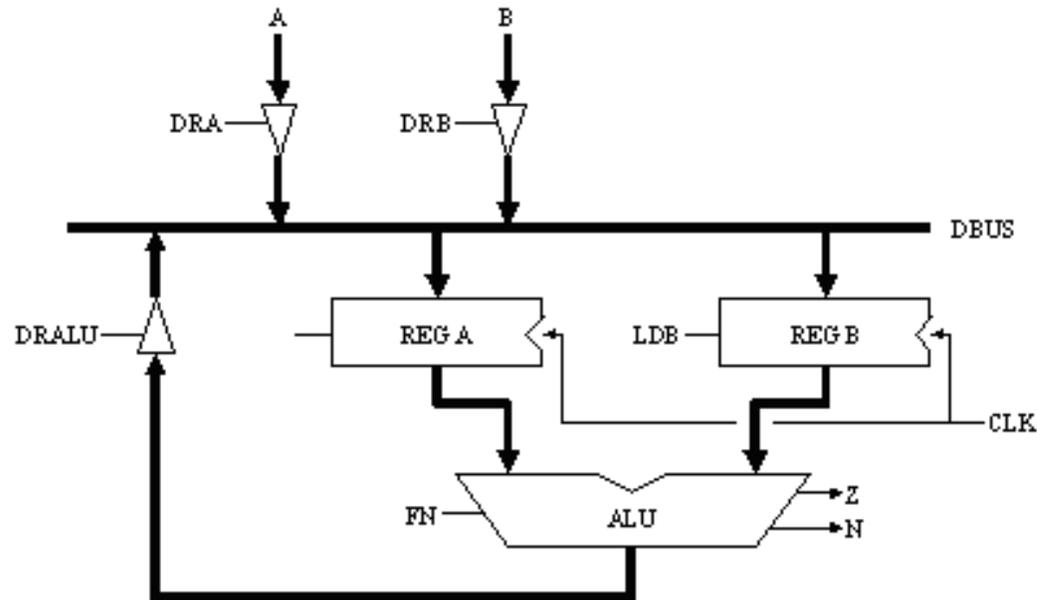
6.004 Computation Structures  
Spring 2009

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

# Programmable machines

---

Problem 1. Consider the following circuit:

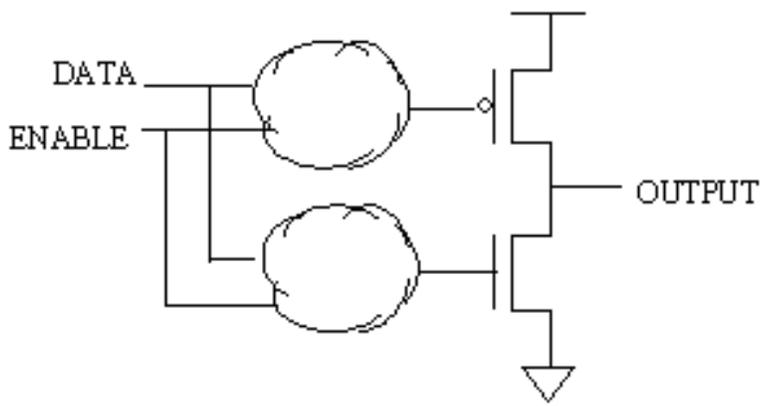


The heavy lines represent busses, which are many signals grouped together, e.g., an eight-bit bus is eight separate signal wires that are treated as a group. When a bus connects to one or more components, it's just shorthand for drawing the individual signal wires between those components.

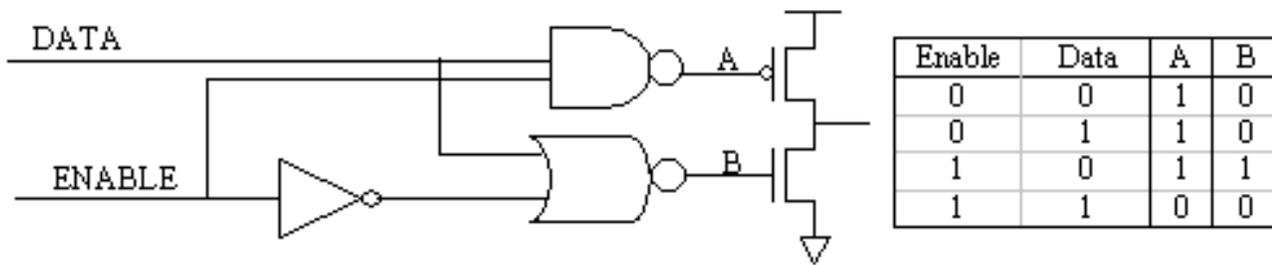
- A. The components with triangular schematic symbols are tristate drivers that operate like buffers except that they have an additional control input called an enable. When the enable is high, the buffer is on and the input value is driven onto the output. When the enable is low, the buffer is off and doesn't drive anything onto its output (i.e., the output pin is in a high-impedance state). What rules should the designer follow when designing the logic that generates DRA, DRB and DRALU in order to ensure that the DBUS signals always have legitimate values?

The designer should have at most one of the signals DRA, DRB, or DRALU asserted during any clock cycle. While these signals are being computed, it is possible that more than one might be asserted simultaneously due to logic glitches or computation paths of different length. The designer should take care to eliminate or at least minimize this occurrence.

- B. ★ Draw a schematic showing how a tristate driver might be implemented using mosfets. Hint: The following schematic shows one way of implementing a tristate driver.

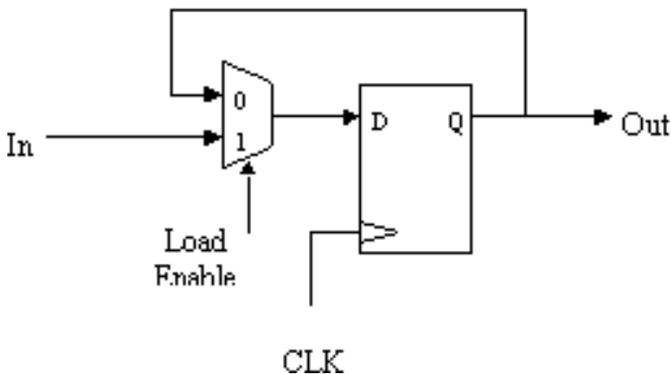


You just have to fill in the logic inside each of the clouds-think about for what values of DATA, ENABLE you want the pullup to be on and replace the upper cloud with one or more logic gates that implement that equation. Ditto for the pulldown and lower cloud.



- C. ★ The register-like symbols labeled "Reg A" and "Reg B" also have an additional enable input and are called load-enabled registers. When the enable is high, the register will be loaded from the incoming data. When the enable is low, the register reloads itself with its previous value. Show how to implement a load-enabled register from a regular D-register and a 2-way multiplexer.

Just add the mux before the register to clock in either the old value or the new value



- D. It's considered bad practice to control the loading of a register by "gating" its clock, i.e., by adding some logic that controls whether or not the register sees a rising clock edge. Briefly explain why "gating the clock" is discouraged. Hint: consider the effects of clock skew and logic hazards.

"Gating" the clock is not good practice for 2 reasons:

(1) *The possibility of glitches in the load enable signal.* Combinational logic often controls the load enable signal, so there may be periods of time when the load enable momentarily changes value. If load enable is supposed to remain low, but changes to a high value while CLK has a high value, then the register will see a rising clock edge and sample its inputs. This unwanted sampling of its inputs may cause the register to remember incorrect values, or even worse, enter a metastable state if the inputs are changing.

(2) *Clock skew.* Placing a logic gate in front of the CLK input of a register introduces delay, as the logic gate has intrinsic delay. This delay skews the clock signal, and as we have seen, clock skew can require slower clocking of the circuit or could cause hold time requirements to be violated.

E. The arithmetic-logic unit (ALU) has two data inputs (A and B) and, in this circuit, can perform only two operations, based on the single control signal FN:

when  $FN = 0$ , the ALU output is  $A - B$

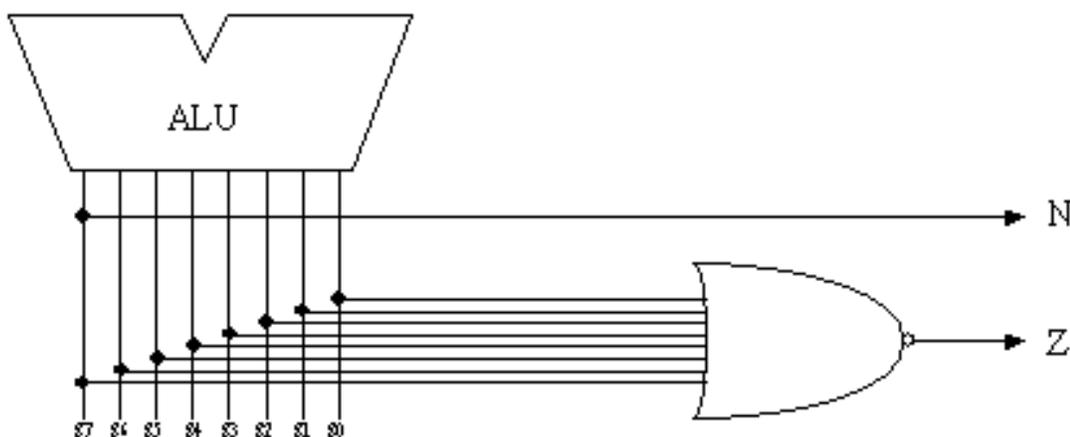
when  $FN = 1$ , the ALU output is  $B - A$

The ALU also generates two condition codes which give us some additional information about the ALU output:

$Z = 1$  when the ALU output is the number "0"

$N = 1$  when the ALU output is a negative number

Assuming that we have 8-bit data values and use a two's complement representation for the data values processed by the ALU, draw gate-level schematics for the logic that generates the Z and N signals from the ALU output value.



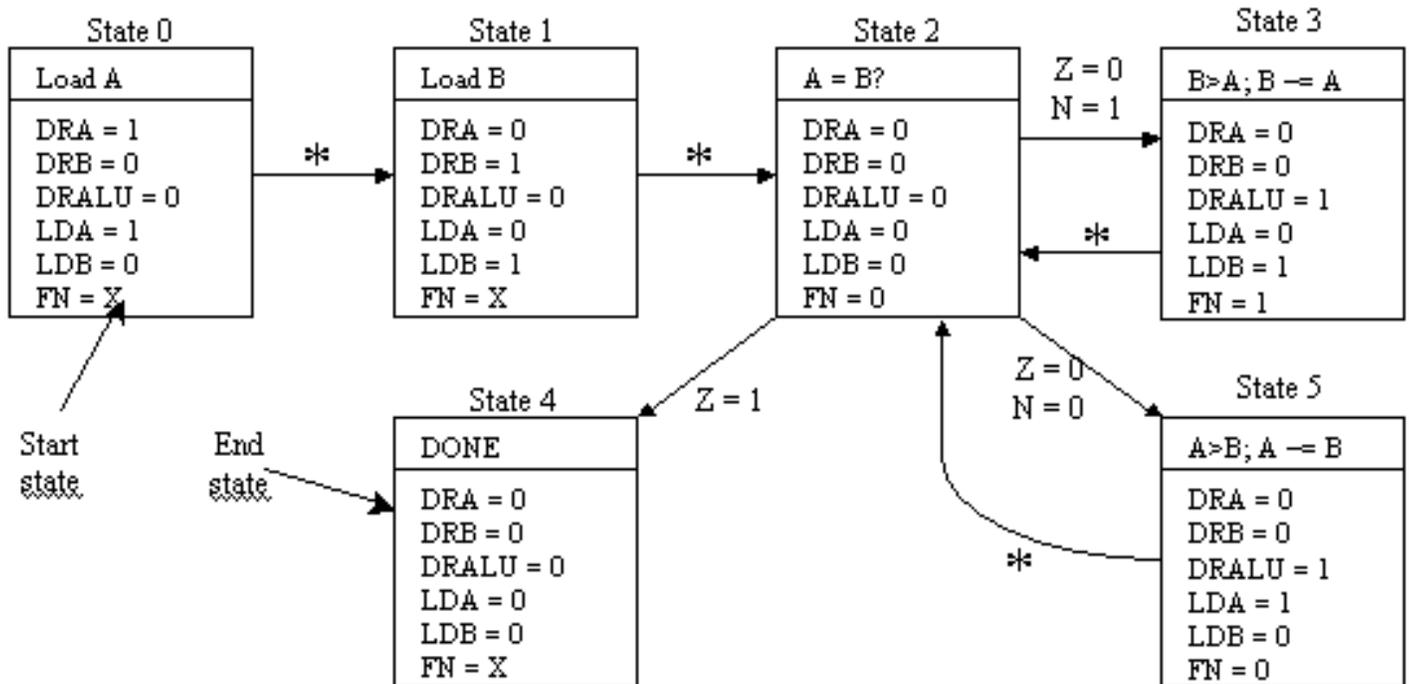
F. ★ Your job is to build a controller that will cause the circuit above to execute the following algorithm which computes the greatest common divisor of two inputs:

```
while (a != b)
    if (a > b) a = a - b;
```

else b = b - a;

The controller will be a state machine that takes 2 bits of input (Z and N) and produces control signals for the data paths (DRA, DRB, DRALU, LDA, LDB, FN).

Draw a state diagram for the controller. Outputs from your FSM should depend only on the current state. Indicate which are the initial and final states of your FSM on the diagram.

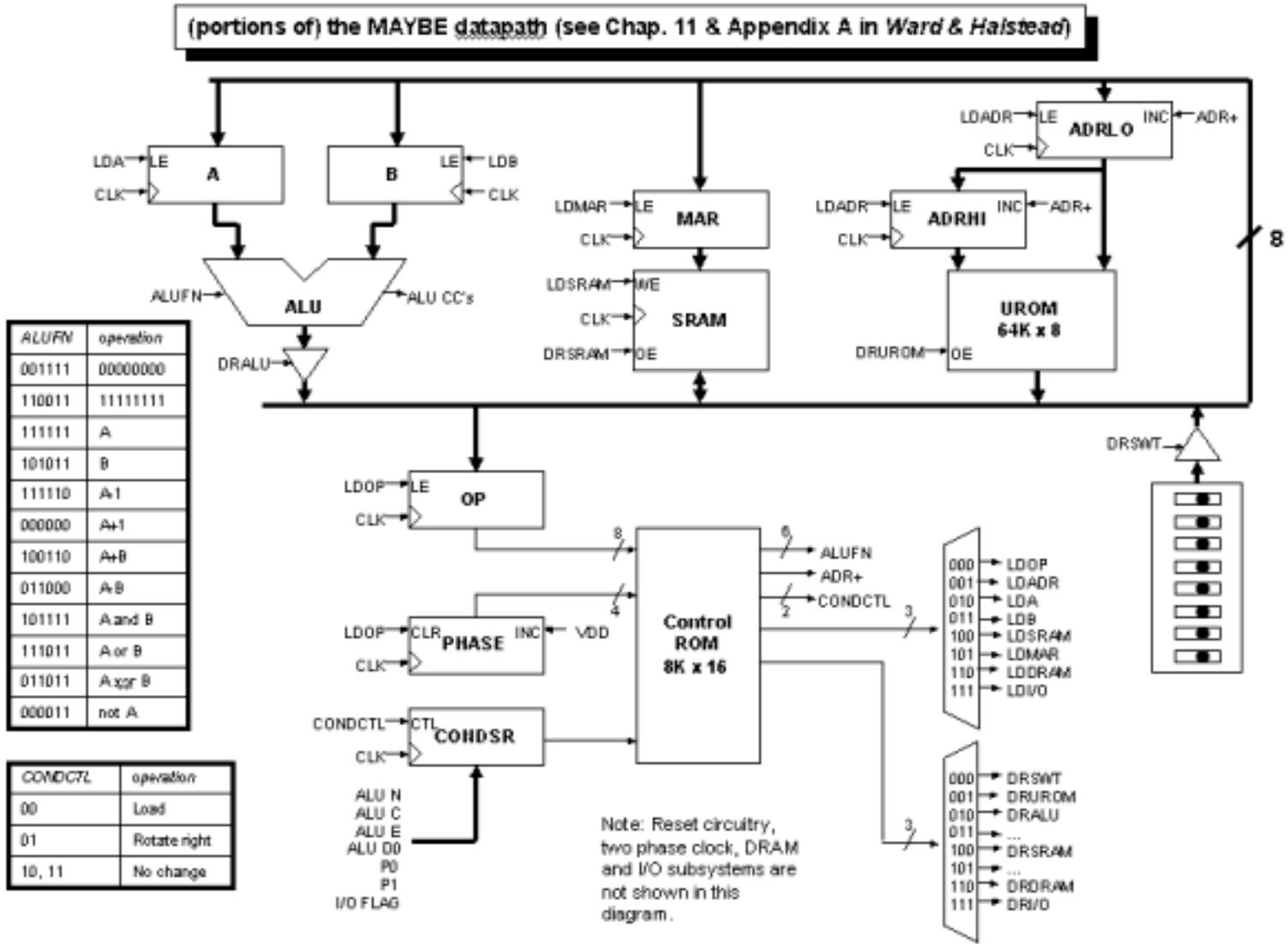


G. ★ Supply a truth table for the logic that generates the control signals.

Z	N	Current state	Next state	DRA	DRB	DRALU	LDA	LDB	FN
-	-	0	1	1	0	0	1	0	X
-	-	1	2	0	1	0	0	1	X
0	1	2	3	0	0	0	0	0	0
1	-	2	4	0	0	0	0	0	0
0	0	2	5	0	0	0	0	0	0
-	-	3	2	0	0	1	0	1	1
-	-	4	4	0	0	0	0	0	X
-	-	5	2	0	0	1	1	0	0

**Problem 2.** The material in this question will not be covered by any quizzes. It's presented here as an extended example of a programmable datapath.

The following diagram shows the datapath and control circuitry for a nifty little microprogrammed architecture the students used to build in the 6.004 lab:



Some features of the MAYBE:

- The datapath is 8 bits wide. All the subsystems connect to a common 8-bit bus which routes data between the subsystems. In a particular cycle, one of the subsystems is selected to drive data onto the bus (chosen by the DR signals) which can then be loaded into the selected destination (chosen by the LD signals).
- The Control ROM is programmed as a nanointerpreter that executes one of 256 microinstructions selected by the contents of the OP register. Each microinstruction consists of up to 16 nanoinstruction; the nanoPC is generated by the PHASE counter which is reset to 0 each time the OP register is loaded.

- The Control ROM is programmed using the following template:

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
*****	****	*	=	1	111111	11	001	000	Opcode = uROM; ADR+

"\*" is used to match either "0" or "1" on the corresponding address line. The template above initializes every nanoinstruction in the Control ROM to an operation that reads the next byte from the UROM (DRSEL = 001), places it in the OP register (LDSEL = 000), and then increments the ADRHI/ADRLO registers (ADR+ = 1).

- The UROM contains a microprogram that is executed by the nanointerpreter. Each microinstruction in the UROM begins with an 8-bit opcode; microinstruction execution begins by loading the opcode into the OP register. As the nanointerpreter executes the nanocode associated with that opcode, additional operand bytes may be fetched from the UROM. The final step in executing a microinstruction is to load the OP register with the opcode of the next microinstruction.
  - The ADRHI/ADRLO registers form a 16-bit program counter for the UROM program. These registers can be loaded from the data bus (eg, when doing a branch) or simply incremented by asserting ADR+. Note that the ADRHI/ADRLO registers are incremented at the end of the cycle after they have been used as the address for the UROM in the current cycle.
- A. If we inadvertently switch connections on two of the wires that run from the MAR register to the address inputs of the SRAM will operation be affected? Is your answer the same if we switch two wires running between ADRHI/ADRLO and the address inputs to the UROM? If the answers are not the same, what constitutes the difference between the SRAM and UROM?

Reordering connections between the MAR and the address inputs of the SRAM won't have any observable effect since there will still be a unique location for each possible MAR value (the actual location in SRAM will change but who could tell?).

Switching the ADRHI/ADRLO connections could be detected since after the switch incrementing the registers would not fetch the immediately adjacent location. If we also permuted the contents of the UROM to match the change in address wiring, the switch would not be detectable.

- B. What, if anything, prevents two drivers from putting conflicting data on the data bus of the MAYBE (ignore transients during propagation delays of the control circuitry)? Can such conflicts happen if there are programming errors in the Control ROM?

The data bus drivers are controlled by a 3-to-8 decoder. For any given 3-bit input, this device asserts only one of its outputs, so no conflicts are possible. Programming errors might result in the wrong value being driven onto the bus, but never multiple values at the same time.

- C. Given a big enough Control ROM, could the LDSEL and DRSEL decoders be eliminated (producing the load and drive signals directly as Control ROM outputs)? If so, what advantage might this have?

Yes, simply replace each 3-bit control value that drives the decoder with an 8-bit value that connects directly to the LDxx or DRxx control signals. Now each signal could be asserted independently, perhaps in concert with other signals. This isn't useful for the DRxx signals (see answer to previous question), but would allow several registers to be loaded with the same data bus value simultaneously.

- D. Execution of a nanoprogram can be influenced by information for the datapath? Explain how a nanoprogram can make data-dependent decisions.

The low-order control ROM address bit comes from a shift register that is loaded with condition codes from the ALU. Non-data-dependent nanoinstructions are loaded twice into consecutive even/odd locations of the control ROM, so the address bit from the shift register will select the same instruction regardless of whether it's 0 or 1. However if different instructions are loaded into the even/odd locations, the nanoprogram will execute differently depending on the output of the shift register.

By shifting the shift register before executing the data-dependent nanoinstruction, it's possible to use any of the latched condition codes.

- E. Given a big enough Control ROM, could the condition shift register be eliminated (using the condition bits directly as Control ROM inputs)? If so, what advantage might this have? How many more (or fewer) outputs and inputs would the Control ROM need to have to implement this? What would be the size (in bits) of the Control ROM?

In theory if we used the 7 condition code signals as additional address signals, we could test all 7 bits at the same time and execute one of 128 different instructions as a result. We might want to add a latch-enabled register to capture the signals on some specific cycle and save them for testing at some later cycle (the current design does this using the CONDCTL signals).

With 7 additional address inputs the control rom would grow from  $2^{13}$  locations to  $2^{19}$  locations. Assuming we need to control a latch-enabled register to capture the signal values, the number of control outputs would decrease by 1 (CONDCTL would go from 2 bits to 1).

- F. The nanoinstruction shown above selects the UROM as the data source and asserts ADR+ during the same clock cycle. How does this work, i.e., is the original or incremented address used when accessing the UROM?

The increment happens at the *end* of the clock cycle (ie, at the next rising edge of the clock), so for this current clock cycle the original address is used.

G. What does the following nanocode program do?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
00001010	0000	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00001010	0001	*	=	0	111111	11	100	010	A = SRAM
00001010	0010	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00001010	0011	*	=	0	111111	11	100	011	B = SRAM
00001010	0100	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00001010	0101	*	=	0	100110	00	010	100	SRAM = A + B; latch CCs
00001010	0110	*	=	1	111111	11	001	000	OP = uROM; ADR+

Implements the "ADD(X,Y,Z)" microprogram instruction which stores the sum of SRAM locations X and Y into SRAM location Z. ADD has an opcode of 00001010 and takes 7 cycles to execute.

H. What does the following nanocode program do?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
00001011	0000	*	=	1	111111	11	001	010	A = uROM; ADR+
00001011	0001	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00001011	0010	*	=	0	111111	11	100	011	B = SRAM
00001011	0011	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00001011	0100	*	=	0	100110	00	010	100	SRAM = A + B; latch CCs
00001011	0101	*	=	1	111111	11	001	000	OP = uROM; ADR+

Implements the "CADD(CX,Y,Z)" microprogram instruction which stores the sum of the constant CX and SRAM location Y into SRAM location Z. CADD has an opcode of 00001011 and takes 6 cycles to execute.

I. What does the following nanocode program do?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
00001100	0000	*	=	1	111111	11	001	010	A = uROM; ADR+
00001100	0001	*	=	0	111111	11	001	001	ADR = uROM
00001100	0010	*	=	0	111111	11	010	001	ADR = A
00001100	0011	*	=	1	111111	11	001	000	OP = uROM; ADR+

Implements the "JMP(addrlo,adrhi)" microprogram instruction which changes the microcode program counter to the specified address. JMP has an opcode of 00001100 and takes 4 cycles to execute.

J. What does the following nanocode program do?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
00010111	0000	*	=	1	111111	11	001	010	A = uROM; ADR+
00010111	0001	*	=	0	111111	01	010	010	Shift CC's
00010111	0010	1	=	0	111111	11	001	001	ADR = uROM
00010111	0011	1	=	0	111111	11	010	001	ADR = A
00010111	0100	1	=	1	111111	11	001	000	OP = uROM; ADR+
00010111	0010	0	=	1	111111	11	001	010	A = uROM; ADR+
00010111	0011	0	=	1	111111	11	001	000	OP = uROM; ADR+

Implements the "JNC(addrlo,adrhi)" microprogram instruction which changes the microcode program counter to the specified address if the carry bit (captured from the ALU by some previous microinstruction) is not set. JNC has an opcode of 00010111 and takes 4 cycles to execute if the branch is not taken and 5 cycles to execute if the branch is taken.

K. What does the following nanocode program do?

Opcode	Phase	COND	=	ADR+	ALU	CC	DRSEL	LDSEL	Comment
00000011	0000	*	=	0	110011	11	010	101	MAR = 11111111
00000011	0001	*	=	0	111111	11	100	010	A = SRAM
00000011	0010	*	=	0	111110	11	010	100	SRAM = A - 1
00000011	0011	*	=	1	111111	11	001	101	MAR = uROM; ADR+
00000011	0100	*	=	0	111111	11	100	011	B = SRAM
00000011	0101	*	=	0	111111	11	010	101	MAR = A
00000011	0110	*	=	0	101011	11	010	100	SRAM = B
00000011	0111	*	=	1	111111	11	001	000	OP = uROM; ADR+

Implements the "PUSH(x)" microprogram instruction which decrements the microstack pointer (stored in SRAM location 255 = 0xFF) and then stores the contents of SRAM location x in the SRAM location pointed to by the microstack pointer. PUSH has an opcode of 00000011 and takes 8 cycles to execute.