Reference Sheet — CS M51A — Spring 2015

Ky-Cuong L. Huynh

10 June 2015

0.0.1 Specifications and Number Systems

What (reasonable) assumptions can we make about the inputs? Are there any "don't care" input cases? The bits of the output are specified separately. Specify the one-set, zero-set, and d.c.-set of each z-bit separately as well. $\underline{z} = z_2 z_1 z_0$. To determine how many bits are needed for z, check what the maximum value of the output will be and remember that n bits give a max value of $2^n - 1$. Normal flow: (1): Decide I/O and output function. (2): Select an encoding scheme. (3): Fill out a truth table or a K-map. (4): Minimize and get SEs. (5): Implement a network based on SEs. For multi-level networks: divide-and-conquer, factor expressions, bubble logic, decompose terms ($a+b+c+d = (a+b)+(c+d)$ and $abcd = (ab)(cd)$) to reduce fan-in, and buffer to increase output load, and try XOR gates.

For a decimal value x, we need at least $n = \lceil \log_{10}(x) \rceil$ bits. Base-2 to base-2^k: group the bits together into the same size group as needed to represent one digit of the new base, and then convert and concatenate groups. Example: `1110 0110` to radix-16: `0xE6`.

To convert from base-2^k to base-2, we do the opposite with digit expansion. Take each radix-8 or radix-16 digit and convert it into binary, then combine them into the final result. Example: `0xD7`, then we get `1101 0111`.

Odd parity: The number of 1's in the sequence should be odd. Have the parity bit be 0 or 1 to make it so. `0110110 => 0110101`. Even parity: The number of 1's should be even: `001110 => 0011101`.

0.0.2 Canonical Forms

SOP (which consists of **minterms**) from truth table: (1): Identify input rows that give 1 as output. (2): For each input, write either a for 1 a' for 0, and AND them together. (3) OR (+) this together with the minterm form of the next 1-row f is true if any of these minterms are true. Example: $m3 + m5 = 011 + 101$

POS (which consists of **maxterms**) from truth table: (1): Identify input rows for which the output is 0 (false). (2) For each input bit, write a for 0 or a' for 1, and OR the input bits together. (3) AND ($\cdot$) this maxterm together with the maxterm of the next 0-row f is false if any of these maxterms are false Example: $M3 \cdot M5 = 011 + 101 = (A + B' + C') \cdot (A' + B + C') = \Pi M(3, 5)$

Minterms to maxterms: Use the maxterms with complementary indices Example: $f(A, B, C) = \sum m(1, 3, 5, 6, 7) = \Pi M(0, 2, 4)$.

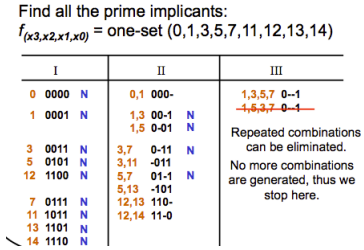
For maxterms to minterms: $f(A, B, C) = \Pi M(0, 2, 4) = \sum m(1, 3, 5, 6, 7)$.

From f to f', use minterms whose indices do not appear: $f(A, B, C) = \sum m(1, 3, 5, 6, 7) \implies f'(A, B, C) = \sum m(0, 2, 4)$.

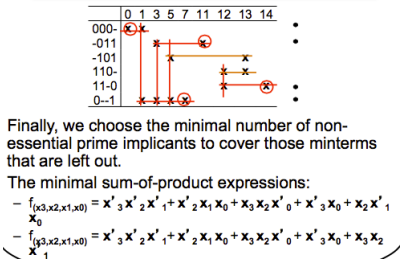
From f to f' for maxterms, $f(A, B, C) = \Pi M(0, 2, 4) \implies f'(A, B, C) = \Pi M(1, 3, 5, 6, 7)$.

0.0.3 Quine-McCluskey

1. Implicant table: List out implicants (binary form of one-set 1-cells) in first column, and group them by number of 1's. Pair up implicants that differ only by one bit, and check them off as non-primes. Keep the pair numbers to track which used.



2. Prime implicant chart: List out the prime implicants along the rows (with their associated minterm pairs), and the minterms along the columns. Mark an 'x' for the minterms that are covered by that prime implicant. Circle the essential prime implicants, i.e., the ones where a minterm only has that prime implicant covering it. Draw out horizontal lines from the essentials, and then vertically from the x's that are crossed out. Finally, write a minimal switching expression based on the essentials plus the needed primes (i.e., choose from the column entries) to cover any remaining minterms.

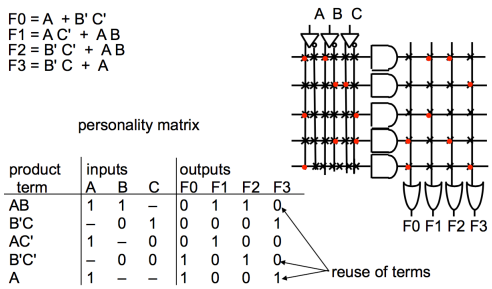


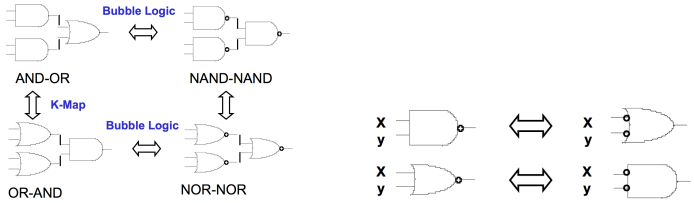
0.0.4 Design with PLAs

(1): Get a minimal POS expression (truth table to K-map to SE).

(2): Write out a personality matrix that lists each of the distinct product terms, their inputs, and the outputs they do/do not feed into. 1 if non-negated, 0 if negated, - if absent. For outputs: 1 if product term used, 0 otherwise.

(3): Place a dot to indicate connections, going top-down with product terms first, then OR'ing them together. The PLA diagram should follow the structure of the personality matrix.





0.0.5 Analysis

Validity analysis: Every gate input connected to a signal source, i.e., no floating wires? Only one source of signal to every input? Any loops, where an input into a gate eventually ends up at that gate again? Cost: Network size, power dissipation/consumption, monetary cost, etc.

Functional analysis: 1. Get SEs for network outputs in terms of network inputs. Divide and conquer into submodules, then define intermediate variables for their I/O, and work backwards from outputs to back-substitute to original inputs. Label gates starting from inputs, and apply bubble logic every other level (starting from output) to simplify negation-rich networks. 2. Get truth table for network outputs. 3. Define high-level I/O variables and use same encoding scheme as the low-level I/O. 4. Determine the high-level function of network.

0.0.6 Timing/Performance Analysis

(1):Find the critical path: number of gates (layers), delay by gate type (XOR > AND/OR > NOT), and fan-in (number of inputs) on each gate (more inputs ⇒ greater delay, as all inputs waited for).

(2):Decide the transition direction (HL or LH) on each gate, working backward from a starting assumption on the network output. E.g., if this AND gate gave a 1, it underwent a LH transition, and the OR gate feeding into it must have had a LH transition.

(3):Label each of the gates, working from the inputs. Write an expression for the total delay in terms of individual gate delays, starting from the inputs. Example: $T_{pLH}(x_1, z_2) = t_{pLH}(O_1) + t_{pHL}(N_1)$. Where x_1 is an abitrary input that fed into the eventual output of z_2.

(4): Plug-in propagation delay values from the datasheet. Pay attention to the the gate type (AND3 vs AND2), transition direction (LH vs. HL) and variable loads. We can compute latter by adding up load factors on gates it feeds into. Clarify on if the question wants this. Don't forget units of [ns].

0.0.7 Signed Arithmetic

SM of same sign: just add them. Of different signs: take one of the operands and complement it +1 to match signs, then add them, and apply the sign of the original operand that was largest. Example: $10010 + 01101 \Rightarrow 1110 + 1101 = 11011$ And we take the sign of the largest, so 01011 is our result.

Get 1's complement: Get binary code for $|x|$, then complement everything bit-wise. To convert to a decimal integer, see if the leftmost is 1. If it's not, treat it as a binary code. If it is, it's a negative, and complement everything before evaluating the magnitude. Example: 10011010 is a negative; complement and we get 01100101 which is 37.

Get 2's complement: get binary code for $|x|$, then complement bit-wise and add +1. To convert to a signed integer, if the leftmost bit is 1, then complement everything and add +1, then get the magnitude as a regular binary number.

Add two 1's comp. numbers: just add them. If there's a carry-out bit beyond the size of our operands, wrap it around and add it to the initial result. This will happen once, and then we're done. 2's comp. addition is the same, but throw away carry-out.

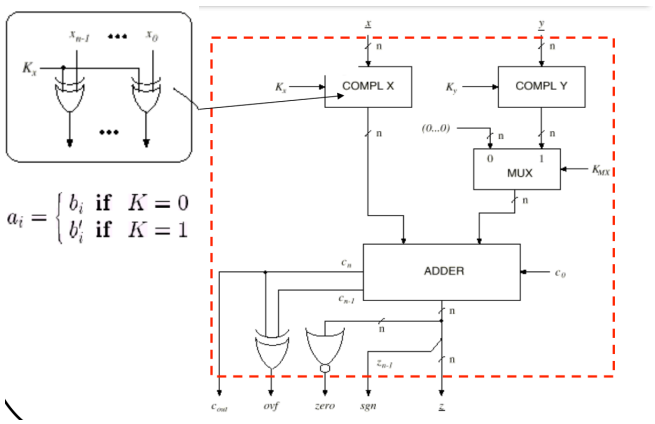
Overflow occurs when the last two carry bits differ, when the MSB suddenly flips signs from the two operands. Sign/range extension is when we copy the MSB repeatedly to get a larger number of bits that can avoid overflow: $0100 = 00000100; 1100 = 111111100$. Before carrying out operations, check what the largest possible value is, and then extend to match the number of bits needed to accomodate that.

Evaluate arithmetic expressions: (1): Rewrite the variables as powers of 2. (2): Range-extend the result if necessary (based on largest value produced in expressions). (3) Left-shift to multiply by 2, right-shift to divide by 2, and complement + 1 to negate (if 2's complement).

To rewrite something as a power-of-2: (1): Rewrite odds as even + odd: $9x = (8+1)x$. (2): Simplify terms until powers of 2 exist: $(8+1)x/2 = (8x+x)/2 = 4x + x/2$. (3): Rewrite powers of 2 explicitly, as those can be accomplished as left shifts.

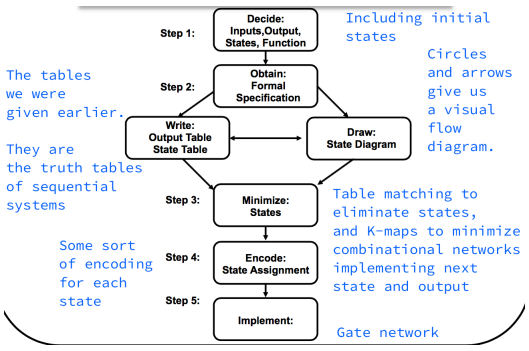
0.0.8 Arithmetic Modules

1-bit half adder (HA) has inputs x, y and outputs s, c (sum, carry). 1-bit means its operands are 1-bit. A 1-bit full adder (FA) has inputs x, y, c_{in} and outputs s, c_{out}. A n-bit full adder has inputs: two n-bit operands x, y and carry-in c_{in}, with outputs: z and c_{out}. If the operands are 2's comp. for full adder, no change, but carry wrap-around for 1's comp. (link carry-out to carry-in; loops at most once).



$$a_i = \begin{cases} b_i & \text{if } K = 0 \\ b_i' & \text{if } K = 1 \end{cases}$$

0.0.9 Sequential System Design



If canonical implementation: use D FFs as registers to store each state bit. Otherwise, use the FF excitation tables to figure out what inputs need to be generated by the combinational network

for next state. Output combinational network will be a function of current state (and inputs if Mealy).

0.0.10 State Minimization

(1): Get I/O, state, initial state, and state transition/output table. (2): Identify which rows of present states give the same outputs, and partition them into state-groups. (3): Fill out new state-transition table that lists which state-group an input-PS combination will jump to. (4): Identify states within groups whose column differs; they belong in their own group. Single item groups are automatically good-to-go. Continue until all columns within groups agree. Now, the states within groups are equivalent, and the minimal number of states is equal to the number of groups.

PS	$x=a$	$x=b$	$x=c$
A	$E,0$	$D,1$	$B,0$
B	$F,0$	$D,0$	$A,1$
C	$E,0$	$B,1$	$D,0$
D	$F,0$	$B,0$	$C,1$
E	$C,0$	$F,1$	$F,0$
F	$B,0$	$C,0$	$F,1$
	NS,z		

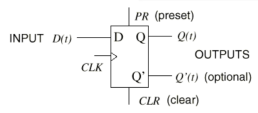
- Row Matching:
  - P_1 = (A, C, E) (B, D, F)
- Column Matching:

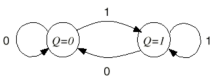
	1			2		
P_1	$(A,$	$C,$	$E)$	$(B,$	$D,$	$F)$
a	1	1	1	2	2	2
b	2	2	2	2	2	1
c	2	2	2	1	1	2

0.0.11 Sequential System Analysis

(1): Break loop at input into FFs. (2): Get SEs for output/next state in terms of PS and external inputs. (3): Make a state-transition table based on all possible PS/inputs based on char. eqns. of FFs. (4): Define encoding schemes for I/O and states. (5): Write a high-level I/O and states spec, plus a new state-transition/output table. (6): Draw a state diagram and/or time-behavior spec (a timeline of input $x(t)$ vs. current state $s(t)$ and output $z(t)$ at each time point).

D (data/delay) FF: basically acts like a register:





Block Diagram

$PS=Q(t)$	$D(t)$	
	0	1
0	0	1
1	0	1
	$NS=Q(t+1)$	

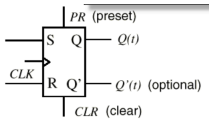
State Diagram

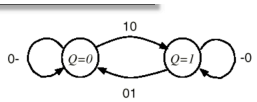
PS	NS	
	0	1
0	0	1
1	0	1
	$D(t)$	

$Q(t+1)=D(t)$ — Characteristic Equation

$D(t)=Q(t+1)$ — Excitation Equation

SR (set/reset) FF: either set current state to 1, or reset it to 0, but never both (conflicting):





Block Diagram

$PS=Q(t)$	$S(t)R(t)$			
	00	01	10	11
0	0	0	1	-
1	1	0	1	-
	$NS=Q(t+1)$			

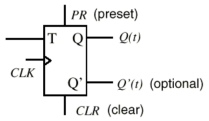
State Diagram

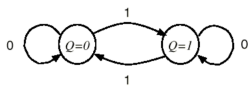
PS	NS	
	0	1
0	0-	10
1	01	-0
	$S(t)R(t)$	

$Q(t+1)=Q(t)R'(t)+S(t)$, $R(t)\cdot S(t)=0$ — Characteristic Equation

Excitation Equation

T (toggle) FF: If 1, then flip the current state, else leave it the same:





Block Diagram

$PS=Q(t)$	$T(t)$	
	0	1
0	0	1
1	1	0
	$NS=Q(t+1)$	

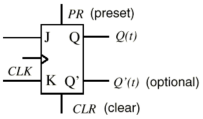
State Diagram

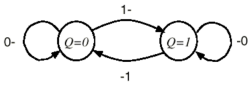
PS	NS	
	0	1
0	0	1
1	1	0
	$T(t)$	

$Q(t+1)=Q(t)\oplus T(t)$ — Characteristic Equation

$T(t)=Q(t)\oplus Q(t+1)$ — Excitation Equation

JK is like SR, but with the 11 condition meaning "toggle", and 00 meaning "hold". It is a universal flip flop; the others can be made with it.





Block Diagram

$PS=Q(t)$	$J(t)K(t)$			
	00	01	10	11
0	0	0	1	1
1	1	0	1	0
	$NS=Q(t+1)$			

State Diagram

PS	NS	
	0	1
0	0-	1-
1	-1	-0
	$J(t)K(t)$	

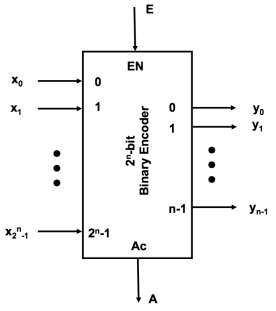
$Q(t+1)=Q(t)K'(t)+Q'(t)J(t)$ — Characteristic Equation

Excitation Equation

0.0.12 Combinational Modules

A binary decoder (DEC) takes an unsigned integer in binary code of n bits and converts into a 1 at the appropriate position among 2^n-1 bits. ENABLE it, or else it gives all zeros. It has NO Active bit output. It gives the minterms of a function, which can then be OR'ed together: $y_0+y_5+y_7$. A binary decoder plus OR gates is a universal set.

A binary encoder (ENC) takes 2^n-1 bits with a single 1 among them and converts it into a unsigned integer in binary code. ENABLE it, or else it gives all zeros. It has an A bit for "active" to distinguish between a 1 for the zero-case and when all the input bits are truly 0.



To create a code converter with a ENC/DEC pair, we take in n encoded bits and feed them into a binary decoder that gives a standardized internal representation with hot bit. We rewrite these connections into a binary decoder, which spits out the data in the other encoding. To figure out the connections, list the standard ordered binary code as input, and figure out what decimal value corresponds to that bit pattern in the first encoding. Then, we link that wire to the corresponding spot among the encoder's inputs.

A simple shifter shifts inputs by one bit to either the left or the right.

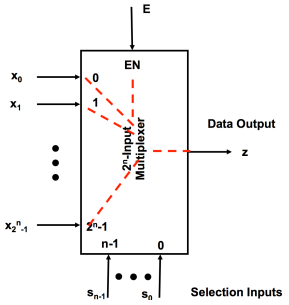
Inputs: $\underline{x} = (x_n, x_{n-1}, \ldots, x_0, x_{-1})$, $x_j \in \{0,1\}$
$d \in \{RIGHT, LEFT\}$
$s \in \{YES, NO\}$
$E \in \{0,1\}$
Outputs: $\underline{y} = (y_{n-1}, \ldots, y_0)$, $y_j \in \{0,1\}$

Function:

$$y_i = \begin{cases} x_{i-1} & \textbf{if } (d = LEFT) \textbf{ and } (s = YES) \textbf{ and } (E = 1) \\ x_{i+1} & \textbf{if } (d = RIGHT) \textbf{ and } (s = YES) \textbf{ and } (E = 1) \\ x_i & \textbf{if } (s = NO) \textbf{ and } (E = 1) \\ 0 & \textbf{if } (E = 0) \end{cases}$$
for $0 \le i \le n-1$.

A p-shifter shifts inputs by p bits to the left or to the right. The shift/no-shift changes to a distance-to-shift input. A barrel shifter shifts inputs in stages, with each shifting or not shifting. By wiring creatively, we shift by different amounts.

A multiplexer (MUX) takes 2^n inputs and selects one of them to send on. It combines multiple data streams for transmission, and sends one a time, as selected by the S bits, where the S bits are interpreted as a unsigned integer. For example, $\underline{S} = 0110 = 6_{10}$, and the input x_6 would be selected.



For implementing a switching function: (1): List out the truth table. (2): If there are more outputs then there are inputs available with the MUX, identify which rows have outputs that are constants (0/1) or that can be described in terms of an input bit. (3): Feed these in as the MUX's data inputs, and the the regular SF's inputs as the selector bits, as they choose a row.

A de-multiplexer (DEMUX) takes a single input and sends it out along one of its 2^n outputs, based on the selecting bits. Its diagram is the reverse of the MUX diagram. Together, a MUX and DEMUX can be used to combine multiple lines into one, with them splitting out output among them and recombining at the end.

0.0.13 Sequential Modules

A register stores state, and can be manually cleared asynchronoously. It updates only upon clock signal.

Inputs: $\underline{x} = (x_{n-1}, \ldots, x_0)$, $x_i \in \{0,1\}$
LD, $CLR \in \{0,1\}$
Outputs: $\underline{z} = (z_{n-1}, \ldots, z_0)$, $z_i \in \{0,1\}$
State: $\underline{s} = (s_{n-1}, \ldots, s_0)$, $s_i \in \{0,1\}$

Function: The state transition and output functions are

$$\underline{s}(t+1) = \begin{cases} \underline{x}(t) & \textbf{if } LD(t) = 1 \textbf{ and } CLR(t) = 0 \\ \underline{s}(t) & \textbf{if } LD(t) = 0 \textbf{ and } CLR(t) = 0 \\ (0 \ldots 0) & \textbf{if } CLR(t) = 1 \end{cases}$$

$$\underline{z}(t) = \underline{s}(t)$$

A shift register is a register that supports shifting operations on current state, inserting an input bit into the spot that frees up. Four types, each sending in/out 1 or all bits at once. SI/SO: $m = n = 1$. SI/PO: $m = 1, n > 1$. PI/SO: $m > 1, n = 1$. PI/PO: $m, n > 1$. SI/SO has one CTRL bit: shift/no-shift, and one right/left insert-bit. SI/PO is the same except for its parallel outputs. PI/SO has two CTRL bits: load/no-load and shift/no-shift. In both the serial-out cases, there's a delay of n cycles until an inserted bit appears, assuming we shift (they're unidirectional) each cycle. The PI/PO case:

Inputs: $\underline{x} = (x_{n-1}, \ldots, x_0), x_i \in \{0,1\}$
$x_l, x_r \in \{0,1\}$
$CTRL \in \{LOAD, LEFT, RIGHT, NONE\}$
State: $\underline{s} = (s_{n-1}, \ldots, s_o), s_i \in \{0,1\}$
Output: $\underline{z} = (z_{n-1}, \ldots, z_0), z_i \in \{0,1\}$

Functions: The state transition and output functions:

$$\underline{s}(t+1) = \begin{cases} \underline{s}(t) & \textbf{if } CTRL = NONE \\ \underline{x}(t) & \textbf{if } CTRL = LOAD \\ (s_{n-2}, \ldots, s_0, x_l) & \textbf{if } CTRL = LEFT \\ (x_r, s_{n-1}, \ldots, s_1) & \textbf{if } CTRL = RIGHT \end{cases}$$

$\underline{z} = \underline{s}$ Always output current state

To recognize patterns, connect a SI/PO to an AND gate. Each of the expected 0 bits from the output are passed through a NOT gate, so that the overall output is not 1 unless the previously seen bits were the full pattern.

A mod-p counter counts up to $p-1$, and gives $TC = 1$ at $s(t) = p-1$ and $x(t) = 1$.

A mod-16 counter has a special LD input that lets us skip numbers while counting. Its native behavior is to increment upon clock and give a TC output of 1 after a cycle is done. $CNT = 1$ means "update" and $LD = 1$ means "use input bits to update". However, if $CNT = 1$ and $LD = 1$, the count-signal is ignored, and $s(t+1) = \underline{I}$.

Have the desired base-state be the input $\underline{I}$, for TC and LD, have an AND gate take in x and the bits of state that would represent the final state before wrap-around, e.g., $s0, s1, s3$ for $1011_2(11_{10})$. Thus, TC is 1 when we wrap-around, and LD is 1 at the same time, and so we'll load our base state, which might be 0001, which gets us a 1-to-11 counter. If we want modulo-k, then wraparound at $k-1$. For 1-to-k, wrap-around at k itself. May need combinational networks for CNT, LD, inputs, and parallel outputs, with external input feeding into all of them.