Logic Gates

Name	Truth Table	Schematic	Boolean Notation	Circuit Diagram (Transistors or Logic Gates)
NAND	A B Q 0 0 1 0 1 1 1 0 1 1 1 0		$Q=\overline{A\cdot B}$	NMOS NAND gate
NOR	A B Q 0 0 1 0 1 0 1 0 0 1 1 0		$Q=\overline{A+B}$	NMOS NOR gate
NOT	A Q 0 1 1 0		$Q=\overline{A}$	NMOS NOT gate; PMOS NOT gate
AND	A B Q 0 0 0 0 1 0 1 0 0 1 1 1		$Q=A\cdot B$	NMOS AND gate; PMOS AND gate
OR	A B Q 0 0 0 0 1 1 1 0 1 1 1 1		$Q=A+B$	NMOS OR gate; PMOS OR gate
XOR	A B Q 0 0 0 0 1 1 1 0 1 1 1 0		$Q=A\oplus B$	
XNOR	A B Q 0 0 1 0 1 0 1 0 0 1 1 1		$Q=\overline{A\oplus B}$	

Notes: You can build any logic gate out of NAND or NOR gates, as shown for the XOR and XNOR gates in the table above using NAND gates. When converting to and from an AND to a NAND, or an OR to a NOR in a circuit, you must put a NOT gate after the new gate to compensate for the difference between the two gates. Adding two NOT gates in a row does nothing to a circuit. A single input connected to the two inputs of a NAND or a NOR is simply a NOT gate. The transistor circuits shown above are discussed later.

© 2007 Review-Mate

Boolean Logic Algebra

The "+" sign signifies an OR gate and the "·" sign signifies an AND gate. The following are properties:

$$x\cdot y=y\cdot x \quad x+y=y+x \quad x\cdot(y\cdot x)=(x\cdot y)\cdot z \quad x+(y+z)=(x+y)+z \quad x\cdot(y+z)=x\cdot y+x\cdot z$$

$$x+y\cdot z=(x+y)\cdot(x+z) \quad x+x\cdot y=x \quad x\cdot(x+y)=x \quad x\cdot y+x\cdot\overline{y}=x \quad (x+y)\cdot(x+\overline{y})=x$$

$$x+\overline{x}\cdot y=x+y \quad x\cdot(\overline{x}+y)=x\cdot y \quad 0\cdot 0=0 \quad 1+1=1 \quad 1\cdot 1=1 \quad 0+0=0 \quad 0\cdot 1=1\cdot 0=0$$

$$1+0=0+1=1 \quad \overline{\overline{x}}=x \quad x\cdot 0=0 \quad x+1=1 \quad x\cdot 1=x \quad x+0=x \quad x\cdot x=x \quad x+x=x \quad x\cdot\overline{x}=0 \quad x+\overline{x}=1$$

DeMorgan's Rule: $\overline{x\cdot y}=\overline{x}+\overline{y}$ or $\overline{x+y}=\overline{x}\cdot\overline{y}$. This can be used to re-write logic functions, or to re-draw circuit diagrams. Ex. You can convert a NAND to two NOTed inputs ORed together.

Determining Logic From Tables

SOP (Sum of Products) using Minterms: a minterm (denoted m) is a product (AND) of input variables such that each variable is a 1 (if $x_1=0$ then use $\overline{x}_1$, or if $x_2=1$ then use x_2) and thus this product evaluates to 1. The Sum of Products technique is then an OR of all the minterms to get the desired output. (example below)

POS (Product of Sums) using Maxterms: a maxterm (denoted with capital M) is a sum (OR) of input variables such that each variable is a 0 (if $x_1=0$ then use x_1, or if $x_2=1$ then use $\overline{x}_2$) and thus the sum evaluates to 0. The Product of Sums technique is then an AND of all the maxterms to get the desired outputs. (example below)

x y z	Output (f)	Minterms	Maxterms
0 0 0	1	$m_0=\overline{x}\overline{y}\overline{z}$	
0 0 1	1	$m_1=\overline{x}\overline{y}z$	
0 1 0	1	$m_2=\overline{x}y\overline{x}$	
0 1 1	0		$M_3=x+\overline{y}+\overline{z}$
1 0 0	1	$m_4=x\overline{y}\overline{z}$	
1 0 1	1	$m_5=x\overline{y}z$	
1 1 0	0		$M_6=\overline{x}+\overline{y}+z$
1 1 1	1	$m_7=xyz$	

Notice that the 0's and 1's are always written from 0 to 8 in binary. Always do this when making tables.

SOP: $f=m_0+m_1+m_2+m_4+m_5+m_7$ since this is the sum of products that evaluate to 1's. Another notation is: $f=\sum m(0,1,2,4,5,7)$.

POS: $f=(M_3)(M_6)$ since this is the product of sums that evaluate to 0's. Another notation is: $f=\prod M(3,6)$.

Note: $\text{minterm}=\overline{\text{maxterm}}$ and vice versa.

Once SOP and POS are written out, draw out the implementation of the circuit that uses the least amount of logic gates. In the above example, the POS would be chosen. To build these circuits, place the inputs or their compliments into the max/minterms and then connect these outputs to a single large AND or OR gate.

CMOS Transistors

PMOS transistors: make up the pull up network of a CMOS circuit. This is called pull up because it brings the output, L, to the V_{CC} voltage which is typically 5V. A PMOS transistor acts as a switch that is closed when the input is 0 and open when the input is 1. To realize a PMOS circuit, take the function, for example for a NOR ($f=\overline{x+y}=\overline{x}\,\overline{y}$) and separate it into individual variables. Then notice that PMOS transistors take the compliment of the inputs already, thus in this example just AND (put in series) each of the PMOS transistors, connecting the bottom end to L and the top to V_{CC}.

NMOS transistors: make up the pull down network of a CMOS circuit. This brings the output, L, to ground (0V). A NMOS transistor acts as a switch that is closed when the input is 1 and open when 0. To realize a NMOS circuit, use the opposite of a function, for example use $\overline{f}=\overline{\overline{x+y}}=x+y$ for a NOR, and separate it into individual variables. Then notice that NMOS transistors don't use the compliments of the inputs, thus for this example just OR (put in parallel) each of the NMOS transistors, connecting the top to L and the bottom to ground.

V_{CC}
Pull up Network (NMOS)
inputs
$\to L$
Pull down Network (NMOS)
ground

The number of transistors required to make the common CMOS logic gates are:
AND/OR = 2 x # of inputs + 2 to invert, NAND/NOR = 2 x # of inputs, NOT = 2 to invert

Karnaugh Maps

A technique used to minimize a logic function so that minimum circuit elements are needed. The steps are:

1. Draw a table with the number of cells equal to the number of values in the output function. The number of columns and rows should be multiples of 2, such as a 2 x 2 table, 2 x 4 table, or 4 x 4 table.
2. Write out one or two variables across the top and one or two along the side as headings, always writing each heading as only one bit away from the next, such as 0, 1 or 00, 01, 11, 10.
3. Fill in the table with the function's outputs as 0's or 1's in each cell.
4. Group together the largest number of 1's as possible using only rectangles in sizes of 2^k such as 1,2,4,6,etc. You must group all the 1's no matter how many rectangles it takes. The rectangles can be made by wrapping around the edges of the Karnaugh map because these maps are cyclic. Look at which variables stay constant for each rectangle and if they are constant at 1 then use that variable or if they are constant at 0 then use the compliment of that variable to write out a minterm for SOP. This can also be done by grouping 0's and using maxterms for POS. Always use the largest rectangles as these need the lease number of variables.

d\wx	00	01	11	10
0	0	0	1	1
1	0	1	1	1

Example (w, xd)

5. **Done Care's:** if some of the outputs do not matter, then when writing the values into each cell, leave them as d's. When making rectangles, you can simply include these and make them 0 or 1 as needed.
6. Write out the functions together as a product for POS or a sum for SOP as done before.

Ring Oscillators

Are composed of an odd number of NOT gates arranged in a circle after which an output is taken from one position. An odd number is needed since an input pulse of 1 will be 0 the first cycle, 1 the next cycle and so on. If an even number was used, the same input would be output always. The period of oscillation is given by:

$$period(T)=2\times(gate\,delay)\times n$$

Where n is the number of oscillators in the ring. The 2 is there since each NOT must go through two changes of value (and therefore gate delays) before the oscillator completes one period (changes back to current value).

Binary and Hexidecimal Numbers

To convert between decimal and binary numbers where $b_n \ldots b_1$ are the digits 0 or 1 in binary:

$$\#\,in\,decimal=b_n\cdot 2^n+b_{n-1}\cdot 2^{n-1}+\cdots+b_1\cdot 2^1+b_0\cdot 2^0$$

To convert between decimal and hexadecimal numbers where $h_n \ldots h_1$ are the digits 0-9, A-F in hexadecimal:

$$\#\,in\,decimal=h_n\cdot 16^n+h_{n-1}\cdot 16^{n-1}+\cdots+h_1\cdot 16^1+h_0\cdot 16^0$$

Multiplexers (MUXs)

An selector input, s_0, can be used to select an output from the inputs depending on its value. In the diagram shown, when $s_0=0$ then A is selected or when $s_0=1$ then B is selected. This is the case for a 2-to-1 multiplexer as shown here, but larger multiplexers can be made by having n selector inputs for 2^n inputs, each being selected by a binary combination of the n selector inputs.

B A
1 0
S0
Z

The logic function of a 2-to-1 MUX is: $z=\overline{s}_0A+s_0B$.

A 2-to-1 MUX can be used to make an AND, OR, or NOT:

AND: let B = b, A = 0, and $s_0=a$, then $z=\overline{a}\cdot 0+ab=ab$.

NOT: let B = 0, A = 1, and $s_0=a$, then $z=\overline{a}\cdot 1+a\cdot 0=\overline{a}$.

OR: let B = 1, A = b, and $s_0=a$, then $z=\overline{a}b+a\cdot 1=a+b$.

Latches, Flip-Flops, and Their Uses

Name/Description	Truth Table	Circuit Layout	Schematic
SR Latch: the most basic memory unit. It is composed of NOR gates. R resets the Q output to 0, and S sets it back to 1.	R S / Action 0 0 / Keeps value 0 1 / $Q=1$ 1 0 / $Q=0$ 1 1 / $Q=0$		
Gated SR Latch: this has a clock, E, so that the S and R inputs can only pass through when the clock is activated.	Clk R S / Action 0 d d / Keeps value 1 0 0 / Useless 1 0 1 / $Q=1,\ \bar{Q}=0$ 1 1 0 / $Q=0,\ \bar{Q}=1$ 1 1 1 / $Q=\bar{Q}=0$		
D Gated Latch: stores the input, D, for one cycle when the clock(C/E) switches to 0. $Q=Q$ when $C=0$, $Q=D$ when $C=1$.	Clk D / Action 0 d / Keeps value 1 0 / $Q=\bar{Q}=0$ 1 1 / $Q=1,\ \bar{Q}=0$		
D Flip-Flop (DFF): stores the input value at each negative clock edge (only once per period). Pos trig. is made if NOT is moved to other D latch.	Clk D / Action Falling 0 / $Q=0$ Falling 1 / $Q=1$		Neg Trig/Pos Trig
JK Flip-Flop: J sets and K resets at pos edge of Clk since the input is $D=J\bar{Q}+\bar{K}Q$. If both on, the output is inverted.	J K / Action 0 0 / Keeps Q 0 1 / $Q=0$ 1 0 / $Q=\bar{Q}+Q=1$ 1 1 / $Q=\bar{Q}$	Not Needed In This Course	
T Flip-Flop: when T is 0 the input is kept, when T is 1 the input is inverted at pos edge of Clk.	Clk T / Action Rising 0 / Keeps value Rising 1 / Flips value	Not Needed In This Course	

The following are some applications of flip-flops and latches:

Name/Description	Circuit Implementation
Asynchronous Count-Down: placing a 1 value at the input to a chain of n T flip-flops will count down from 2^n-1 to zero, each number triggered at the pos edge. Since there is a delay caused by one output being the clock of the next flip flop, the more gates there are, the longer until the output bytes will be realized. The last bit (q2) is the most sig. bit in the binary count.	

Synchronous Count-Down: this solves the clocking problem found in the previous example by using AND gates. The AND gate is activated when q0 and q1 are both 1, thus resetting the value of q2. Notice that all the clocks are the same.

Asynchronous Count-Up: similar to the count down where Q from each flip flop gives the output bits, but the $\bar{Q}$ value is used for the next flip flop's clock. When q0 is 0, then q1 flips, and so on.

Synchronous Count-Up: similar to the sync. Count-down in that AND gates are required. The AND gate is activated when q0 and q1 are both 1, which then flips q2. All the clocks are again identical.

Shift Register: a single input x is shifted through a series of DFF's from one to the next at each positive clock edge. The Q from each flip flop is then an output, thus n DFF's are needed to have n bits of output, which will take n clock cycles to load.

Register: simply n DFF's with a common clock, n inputs into their D inputs and n outputs from their Q outputs. This is the same design as the Shift Register but each Q is not connected to the following D in a normal register. The circuit symbols for both are shown to the right with the Register on the left and the Shift Register on the right.

Synchronous Up-Counter with Cutoff: implemented for a 2-bit count-up by connecting the q1 output back through a NOT gate to TFF1's input, and through an OR gate with TFF1's output to the input of TFF2.

Synchronous UP/Down Counter: implemented using multiplexers to select Q or $\bar{Q}$ as the input to each T.

Adders

Half Adder: this simply adds two bits, x and y giving a sum as, $s_i = x_i\bar{y}_i + \bar{x}_i y_i = x_i \oplus y_i$ and a carry over bit, $c_{i+1} = x_i y_i$ which is required to carry a second bit when $x = y = 1$.

N-Bit Ripple Carry Adder: composed of N full adders in series with each carry out being the next adder's carry in. This adds two N bit numbers $(x_0,\ldots,x_{n-1})$ and $(y_0,\ldots,y_{n-1})$.

Full Adder: this involves a carry in bit and the other bits from the half adder: $c_{i+1} = x_i y_i + x_i c_i + y_i c_i$, $s_i = c_i\bar{x}_i\bar{y}_i + \bar{c}_i x_i \bar{y}_i + \bar{c}_i\bar{x}_i y_i + c_i x_i y_i$, or $s_i = x_i \oplus y_i \oplus c_i$.

A full adder can be made from two half adders as shown to the right.

Fast Adder: this adder looks ahead using a SOP circuit by using, $c_1 = x_0 y_0 + x_0 c_0 + y_0 c_0$ and subbing this into $c_2 = x_1 y_1 + x_1 c_1 + y_1 c_1$ and so on. This look ahead technique can reduce the time needed to calculate sums since the initial carry in, c_0 can be used to calculate all future carry outs, instead of having to wait for each step to be

Signed Numbers

Sign Bit: most significant bit now represents either positive (0) or negative (1). The range is: $\left(-2^{n-1}+1, 2^{n-1}-1\right)$.

1's Compliment: simply flip all of the bits. Ex. $-6=-(00110)=11001$. The range is: $\left(-2^{n-1}+1, 2^{n-1}-1\right)$.

2's Compliment: flip all the bits and add 1. Ex. $-6=-(00110)+1=11001+1=11010$. Range: $\left(-2^{n-1}, 2^{n-1}-1\right)$.

Another method to find the 2's compliment is to find the first 1 from the right side of the binary number, and then to flip all the digits to the left of that 1.

Only 2's Compliment works in all cases for addition and subtraction of positive and negative numbers. Note: having a signed bit reduces the maximum value of that number of binary bits since one less bit can be used. See the following example: 111 is: 7 when given as unsigned, -3 when given as signed, -(000)= -0 when given as 1's comp., and -(000+001)= -(001)= -1 when given as 2's compliment.

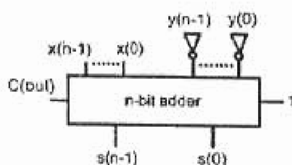
Note: converting between a number given in 1's or 2's compliment to a binary number is the same as converting a number into 1's or 2's complement from a binary number respectively. Also, in all cases of signed bit numbers, the 1 in the signed bit represents negative, and the positive version of the numbers are all the same.

Over Flow in Adders: this occurs when c_n and c_{n-1} are not equal. $overflow = c_n \oplus c_{n-1} = \bar{c}_n c_{n-1} + c_n \bar{c}_{n-1}$ where c_n is the signed bit and c_{n-1} is the carry from the MSB before the signed bit.

Subtraction: is carried out for 2's compliments by:

$X-Y=X+(-Y)=X+(\text{flipped } Y)+1$ and for 1's compliments by:

$X-Y=X+(-Y)=X+(\text{flipped } Y)$. During subtraction of two numbers, if a carry out occurs from the signed bit position then ignore it if you are using 2's compliment numbers, or add it to the lease significant bit for 1's compliment numbers.



Subtractor: simply an n-bit adder with one set of inputs inverted and a 1 in the carry in place (since 2's compliment #'s). This is shown to the right:

Adder/Subtractor: adding a series of multiplexers to one set of inputs and having one input to each MUX being the original input value, while the second input being the inverse of the input value, therefore allowing you to switch between an adder (original inputs) and a subtractor (inversed inputs) by setting the MUX selector value.

Finite State Machines (FSMs)

Moore FSMs: outputs in this type of FSM are tied to a state. Next state depends on the inputs and current state.

Mealy FSMs: outputs and next states in this type of FSM depend on the current state and the inputs. This type looks ahead even though it is in one state at the current time, the output can be changed by changing inputs.

State Bits: are binary digits that assign a number to keep track of each state, these are denoted by $s_0, s_1, \ldots$. The next state is also assigned state bits, denoted as $s_0^+, s_1^+, \ldots$. The next state is selected on each clock pulse.

State Diagram: displays each state as a circle and the transitions between states as arrows. For Moore FSMs the outputs are within the states, while in Mealy the outputs are along the arrows. Inputs are along arrows in both.

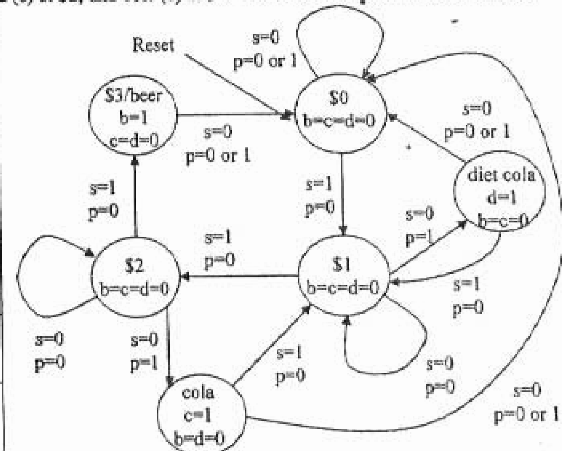
State Transition Table: shows the relationships between the current state, next state, inputs and outputs. Note: that in the Mealy table, the outputs will depend on the inputs as well as the current state.

Karnaugh Maps: after the state transition table is made, the next state bits and the output bits are determined using Karnaugh maps. The current state bits and the inputs are placed as the row and column headings to determine a FSM's next state bits and outputs, except in a Moore FSM where only the current state bits are used to find the outputs. Using the resulting SOP or POS functions created from the maps, the functions are created on a series of DFFs, one for each state bit. Actually the input to each DFF is one of the next state bits (ex. s_0^+), while the output is the matching current state bit (ex. s_0). This is done since these output bits loop back through logic gates to the inputs of the DFFs again, controlling the next states of the FSM. The output bits are realized through logic gates connected to various places in the circuit using either current/next state bits or inputs (Mealy).

© 2007 Review-Mate

The following shows Mealy and Moore implementations for a vending machine FSM. The machine only accepts one dollar increments denoted input, s, and will dispense when the button, p, is pushed. Three outputs of the machine are diet cola (*d*) at $1, regular cola (c) at $2, and beer (b) at $3. The **Moore implementation** is first:

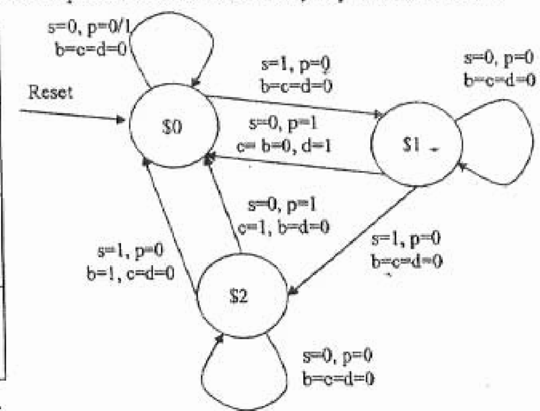
State	State Bits $s_2 s_1 s_0$	Input s p	Next State $s_2^+ s_1^+ s_0^+$	Output d c b
$0	000	00 01 10	000 000 001	0 0 0
$1	001	00 01 10	001 011 010	0 0 0
$2	010	00 01 10	010 100 101	0 0 0
d	011	00 01 10	000 000 001	1 0 0
c	100	00 01 10	000 000 001	0 1 0
b	101	00 01 10	000 000 ddd	0 0 1



Notice there was only 1 output for 4 possible inputs in each state. Also the s p = 11 input was not written here. This was simply to save space. Normally you write this state, even though not possible as given in this question, and write Don't Care's (d) for all next states and outputs as was done for the Mealy Implementation below.

Mealy Implementation:

State	State Bits $s_1 s_0$	Input s p	Next State $s_1^+ s_0^+$	Output d c b
$0	00	00 01 10 11	00 00 01 dd	0 0 0 0 0 0 0 0 0 d d d
$1	01	00 01 10 11	01 00 10 dd	0 0 0 1 0 0 0 0 0 d d d
$2	10	00 01 10 11	10 00 00 dd	0 0 0 0 1 0 0 0 1 d d d



Notice how in the Mealy Implementation the outputs depend on each combination of inputs for each current state.

The next step is to draw the Karnaugh map for each output and next state bit. This will only be shown for Mealy.

© 2007 Review-Mate

Since in Mealy, the next state and outputs depend on the current state and on the inputs, all of these must be factored into the Karnaugh map for the next state bits and the outputs. (in Moore FSMs you don't factor in the inputs while determining the outputs, but you do when finding the next state bits). Below this is shown for s_1^+:

$s\,p \backslash s_1 s_0$	00	01	11	10
00	0	0	d	1
01	0	0	d	0
11	d	d	d	d
10	0	1	d	0

This Karnaugh map gives: $s_1^+ = ss_0 + \overline{s}\overline{p}s_1$.

Doing this for all of the outputs and the other state bits gives:

$s_0^+ = \overline{s}\overline{p}s_0 + \overline{s}_1\overline{s}_0 s$, $d = ps_0$, $c = ps_1$, and $b = ss_1$. This is then implemented on two DFFs with the input to the first one being $s_0^+ = \overline{s}\overline{p}s_0 + \overline{s}_1\overline{s}_0 s$ and its output being s_0, while the input of the second being $s_1^+ = ss_0 + \overline{s}\overline{p}s_1$ and s_1 is its output.

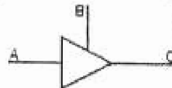
As you can see from these functions, the inputs of the DFFs are derived from the outputs of the DFFs, which makes sense since the next state (input) is derived from the current state (output). Thus connecting these along with the outputs through logic gates to s_0 and s_1, the finite state machine will be complete.

Multiplying: a finite state machine can be used to control a pair of adders that represent multiplication. Using adders, multiplication (for example if you want $X \times Y$) is done by adding X to another variable M, Y times. A common clock controls two registers that both have an enable on the clock input so that they can be operated on their own when needed. The inputs to the Y-register can be selected between the initial Y value, or the output from an adder in that circuit. The inputs to this adder are the Y register and -1, thus the adder is used to decrease Y from its initial value to zero. The second adder circuit has the M register which has inputs that can be selected as either zero to initialize, or the output of the adder in the circuit. The adder adds X to initially zero, and continues to add X to the sum until we know that $Y = 0$. To determine if $Y = 0$, you NOR all the bits of Y output from the Y register since if any one of them is a 1 (not zero overall) then this output bit will be 0. A FSM is used to control the clocks, and the MUXes that control the initial inputs to the registers.

Processor Basics

An in-depth description of a basic processor is discussed in class as is the code for the 68K processor so the only the basic components of a processor will be considered here.

Transmission Gate: is used as a switch, that is closed when B = 1, and is open when B = 0. These are used extensively to connect/disconnect processor components to the bus.



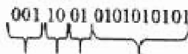
Bus: is a series of wires (n wires for an n-bit bus). These wires connect to most if not all the components of the processor such as registers, ALU units, memory, etc. This functions as the spine of the processor, transferring data to and from the components so they can communicate to each other. Each time data wants to be sent to the bus, a control bit must switch a transmission gate, then only the component with another control bit turning allowing the data in will actually use the date from the bus.

Check Bit: used to determine if there is any data on the bus. It is simply a large n input NOR gate, indicating 1 only if all the bits on the bus are 0.

Arithmetic Logic Unit (ALU): a unit that can add/subtract in simple processors (may do more complex calculations in complex processors). This does not need a clock to operate, just like adders/subtractors. An extra register is connected to the ALU and the bus so that you can preload one set of data in this and then get the second set of data directly from the bus for the add/subtract operations.

Program Counter (PC): acts as a pointer to an address in memory. An adder can be used to increase the PC either by a predetermined value from data, or simply by 1 to select the next address of memory.

Bits in Memory: typically arranged as:



where the first set of bits is used to indicate a software instruction such as add, sub, load, etc., the second set of bits indicates the source register, and the third set is the destination register. The remaining set is the data to be used by the operation. If either of the registers or the data are not needed for an operation, the corresponding bits are left as zeroes.

© 2007 Review-Mate

Verilog

Verilog: is a programming language used to model electronic circuits. There is a variety of new syntax to learn, too much to cover here, but it is summarized in tables in the textbooks for the course.

General Layout for a Moore FSM in Verilog:

```
module name(w, z, Clock, Resetn)
  input Clock, Resetn, w;                      defines what are inputs
  output z;                                    defines what are outputs
  reg [0:1]y, Y;                               registers y as current state, Y as next state
  parameter A=2'b00, B=2'b01, C2'b10;          sets constants for state names
  wire ...(if needed, not in this example)     used for connecting two devices

        //Define the next state combinational circuit
always @(w or y)                               changes whenever these parameters change
  case(y)                                      checks current state
    A: if(w == 1)   Y=B;                       next state based on the input
       else         Y=A;                       next state in else case
    B: if(w == 1)   Y=C;                       next state based on the input
       else         Y=A;                       next state in else case
    C: if(w ==1)    Y=C;                       next state based on the input
       else         Y=A;                       next state in else case
    default:        Y=2'bxx;                   default case
endcase

        //Define what happens on Clock (move to next State) and Reset
always@(negedge Resetn or posedge Clock)
  if (Resetn == 0)  y <= A;                    resets to state A
  else  y <= Y;                                assigns next state to current state reg as defined above

        //Define Ouputs here since they don't depend on inputs above in the first always block
assign z = (y == C)                            sets z=1 when in state C

endmodule
```

Note: you don't need "reg z" since you only need reg when changing a register in an always block.

© 2007 Review-Mate

General Layout for Mealy FSM:

```
module name(w, z, Clock, Resetn)
 input Clock, Resetn, w;                 define everything you need (same as Moore)
 output z
 reg y, Y, z;                            need reg z since it changes in an always block this time
 parameter A=0, B=1;
 wire ...(if needed, not in this example)

      //Define the next state and the output combinational circuit
      //You define outputs here because they depend on your inputs
always @(w or y)                         changes whenever these parameters change
 case(y)                                 checks current state
  A: if(input)
      begin  z=0;                        set output depending on input
             Y=B;                        next state
      end
      else
      begin  z=0                         set output depending on input
             Y=A;                        next state in this situation
      end
  B: if(input)
      begin  z=1;                        set output depending on input
             Y=B;                        next state
      end
      else
      begin  z=0                         set output depending on input
             Y=A;                        next state in this situation
      end
endcase

      //Define what happens on Clock (move to next State) and Reset (same as Moore)
always@(negedge Resetn or posedge Clock)
 if (Resetn == 0)  y <= A;               resets to state A
 else y <= Y;                            assigns next state as defined above

endmodule
```

Notes about all Verilog:

'b10 = binary number 10
'h10 = hexadecimal number 10
4'b100 = 4-bit binary number 0100
4'bx = unknown binary 4-bit number xxxx
wire A[7:0] means 8-bit wire. Access an element i by A[i].
Setting $A[7,0] = 7'b0100111$ sets each elements as: $A[7] = 0, A[6] = 1, A[5] = 0, etc.$
tri [7:0] makes an 8-bit three way connector.
Arrays can be set equal to each other, and each element will then be copied over between them.

© 2007 Review-Mate

Assembly on the 68K

Representing Numbers:
#1000 – is the decimal number one thousand
#$1000 – is 1000 in hexadecimal or, 4096 in decimal.

Instructions: are of the form: expression source, destination comment
For instructions such as add, would do: dest = dest + source.

Bit Sizes: Most instructions are 16-bit, but some may be larger depending on the size of the data required. Addresses are always 32-bit. In the 68K there are 8 data regs and 8 address regs.
A ".b" after an expression uses bytes (8-bit), ".w" after an expression uses words (16-bits), and ".l" after an expression uses long words (32-bits). If none are placed after an expression then the word size is the default.

Registers: Address registers on the 68K are used to store (or point) to address locations and data registers on the processor are used to temporarily store information so that operations such as addition and subtraction can easily be done to the numbers stored. All registers in the 68K are long word size, but the data registers d0,…,d7 can use bytes, words, or long words, and the address registers a0,…,a7 can use words and long words.

Labels: are all capital letters and represent an address.

org statement: used to tell the processor where the start of your code is. So in the below example, the first code following this org statement would start at address 1000 in hexadecimal (address 4096). Each number is one byte (8 bits) in memory.
org $1000

Addressing Modes: there are many ways to move data in assembly:

move #10,a0	a0=10, (address stored in register = 10)
move #100,(a0)	puts 100 into the location where a0 points to. Therefore, (10)=100. In other words (1000) means location 1000.
move.b #10,(a0)+	moves 10 to value pointed to by a0, then a0 is incremented by 1 to the next memory location. This is as follows, (a0)=10 then a0=a0+1. (a0)- decrement afterwards. If move.w was used then address would be incremented by +2 at the end, and +4 for move.l .
move #10,-(a0)	pre-decrements, thus a0=a0-2 (2 since a word), then (a0)=10.
move 5(a0),d0	d0=contents of memory location (a0+5). Useful for passing parameter into subroutines as will be discussed later. a0's address does not change after this instruction is complete.
move 100(a7,d0),d1	moves value at location, 100 + a7 + d0. Assume a7 = 1002, and d0 = 6, then the location is 1108 = 1002 + 6 + 100

An Example:

org 1000	starts at 1000th byte
move #21,11	places the number 21 into memory location 11.
move 0,d0	places 0 into the register d0.
move 11,d1	d1=(11)=21, puts the number 21 into the register d1.
add d0,d1	d1=d1+d0 therefore d1=21.
add d1,11	d1=d1+(11), adds 21 to 21 and stores in d1. So d1=42.
move d1,10000	puts 42 into memory location 10000.

Example Memory Sizes: all given in bytes:

Instruction	Size of Instruction	Extra Data	Reasoning For Data
org $1000			
move d0,d1	2	0	registers = 0
add.l #1,d2	2	4	long = 4
move d2,#2000	2	4	address = 4
cmp #1,d2	2	2	word = 2
beq LOOP	2	0	branches = 0

© 2007 Review-Mate

Conditional Branches:

cmp statement used to set the Status Register (SR, a register on the 68K processor reserved for storing compare values amongst other things) values for c,z,n, etc. to values depending on the arguments of the cmp statement. Example:

cmp d0,d1 sets SR bits based on d1-d0, but doesn't store result back in d1.

After a compare (cmp) statement you typically want to branch to another address using one of the following:

Operation (bxx) or (dbxx)	Braches if source is…destination
bge	Greater than or equal to
bgt	Greater than
blt	Less than
ble	Less than or equal to
bne	Not equal to
beq	Equal to
bra	Branches Always
bsr	Braches to Subroutine

As an example of the SR usage, bgt branches when z=0 AND n=0.

All these operations can be done using "db" instead of "b" which decrements the value and then branches to the location. For example:

dbra d1,#1000 decrements d1 by 1 and then branches to address #1000.

Analogy of Assembly with C Language:

Code in C Language	Code in Assembly (d0=a,d1=b)
a=2;	move #2,d0
for(b=10;b>0;b--){	move #10,d1
a=a+2;	LOOP7 add d0,d1
}	sub #1,d1
	comp #0,d1
//adds 2 to itself 10 times	bgt loop7

The 68K Stack: the bottom of the stack has the highest memory locations and the top has the lowest memory locations. So a push onto the stack requires a decrement of the address pointer. Address register "a7" is the address pointer reserved by the processor for stack operations. The stack uses last in first out (LIFO) ordering just as in the common C language stack.

0	
1	
2	
…	…
…	…
100	
101	
…	

Push: place or write to top of stack

move #10,-(a7) decrements the top of stack to empty location and then places ten as the value at that location

Pop: read or take off top of stack

move (a7)+,d0 increments the pointer to the one below the top after the value at the top of the stack has been placed into d0

© 2007 Review-Mate

Using the stack on the 68K processor to call subroutines:

You do not always need to push and pop to call subroutines in assembly with the 68K processor. The procedure is simplified by using the following two commands:

bsr adr pushes the address immediately after where this bsr is in the main program and branches to the address specified by "adr", (adr can be a LABEL).

rts pops the top 4 bytes (the return address) off the stack and jumps to it

Therefore simply type branch-to-subroutine (bsr) and specify the address where the subroutine starts in order to call the subroutine you want. Then at the end the subroutine you must have rts to get back to the next line where you left off in your program.

In order to pass parameters to the subroutines you can do one of two things. Place them in registers as these are available to all subroutines. Or you can place them on the stack and use the last addressing method shown above to get these values back. For example:

```
        move  #14,-(a7)     places 14 on the stack
        bsr   sqr           branches to sqr subroutine
        move  (a7)+,d1      assume this to be at address $1000 so program continues from here
...
sqr     move  4(a7),d0      skips the return address to get 14 off of stack and puts in d0
        mult  d0,d0         does an operation to d0
        move  d0,4(a7)      moves result back below return address on stack
        rts                 returns to address $1000 in main program
```

This program places 14 on the stack by decrementing the a7 pointer. Then branches to subroutine which puts the long word address assumed to be $1000 in this example onto the top of the stack and decrements the stack pointer, a7, by 4. Then in the subroutine, sqr, the 14 is retrieved by getting the value at (a7+4). The value is squared and returned to that same location. Then rts returns you back to address $1000 where the a7 register was left. The value held here (14^2-196) is then put into the data register d1, and then the top of the stack is moved down by 1.

Polling Input/Output: used to stop the main program from it's operations when a new input is ready for the processor, for example a keyboard button is pressed. Assume when $200000 has the value 0, no key is pressed, and when it is 1 then a key is pressed. Also assume the address $200001 is the value of the key pressed. Then the code may be:

```
wait    move.b   $200000,d0
        cmp.b    #0,d0
        beq      wait        keeps looping until key is pressed
        move.b   $200001,d1  stores which key is pressed
```

This method is resource intensive because the processor is constantly being used to determine if a key was pressed or not. A better method to use is interrupts.

Interrupts: an interrupt is a signal sent into the processor that when equal to 1 pauses the processor's current operation and begins to process an interrupt service routine (which first involves pushing the return address onto the processor stack, then pushing the status register onto the stack as well so that no information is lost for when you return to your original program that has been interrupted). The interrupt service routine is programmed in assembly and each specific interrupt (one per device) has a different interrupt service routine that is stored in the interrupt vector table. At the end of each interrupt service routine you must have an "rte" statement which acts like an "rts" statement, but also restores the status register which is saved when the interrupt occurs. The difference between a subroutine and an interrupt service routine is that you don't have to call the interrupt service routine like you do with an subroutine, and thus it can happen at any time when another program is running.

© 2007 Review-Mate

68K Memory Bus: there are 24 address bits (output from processor), 16 data bits, and 3 control bits that connect the 68K processor to memory and devices. The three control bits are:

MR = 1 when master is ready, output from processor
SR = 1 when slave is ready, output from device
$R/\overline{W}$ 1 for reading/0 for writing, output from processor

The address bits are set by the processor according to the address required as follows:
Assume address $7000 is needed, then this hexadecimal number is represented in binary as:

$a_{23}..a_0$ = 000000000111000000000000

0 0 7 0 0 0

So each four bits represent one hexadecimal number since they can equal up to 16. Therefore in this case: $a_{14}=1, a_{13}=1, a_{12}=1$.

Examples in relation to assembly:

move $1000,d0 reading from address

1. Proc: sets all address bits to correct values as determined by above method. $R/\overline{W}=1$ and MR = 1.
2. $d_{15}...d_0$ = contents of $1000 and $1001 (since a word is two bytes). SR = 1.
3. Processor sets MR = 0.
4. Memory sets SR = 0.

move #11,$2000 writing to address

1. Proc: sets all address bits to correct values as determined by above method. $d_{15}...d_0$ =0000000000001011= #11. $R/\overline{W}=0$ and MR = 1.
2. When write complete, SR = 1.
3. Processor sets MR = 0.
4. Memory sets SR = 0.

© 2007 Review-Mate

Ports

Ports are used to standardize the connection between a processor and a device since the address bits, data bits, and control bits may be different for each processor type. There are two types of ports that will be discussed, these are parallel ports and serial ports, both depicted on the following page.

Parallel Ports: the 68K processor's parallel port consists of 8 input lines of data, 8 output lines of data, one input ready signal, and one output ready signal. To setup the port the following must be done:

1. AND together all 24 address bits, with the bits that are supposed to be ones corresponding to straight wires, while bits that are supposed to be zeroes corresponding to wires that have NOT gates on them. The port itself has an address ($7000 in the examples in class), and this is the address that you need the bits to match. Therefore, only $a_{14}=1, a_{13}=1, a_{12}=1$ shouldn't have NOT gates.
2. The output of this large 24-bit AND gate is then sent to two, three input AND gates with the MR bit, and the $R/\overline{W}$ sent to each. The $R/\overline{W}$ will need to have a NOT gate before the output circuit AND gate (writing to the device connected to the port) and does not need a NOT gate for the AND gate on the input circuit (reading from device).
3. Let x be output from the read op AND gate, and y the output from the write op AND gate.
4. Input from the device is stored from $in_7...in_0$ in a register which has the input ready as the clock input.
5. The outputs of this register have transmission gates controlled by x, so that when x = 1, the inputs are copied to $d_7...d_0$, the data bits used by the processor.
6. When y = 1, the data bits are copied to $out_7...out_0$.
7. The SR (comes from port, not device) is connected to a transmission gate with the constant value of 1 where the input to this transmission gate is controlled in by x + y ("+" is an OR gate).

Examples of how parallel ports work:

move #22,$7000 writing to device connected to port

1. $a_{23}...a_0$=$7000, $R/\overline{W}=0$, MR = 1, $d_7...d_0$ = 22.
2. on port: y=1, x = 0, $d_7...d_0$ = 22, SR = 1. Someone connected to parallel port outside device reads #22 from port.
3. Processor sets MR = 0.

move $7000,d0 reading from device connected to port

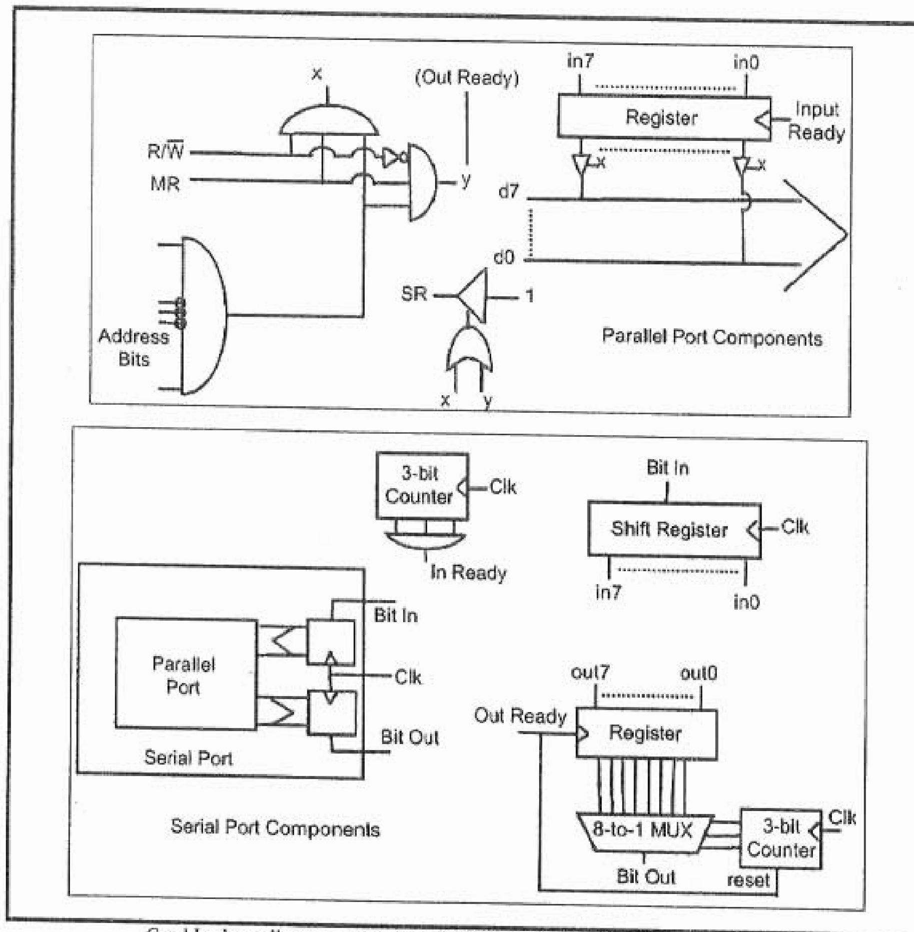
1. $a_{23}...a_0$=$7000, $R/\overline{W}=1$, MR = 1.
2. on port: y = 0, x =1 $d_7...d_0$ = 50 (what the connected device is displaying to the port), SR = 1.
3. Processor reads $d_7...d_0$ = 50, copies #50 to d0, and sets MR = 0.

move $6000,d0
Nothing happens with the port since $7000 is the only address used by the port for communications.

Serial Ports: only one bit in, one bit out, and a clock are needed since a serial port utilizes a parallel port for most of the functioning. The single input stream goes through an 8-bit shift register that has outputs connected to $in_7...in_0$ of the parallel port. The input ready of the parallel port is controlled by a 3-bit counter that counts up to 8 and which has its three inputs ANDed so that when 8 is reached (all 8 bits have been transferred through the shift register) the input ready becomes 1.

The $out_7...out_0$ bits connect to a regular register which has the clock coming from the output ready of the parallel port. The outputs of a second 3-bit counter control an 8-to-1 multiplexer that's inputs are the outputs from this register. This 3-bit counter has the output ready connected to its reset so that if there are no data bits on $out_7...out_0$ (thus output ready = 0 and the register in the serial port stores no data) then the 3-bit counter resets itself to zero. The output from the multiplexer is selected using this second 3-bit counter which cycles through 8-bits at a time outputting them one by one in a single wire which is the required single bit output.

© 2007 Review-Mate



Good Luck on all your exams. We wish you the very best in all your Academics!

© 2007 Review-Mate