Computation Structures

Basics of Information Worksheet

Concept Inventory:

- Measuring information content; entropy
- Two's complement; modular arithmetic
- Variable-length encodings; Huffman's algorithm
- Hamming distance, error detection, error correction

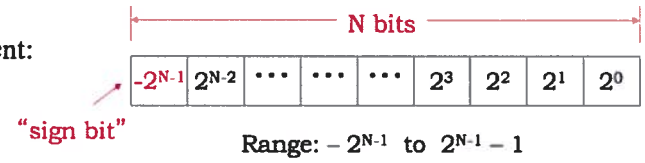
Notes:

Measuring information: $I(x_i) = \log_2(1/p_i)$ bits

N equally-probable choices down to M choices: $\log_2(N/M)$ bits

Entropy: $H(X) = E(I(X)) = \sum_i p_i \log_2(1/p_i)$

N-bit 2's complement:



Variable-length encoding:

Symbols with smallest p_i have longest encodings, symbols with largest p_i have shortest encodings.

Huffman's algorithm:

- Build binary decoding tree bottom-up starting with symbols that have smallest p_i.
- Each step: combine the two symbols or subtrees with smallest p_i into new subtree.

Hamming distance:

- HD = # of bit positions that differ between to codewords
- need to know min Hamming distance (HD_{min}) considering all pairs of codewords
- # of errors detected = $HD_{min} - 1$
- # of errors corrected = $\left\lfloor \frac{HD_{min}-1}{2} \right\rfloor$

Name: _____

1. Information Content and Entropy

A. You are given an unknown 3-bit binary number. You are then told that the binary representation contains exactly two 1's. How much information have you been given?

3 bits ⇒ 8 choices

exactly two 1's ⇒ 011, 101, 110 ⇒ 3 choices

$\log_2(8/3)$ bits

B. You are then given the **additional** information that the number is also odd. How much additional information have you been given?

start ⇒ 3 choices

odd ⇒ 011, 101 ⇒ 2 choices

$\log_2(3/2)$ bits

C. A random variable X represents the outcome of flipping an unfair coin, where p(HEADS) = 0.6. Please give the value for the entropy H(X). You may express your answer as a numeric expression (i.e., you don't have to actually do the arithmetic).

$H(X) = (0.6)\log_2\left(\frac{1}{0.6}\right) + (0.4)\log_2\left(\frac{1}{0.4}\right) = 0.97$ bits

D. A single decimal digit is chosen at random and you're told that its value is 0 mod 3. How much information have you learned about the digit?

decimal digit ⇒ 10 choices

0 mod 3 ⇒ 0, 3, 6, 9 ⇒ 4 choices

$\log_2(10/4)$ bits

E. X is an unknown 8-bit binary number. You are given another 8-bit binary number, 10101100, and told that the Hamming distance between X and 10101100 is one. How many bits of information about X have you been given? You can give a formula if you wish.

8 bits ⇒ 256 choices

HD=1 from 10101100 ⇒ 8 choices

$\log_2\left(\frac{256}{8}\right) = 5$ bits.

F. We wish to transmit messages comprised of the four symbols shown below with their associated probabilities and 5-bit fixed-length encoding.

Symbol	p(symbol)	encoding
α	0.5	00000
β	0.125	11100
γ	0.25	11011
δ	0.125	10111

An unknown symbol is received and you are told it's not δ. How much information have you received?

$p(\text{not } \delta) = 7/8 \Rightarrow \log_2(1/p) = \log_2(8/7)$ bits

Name: _____

G. When transmitting a message comprised of these four symbols with the probabilities as given above, what is the expected amount information received when you are told the next symbol in the message?

Entropy = $(0.5)\log_2(\frac{1}{.5}) + (0.25)\log_2(\frac{1}{.25}) + 2(.125)\log_2(\frac{1}{.125})$

= 1.75 bits.

H. You are given an unknown 5-bit binary number. You are then told that the first and last bits are the same. How much information have you been given?

5 bits ⇒ 32 choices

1st and last bits the same ⇒ 0xxx0, 1xxx1 } 16 choices

$\log_2(32/16) = 1$ bit.

I. I've randomly selected a letter from the alphabet and tell you that my selection is neither "X", "Y", nor "Z". How much information have I given you about my letter?

random letter ⇒ 26 choices

not X, Y, Z ⇒ 23 choices

$\log_2(26/23)$ bits

J. I make up a random 4-bit **two's complement** number by flipping a fair coin to determine each bit. You're trying to guess the number. If I tell you that the number is positive (> 0), how many bits of information have I given you? Be precise; you may answer by a formula or a number.

4 bits ⇒ 16 choices

positive ⇒ 7 choices
>0

$\log_2(16/7)$ bits

Name: _____

2. Two's Complement

A. What is the 6-bit two's complement representation of the decimal number -21?

$21_{10} = 010101$

$-21_{10} = \sim 21_{10} + 1 = 101010 + 1 = \boxed{101011}$

B. What is the hexadecimal representation for decimal -51 encoded as an 8-bit two's complement number?

$51 = 00110011$

$-51 = \sim(51) + 1 = (11001100) + 1 = 11001101$ (C D)

$\boxed{0xCD}$

C. The hexadecimal representation for an 8-bit two's complement number is 0xD6. What is its decimal representation?

$0xD6 = 11010110 = -128 + 64 + 16 + 4 + 2 = \boxed{-42}$

D. Since the start of official pitching statistics in 1988, the highest number of pitches in a single game has been 172. Assuming that remains the upper bound on pitch count, how many bits would we need to record the pitch count for each game as a *two's complement* binary number?

need 9 bits : $+2^8 \mid 2^7 \mid 2^6 \mid 2^5 \mid 2^4 \mid 2^3 \mid 2^2 \mid 2^1 \mid 2^0$

largest positive # $= 2^8 - 1 = 255$

E. Can the value of the sum of two 2's complement numbers 0xB3 + 0x47 be represented using an 8-bit 2's complement representation? If so, what is the sum in hex? If not, write NO.

```
 0xB3    10110011
+0x47   +01000111
------  ---------
 0xFA    11111010
          F    A
```

F. Can the value of the sum of two 2's complement numbers 0xB3 + 0xB1 be represented using an 8-bit 2's complement representation? If so, what is the sum in hex? If not, write NO.

```
 0xB3    10110011
+0xB1   +10110001
------  ---------
       (1)01100100
```

overflow!

$\boxed{No}$

Name: _____

G. Please compute the value of the expression 0xBB – 8 using 8-bit two's complement arithmetic and give the result in decimal (base 10).

$-8 = N(8)+1 = N(00001000)+1$
$= 11110111+1 = 11111000$

$$\begin{array}{r} 11111000 \\ +\ 10111011 \\ \hline 1\ 10110011 \end{array} = -128+32+16+2+1 = \boxed{-77}$$

H. What is the smallest (most negative) integer that can be represented as an 8-bit two's-complement integer? Give your answer as a decimal integer.

most negative = 1000 0000 = $\boxed{-128}$

I. The following operations are performed on an 8-bit adder. Give the 8-bit sum produced for each, in **hexadecimal**.

$$\begin{array}{r} F0 \\ +34 \\ \hline \end{array} \quad \begin{array}{r} 11110000 \\ 00110100 \\ \hline 1\ 00100100 \end{array}$$
2 4

$$\begin{array}{r} F0 \\ +80 \\ \hline \end{array} \quad \begin{array}{r} 11110000 \\ 10000000 \\ \hline 1\ 01110000 \end{array}$$
overflow!

0xF0 + 0x34 = 0x 24

0xF0 + 0x80 = 0x 70

J. Using a 5-bit two's complement representation, what is the range of integers that can be represented with a single 5-bit quantity?

$\boxed{-2^4 \mid 2^3 \mid 2^2 \mid 2^1 \mid 2^0}$

range = -2^4 to 2^4-1
-16 to 15

K. Consider the following subtraction problem where the operands are 5-bit two's complement numbers. Compute the result and give the answer as a decimal (base 10) number.

$$\begin{array}{r} 10101 \\ -\ 00011 \\ \hline 10010 \end{array} = -16+2 = \boxed{-14}$$

Name: _____

3. Variable-length Encodings

A. Given a variable X that can take on one of four values A, B, C, or D with the following probabilities.

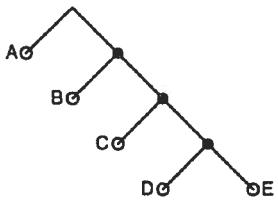
Symbol	Probability
A	0.5
B	0.3
C	0.1
D	0.1

If you encoded this variable using a Huffman encoding, how many bits would be in the encoding of each of the symbols?

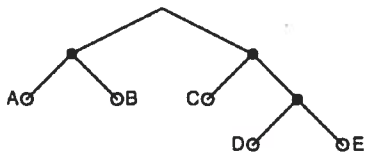
A = 1 bit
B = 2 bits
C, D = 3 bits each

For each of the probability distributions for symbols A-E, select the Huffman encoding tree or trees that could result from running Huffman's algorithm on those probability distributions.

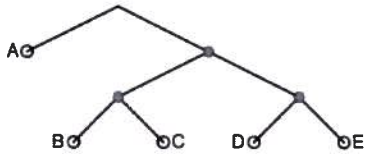
Tree #1



Tree #2



Tree #3



B. p(A) = 0.3, p(B) = 0.3, p(C) = 0.2, p(D) = 0.1, p(E) = 0.1. Tree(s): 2

C. p(A) = 0.6, p(B) = 0.1, p(C) = 0.1, p(D) = 0.1, p(E) = 0.1. Tree(s): 3

D. p(A) = 0.5, p(B) = 0.15, p(C) = 0.15, p(D) = 0.1, p(E) = 0.1. Tree(s): 3

E. p(A) = 0.5, p(B) = 0.2, p(C) = 0.15, p(D) = 0.05, p(E) = 0.1. Tree(s): 1

Name: _____

Baseball loves statistics! There are many different types of pitches that a pitcher can throw – the table below shows the probability for each type of pitch during 2014.

Type of pitch	*Probability*
Fastball	0.34
Change-up	0.14
Curveball	0.08
Slider	0.28
Other	0.16

F. How much information have you received when learning that particular pitch was NOT a fastball? You ~~can express your~~ answer as a formula if you wish.

p(not fastball) = 1 - .34 = .66

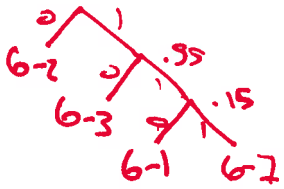
$\log_2\left(\frac{1}{.66}\right)$

G. To save on storage costs, Major League Baseball would like to use an optimal variable-length code to record, one at a time, the type of each pitch (i.e., to record one of the 5 types shown in the table above). Use Huffman's algorithm to derive such a code and list the resulting binary encodings below.

Fastball 00
Slider 01
other 10
Curveball 110
Changeup 111

H. The table below shows the 2012-13 enrollments in the various EECS majors. To save a bit of space in their database the department would like to use a variable-length Huffman code to encode a student's choice of major. For each of the four majors, please give the encoding the department should use.

Major	**Count**	***p***	***p* log$_2$(1/*p*)**	encoding
6-1	74	0.09	0.30	110
6-2	387	0.44	0.52	0
6-3	360	0.41	0.53	10
6-7	54	0.06	0.25	111
Total	*875*	*1.00*	*1.60*	



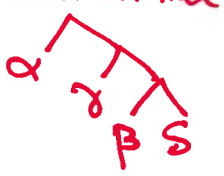
I. We wish to transmit messages comprised of the four symbols shown below with their associated probabilities and 5-bit fixed-length encoding

Symbol	p(symbol)	encoding
α	0.5	00000
β	0.125	11100
γ	0.25	11011
δ	0.125	10111

Huffman's algorithm is used to construct a variable-length code for the four symbols for transmitting a single symbol at a time. The resulting encoding could be

(1) α: 00, β: 01, γ: 10, δ: 10
(2) α: 00, β: 01, γ: 100, δ: 101
(3) α: 1, β: 01, γ: 000, δ: 001
~~(4)~~ α: 0, β: 110, γ: 01, δ: 111
(5) none of the above

Huffman tree



Name: _____

NerdLink is a new web-based startup that aims to keep MIT EECS students in touch with their parents. NerdLink streamlines parental communication by providing each student with an online choice of one of the five messages, then automatically fills in boilerplate and emails the parent a long and charming version of the message. The five messages, and their relative probabilities, are listed below:

Message #	*Message to parents*	*p(Message)*
M1	Send money!	60%
M2	I love this course called 6.004	8%
M3	I'm changing my major to Poetry	2%
M4	I'm getting a 5.0 this term!	1%
M5	Nothing much is new… (none of the above)	29%

NerdLink's initial implementation conveyed each message using a fixed-length code.

J. What is the average number of bits needed to convey a message, using a fixed-length code?

5 choices ⇒ need 3 bits to encode

K. Given the probability distribution of the messages, what is the *actual* amount of information conveyed by message M5? Your answer may be a formula.

$p(M5) = .29$ $\quad I(M5) = \log_2(1/.29)$

L. To enable error correction, the fixed-length code for a given message is sent *five* times. Using the five copies of the received message, in the worst case how many bit errors can be corrected at the receiver?

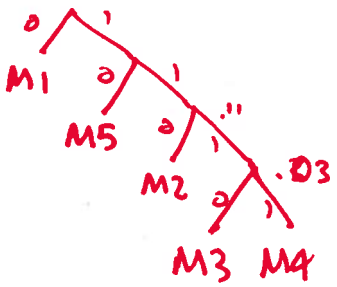
min HD of original fixed-length code = 1 bit

min HD of replication 5 times = 5 bits

correction = $\lfloor \frac{HD-1}{2} \rfloor$ = 2 bits

NerdLink, wanting to economize on communication costs, has hired you as a consultant to design a Huffman code for sending the messages.

M. Give the **number of bits** sent by your Huffman code for each message (M1 though M5), and the average number of bits transmitted per message using your code (a formula will be fine).



	# bits
M1: 0	1
M2: 101	3
M3: 1110	4
M4: 1111	4
M5: 10	2

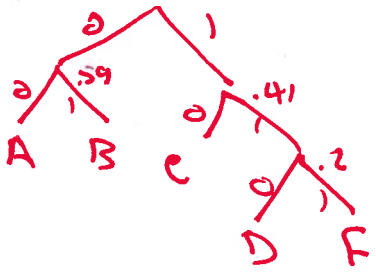
avg:
.6(1) + .3(3) +
.02(4) + .01(4) + .29(10)
= 1.54 bits

Name: _____

The Registrar's office would like to encode the letter grades (A, B, C, D, F) from a large GIR with 1000 students. They plan to encode each grade separately using a variable-length code. An analysis of previous terms has produced the following table of grade probabilities. In case it's useful, a thoughtful former 6.004 student has augmented the table by computing $p\log_2(1/p)$ for each grade.

Grade	p	$p\log_2(1/p)$
A	0.24	0.49
B	0.35	0.53
C	0.21	0.47
D	0.13	0.38
F	0.07	0.27
Totals	1.00	2.14

N. Use Huffman's algorithm to construct an optimal variable-length encoding.



A: 00
B: 01
C: 10
D: 110
F: 111

O. Two 6.004 students have proposed competing variable-length codes. Alice says that encoding 1000 grades using her code will, on the average, produce messages of 2200 bits. Bob says that encoding 1000 grades using his code will, on the average, produce messages of 1950 bits. Which of the following is your best response when the Registrar asks your opinion?

(A) Choose Bob's: it has the shorter average length
(B) Choose Alice's: more bits means more information is transmitted
(C) Choose Bob's: Bob's average message length is less than the information entropy
(D) Choose Alice's: Bob's average message length is less than the information entropy
(E) Choose neither: a fixed-length code will have lower average message size

Best Choice (A through E): D

Bob's code is ambiguous since it sends fewer bits than entropy lower bound

Name: _____

4. Error Detection and Correction

A. A message about the suit of a card is sent using the encoding shown at the right. Using this encoding, how many bit errors can be detected? How many bit errors can be corrected?

```
Club:     000
Diamond:  011
Heart:    101
Spade:    110
```

minHD=2 detected = HD-1 = 1

corrected = $\lfloor \frac{HD-1}{2} \rfloor = 0$

B. A message about the suit of a card is sent using the encoding shown at the right. Give an example of a 5-bit received message with an uncorrectable single-bit error or write NONE if all single-bit errors can be corrected.

```
Heart:    00000
Diamond:  11001
Spade:    10111
Club:     01011
```

- Find 2 codewords with HD=2:
- introduce 1-bit error

11011 or 01001

11001 diamond 01011 club

C. The MIT baseball coach likes to record the umpire's call for each pitch (one of "strike", "ball" or "other"). Worried that bit errors might corrupt the records archive, he proposes using the following 5-bit binary encoding for each of the three possible entries:

Strike	11111
Ball	01101
Other	00000

] min HD = 2

Using this encoding what is the largest number of bit errors that be *detected* when examining a particular record? The largest number of bit errors that can be *corrected*?

detected = HD-1 = 1

corrected = $\lfloor \frac{HD-1}{2} \rfloor = 0$

D. When transmitting the information about EECS majors over a noisy communication link, the department has chosen to use the 7-bit encoding shown on the right in the hopes that they'll be able to correct multiple-bit errors during transmission. Using this code, how many bit errors in a message about a single major will the receiver be able to correct?

6-1: 0101010
6-2: 1001100
6-3: 0110001
6-7: 1010010

] min HD = 4

correct = $\lfloor \frac{HD-1}{2} \rfloor$ = 1 bit error correction

Name: _____

E. We wish to transmit messages comprised of the four symbols shown below with their associated probabilities and 5-bit fixed-length encoding

Symbol	p(symbol)	encoding
α	0.5	00000
β	0.125	11100
γ	0.25	11011
δ	0.125	10111

min HD = 2

If we transmit messages using the 5-bit fixed-length encoding shown above, will it be possible to perform single-bit error detection and correction at the receiver?

correction : $\lfloor \frac{HD-1}{2} \rfloor = 0 \Rightarrow$ NO!

F. What is the Hamming distance between the encodings for A and B? Using an encoding scheme with this Hamming distance, how many bits of error can be detected? How many bits of error can be corrected?

A: 010010
B: 110101

HD = 4

detected = HD − 1 = 3

corrected = $\lfloor \frac{HD-1}{2} \rfloor = 1$

Name: _____

An internet Sudoku gaming site transmits messages containing nine data bits and seven parity bits, arranged in a rectangle as follows:

D_{00}	D_{01}	D_{02}	P_{0x}
D_{10}	D_{11}	D_{12}	P_{1x}
D_{20}	D_{21}	D_{22}	P_{2x}
P_{x0}	P_{x1}	P_{x2}	P_{xx}

if this changes → row bit changes

col bit changes → overall bit changes

Each D_{ij} in the above diagram indicates a data bit, equally likely to be a 0 or 1. Each P_{ix} and P_{xj} is an odd parity bit chosen to make the total number of 1s in the i^{th} row or j^{th} column, respectively, odd. P_{xx} is an odd parity bit chosen to make the total number of 1s in the entire transmission odd. Thus in an error-free transmission, the total number of 1s in 4-bit columns 0 thru 2 and 4-bit rows 0 thru 2, as well as in the entire 16-bit transmission, is odd.

Note that each 9-bit data word determines a unique 16-bit *valid codeword* to be transmitted.

G. What is the minimum Hamming distance between valid codewords? [Hint: flipping one bit of the data word changes how many bits of the codeword?]

4

flipped data bit ⇒ flip row & col parity
∴ 3 bits have changed so far ⇒ flip overall parity

Each of the following represents a transmission received, with at most a single-bit error. For each message, circle the bit, if any, that was changed due to a transmission error.

H.

1	0	1	1
0	1	1	1
1	1	0	1
1	1	1	1

no errors

I.

1	0	1	1
1	1	0	1
0	1	1	1
1	0	1	1

P_{x1} must be wrong

1-bit error

J.

0	1	0	1
0	0	1	0
1	1	0	1
1	1	0	0

D_{00} must be wrong

1-bit error

K.

0	1	0	0
1	0	1	1
0	1	1	1
0	1	1	1

P_{xx} must be wrong

1-bit error

check row, col, overall parity
↳ # of 1's should be odd

6.004 Computation Structures
Spring 2017

Computation Structures

The Digital Abstraction Worksheet

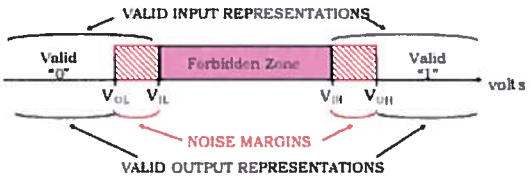
Signaling:

- Analog: each processing step accumulates noise
- Digital: each processing step restores output to a valid digital level

Needed: Noise Margins!

Proposed fix: separate specifications for inputs and outputs

- digital output: "0" ≤ V_{OL}, "1" ≥ V_{OH}
- digital input: "0" ≤ V_{IL}, "1" ≥ V_{IH}
- $V_{OL} < V_{IL} < V_{IH} < V_{OH}$



A combinational device accepts marginal inputs and provides unquestionable outputs (to leave room for noise).

A Digital Processing Element

A combinational device is a circuit element that has

Static discipline:

- one or more digital *inputs*
- one or more digital *outputs*
- a *functional specification* that details the value of each output for every possible combination of valid input values
- a *timing specification* consisting (at minimum) of an upper bound t_{PD} on the required time for the device to compute the specified output values from an arbitrary set of stable, valid input values

A Combinational Digital System

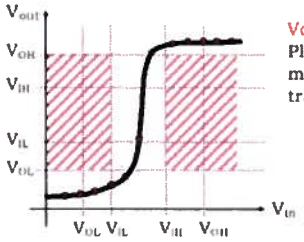
A set of interconnected elements is a combinational device if

- each circuit element is combinational
- every input is connected to exactly one output or to some vast supply of constant 0's and 1's
- the circuit contains no directed cycles

Why is this true?

A Buffer

A simple *combinational device*: 0 → 0 1 → 1



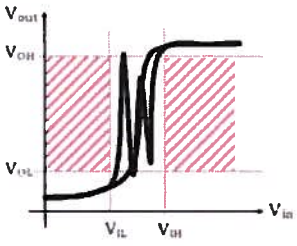
Voltage Transfer Characteristic (VTC): Plot of V_{out} vs. V_{in} where each measurement is taken after any transients have died out.

Note. VTC does not tell you anything about how fast a device is — it measures static behavior not dynamic behavior

Static Discipline requires that the VTC avoid the shaded regions (aka "*forbidden zones*" which correspond to *valid* inputs but *invalid* outputs.

A Buffer

Voltage Transfer Characteristic



1) Note the VTC can do anything when $V_{IL} < V_{IN} < V_{IH}$.

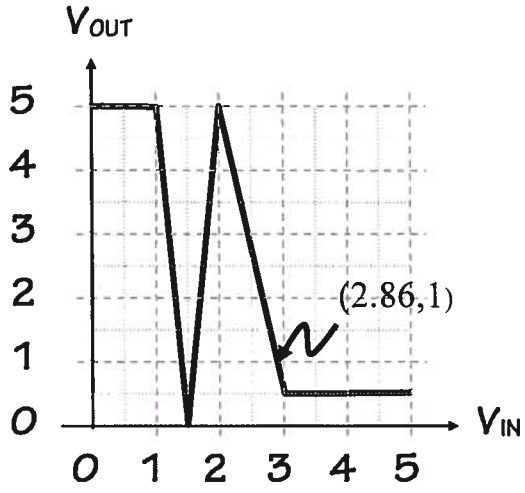
2) Note that the center white region is taller than it is wide (V_{OH}-V_{OL} > V_{IH}-V_{IL}). Net result: combinational devices must have GAIN > 1 and be NONLINEAR.

Name: _____

Problem 1.

Ms. Anna Logge, founder at a local MIT start-up, has developed a device to be used as an inverter. Anna is considering the choice of parameters by which her logic family will represent logic values and needs your help.

The voltage transfer curve of a proposed inverter for a new logic family is shown to the right (spare copies of this diagram can be found below).



Several possible schemes for mapping logic values to voltages are being considered, as summarized in the incomplete table below. **Recall that Noise Immunity (last row) is defined as the lesser of the two noise margins.**

Complete the table by filling in missing entries. Choose each value you enter so as to maximize the noise margins of the corresponding scheme. **If the numbers in a scheme can't be completed such that the device functions as an inverter with positive noise margins, put an X in the entries for that column.**

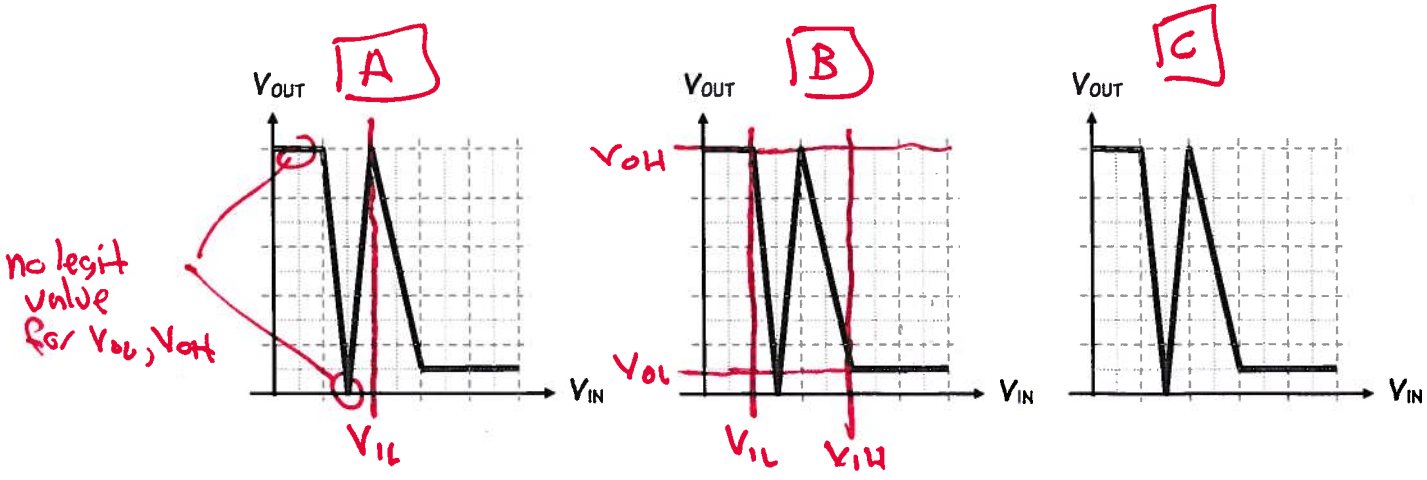
(complete table – 10 entries)

LNI's Possible Logic Mappings:

	Scheme A	Scheme B	Scheme C
V_{OL}	X	0.5	1
V_{IL}	2	1	0.5
V_{IH}	X	3	X
V_{OH}	X	5	X
Noise Immunity	X	0.5	X

oops: $V_{OL} > V_{IL}$!

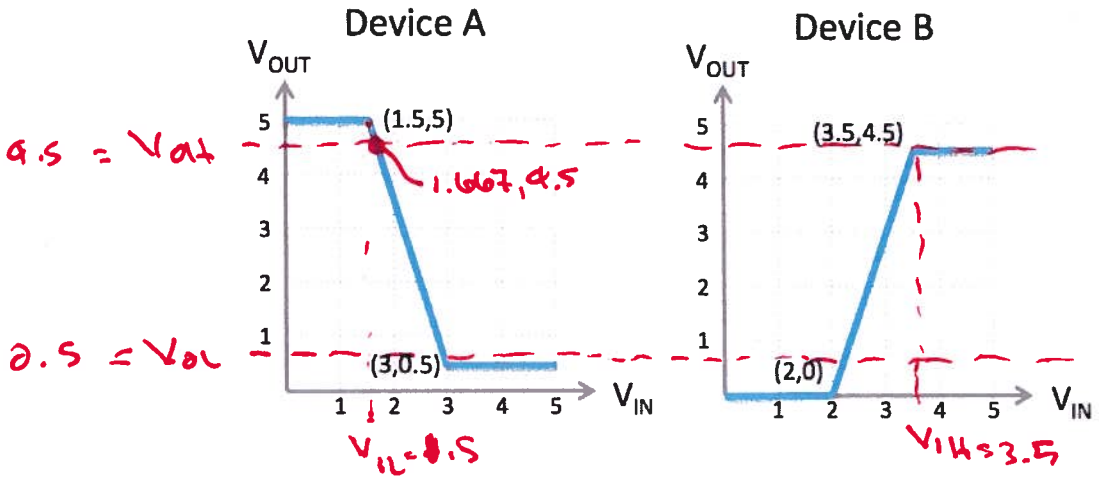
smaller of two noise margins



Name: _____

Problem 2.

The following are voltage transfer characteristics of single-input, single-output devices to be used in a new logic family:



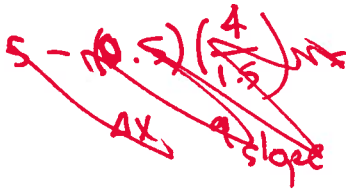
Your job is to choose a single set of signaling thresholds V_{OL}, V_{IL}, V_{OH}, and V_{IH} to be used with both devices to give the best noise margins you can. Recall that the VTC can touch the edge of the forbidden regions but not pass through those regions. Fill in your answers below, together with the resulting noise margins. You'll get partial credit for anything that works with nonzero noise margins; for full credit, maximize the noise immunity (i.e., the smaller of the two noise margins).

V_{OL} = 0.5 V_{IL} = 1.5 * V_{IH} = 3.5 V_{OH} = 4.5

Low Noise Margin = 1 **High Noise Margin** = 1

* also V_{IL} = 1.667
works if V_{OH} = 4.5
⇒ low noise margin = 1.667 − 0.5
= 1.167

⇒ but noise immunity unchanged!



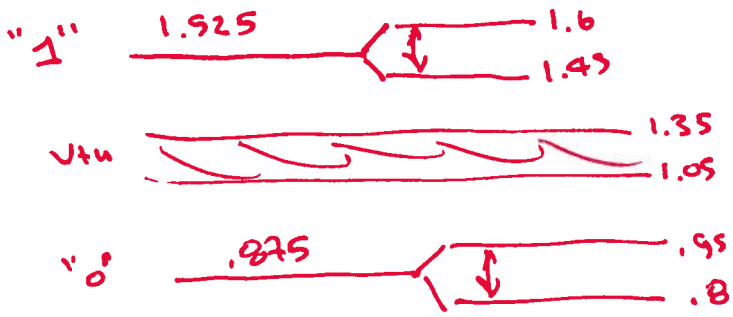
Name: _____

Problem 3.

Massachusetts Instruments manufactures the XYZZY family of combinational logic devices, which have the following specifications:

- When signaling a "0" on a device output, XYZZY devices are guaranteed to produce an output voltage of 0.875 ± 0.075 volts.
- When signaling a "1" on a device output, XYZZY devices are guaranteed produce an output voltage of 1.525 ± 0.075 volts.
- XYZZY device inputs compare the incoming voltage against a logic threshold V_{TH}. Input voltages less than or equal to $V_{TH} - 0.05V$ are guaranteed to be interpreted as "0". Input voltages greater than or equal to $V_{TH} + 0.05V$ are guaranteed to be interpreted as "1". V_{TH} is an internal voltage in the range 1.2 ± .1 volts.

(A) Please give the appropriate values for the four digital signaling thresholds:



V_{OL} (volts): 0.95

V_{IL} (volts): 1.05

V_{IH} (volts): 1.35

V_{OH} (volts): 1.45

(B) The noise immunity of a signaling specification is the minimum of the two noise margins. What is the noise immunity of your signaling specification?

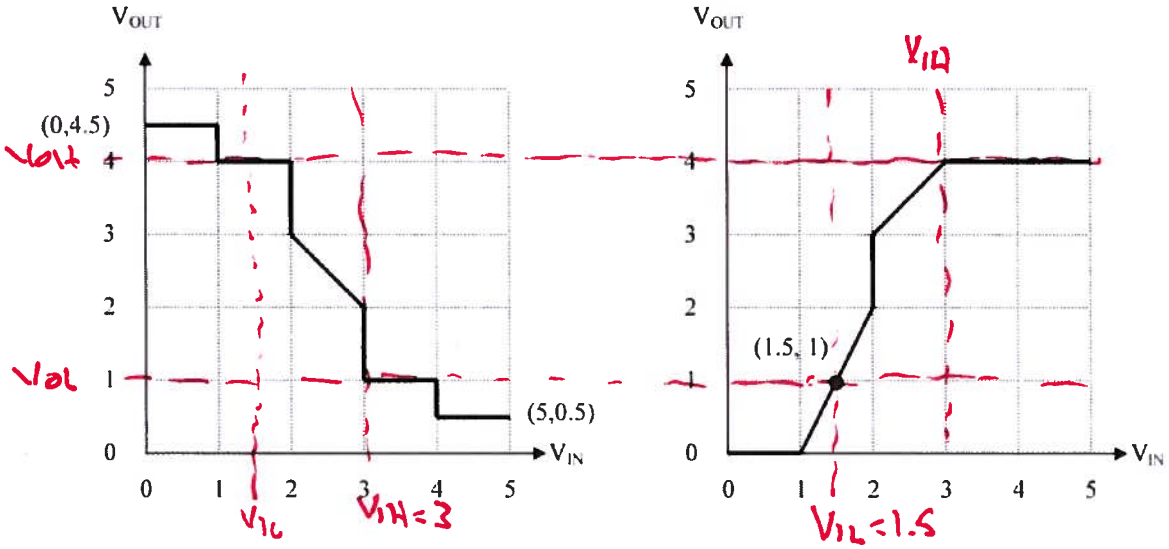
Noise immunity (volts): 0.1

$\min(V_{IL}-V_{OL}, V_{OH}-V_{IH})$

Name: _____

Problem 4.

The following are voltage transfer characteristics of devices to be used in a new logic family as an inverter and buffer, respectively:

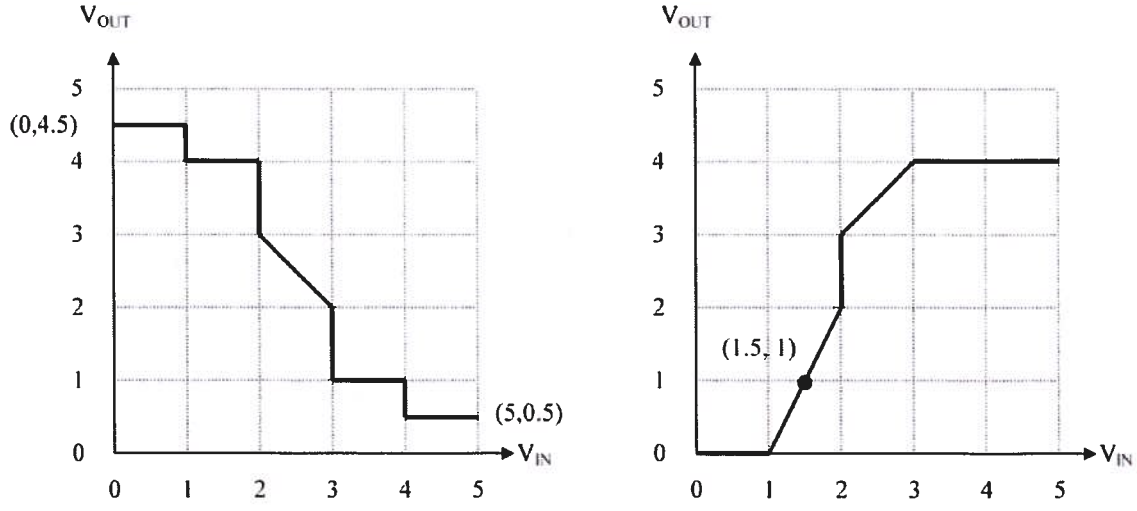


Your job is to choose a single set of signaling thresholds V_{OL}, V_{IL}, V_{OH}, and V_{IH} to be used with both devices to give the best noise margins you can. Recall that the VTC can touch the edge of the forbidden regions but not pass through those regions. Fill in your answers below, together with the resulting noise margins. You'll get partial credit for anything that works with nonzero noise margins; for full credit, maximize each of the noise margins.

V_{OL} = 1 V_{IL} = 1.5 V_{IH} = 3 V_{OH} = 4

Low Noise Margin = 0.5 **High Noise Margin** = 1

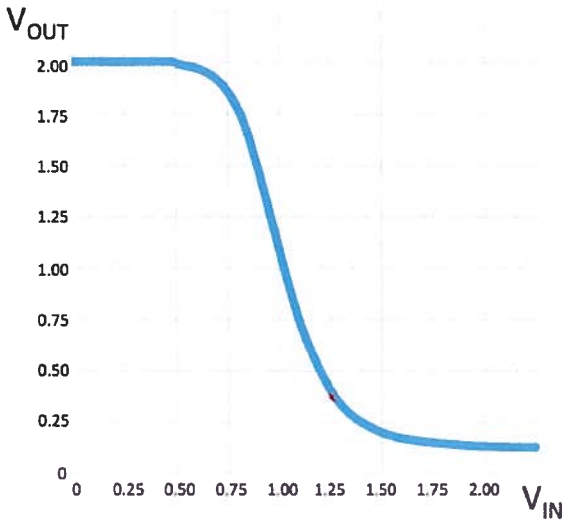
Scratch copy of the VTC diagrams:



Name: _____

Problem 5.

The voltage transfer curve for an NMOS inverter is shown to the right.



The manufacturer decided to crowd-source the digital signaling specifications for the NMOS inverter and has received some suggestions for V_{OL}, V_{IL}, V_{IH}, and V_{OH}, presented below in tabular form. For each suggested specification determine if the NMOS inverter above would be a legitimate combinational device obeying the static discipline with non-zero positive noise margins. If it is a legitimate combinational device, give the noise immunity of the inverter (the smaller of the low and high noise margins) when operating under that specification. If the inverter wouldn't be a legitimate combinational device, please write NOT LEGIT in the rightmost column.

Fill in rightmost column for each suggested specification.

lowest output 0.125

Suggestion	V_{OL}	V_{IL}	V_{IH}	V_{OH}	*Noise immunity, or NOT LEGIT*
#1	0.00	0.50	1.50	2.00	not legit
#2	0.25	0.75	1.25	1.75	not legit
#3	0.50	0.75	1.25	1.50	0.25
#4	0.75	0.50	1.75	1.50	not legit

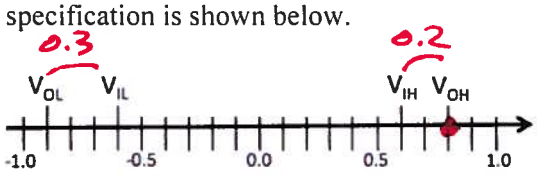
negative noise margins

when IN=1.25, out > V_{OL} (0.25)

Problem 6.

A new family of logic devices uses signaling voltages in the range −1V to +1V. One proposed assignment of our voltage specification is shown below.

0.1/div



(A) The *noise immunity* of a signaling specification is the smaller of the two noise margins. What is the noise immunity for the signaling scheme proposed above?

0.2 = min($V_{IL}-V_{OH}$, $V_{OH}-V_{IH}$)

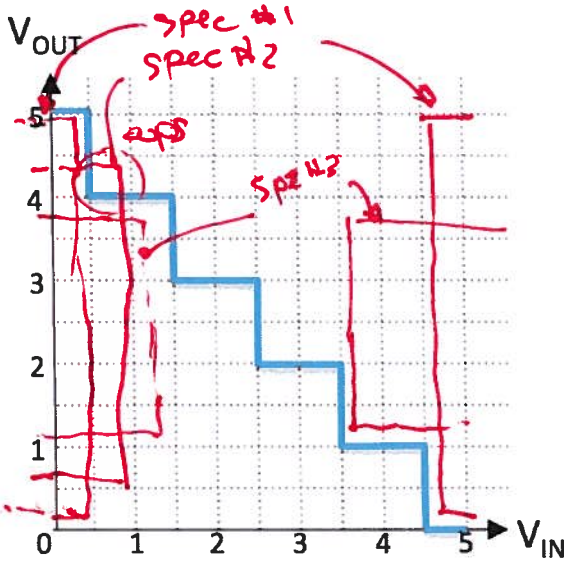
(B) The output voltage of an inverter is measured to be 0.9V in the steady state. The inverter is a combinational device obeying the signaling specification shown above. What is the best characterization of the steady-state input voltage V_{IN} of the inverter when the measurement was made?

V_{OUT} = 0.9 ≥ V_{OH} ⇒ $V_{IN} < V_{IH} = 0.6$!

Name: _____

Problem 7.

Ivan Idea, a resident of Chelyabinsk who's been watching the 6.004 videos on YouTube, was inspired to attach electrodes to opposite ends of a meteor fragment that came through his roof and produce a voltage transfer curve (VTC) of the resulting device, which is shown below.



Amazingly all the "corner points" of the VTC fall on the 0.5V grid.

Ivan is hoping he can sell his device as the world's only extraterrestrial combinational inverter and has provided the table below suggesting possible voltage thresholds to achieve 0.3V noise margins. He's happy to report that for any input voltage, the output voltage becomes stable within 1ns of the application of a new, stable input voltage. For each proposed specification please circle "YES" if the device obeys the static discipline and "NO" if it does not.

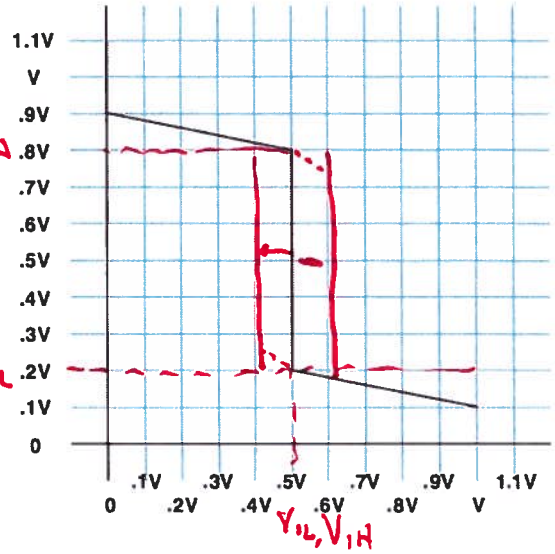
Circle YES or NO for each proposal below

	V_{OL}	V_{IL}	V_{IH}	V_{OH}	Obeys static discipline?
Specification #1	0.1	0.4	4.6	4.9	**YES** **NO**
Specification #2	0.6	0.9	4.1	4.4	**YES** **NO**
Specification #3	1.1	1.4	3.6	3.9	**YES** **NO**

Name: _____

Problem 8.

Consider a device whose voltage transfer characteristic is specified as a function of the supply voltage V as follows



Note that the device has a sharp (infinite gain) threshold at 0.5V.

(A) Using this device as an inverter, if V_{OL} is chosen to be 0.2V, what value for V_{IH} will maximize the high noise margin?

$V_{IH} = 0.5$

(lowest possible V_{IH})

(B) What is the maximum noise immunity that can be realized using this device as an inverter, with an appropriately chosen signaling specification?

$\min(V_{IL} - V_{OL}, V_{OH} - V_{IH})$

$= \min(0.5 - \epsilon - 0.2, .8 - (.5 + \epsilon))$

$= \min(0.3 - \epsilon, 0.3 - \epsilon) = 0.3 - \epsilon$

(C) Suppose manufacturing variations for the above device now allow the threshold voltage to vary between 0.4V and 0.6V, rather than always being 0.5V exactly. If the signaling specifications were adjusted to accommodate this variation, what would be the maximum possible noise immunity?

now $V_{IL} = 0.4$

$V_{IH} = 0.6$

$\min(0.9 - 0.2, 0.8 - 0.6) = 0.2$

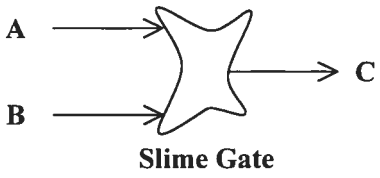
could be lower by $\frac{.1}{5} = 0.02$

could be higher by $\frac{.1}{5} = 0.02$

0.22

Name: _____

Problem 9.



Slime Gate

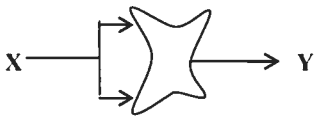
Organic Logic, Inc., is a Cambridge startup that has developed an interesting device built using unidentified organic sludge from the depths of the Charles river; they would like to use it to perform logic functions. Their device, termed a Slime Gate, has two inputs A and B, and one output C (in addition to power and ground connections):

With a 3 volt power supply, they have noted that Slime Gates reliably behave as follows:

- The output C is always in the range 0 volts < C < 3 volts.
- When either (or both) A or B has been less than 1 volt for a nanosecond or more, the voltage at C is greater than 2.5 volts.
- When A and B have both been more than 2 volts for at least a nanosecond, C carries a voltage of less than 0.5 volts.

Aside from the above constraints, the voltage at C is generally unpredictable; it varies widely between individual Slime Gate devices.

As an O.L.I. consultant, you have proposed the following circuit as an inverter in the evolving family of Slime Gate logic:



Inverter

(A) (2 points) Give logic representation parameters yielding 0.5-volt noise margins and for which the above diagram depicts a valid inverter.

V_{OL}: 0.5 ; V_{IL}: 1 ; V_{IH}: 2 ; V_{OH}: 2.5

(B) (2 points) Give appropriate specifications for propagation and contamination delays for this inverter.

t_{PD}: 1 ns; t_{CD}: N/A. ns

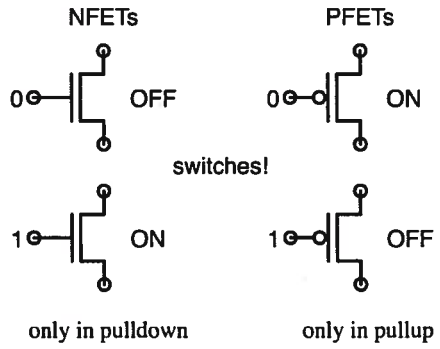
Computation Structures

CMOS Technology Worksheet

Concept Inventory:

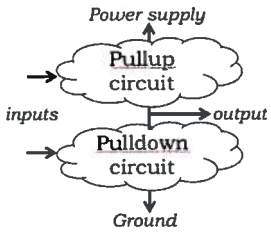
- PFET, NFET: voltage controlled switches
- CMOS composition rules: complementary pullup and pulldown
- CMOS gates are naturally inverting
- t_{PD} and t_{CD} timing specifications
- Lenient gates

Notes:



We want *complementary* pullup and pulldown logic, i.e., the pulldown should be "on" when the pullup is "off" and vice versa.

pullup	pulldown	F(inputs)
on	off	driven "1"
off	on	driven "0"
on	on	driven "X"
off	off	no connection



CMOS gates are naturally inverting:

- Rising input (0 to 1): NFETs turn on, PFETs turn off; if output changes, it falls (1 to 0)
- Falling input (1 to 0): NFETs turn off, PFETs turn on; if output changes, it rises (0 to 1)

Timing:

- t_{PD} (propagation delay): how long after inputs are stable and valid until outputs are stable and valid = max over all paths from input to output (sum of component t_{PD} along path)
  - t_{PD} specification is an upper bound on all measured propagation delays
- t_{CD} (contamination delay): how long output stays valid after inputs go invalid = min over all paths from input to output (sum of component t_{CD} along path)
  - t_{CD} specification is a lower bound on all measured contamination delays

Lenient gate:

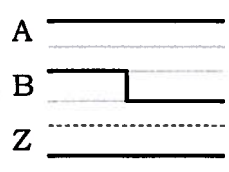
- If a subset of a lenient gate's inputs is suffice to guarantee an specific output value (i.e., the values of the other inputs don't matter in this case), then the output will remain valid and stable by transitions on the irrelevant inputs.
- CMOS gates are naturally lenient

NOR:

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

Lenient NOR:

A	B	Z
0	0	1
X	1	0
1	X	0



Problem 1.

(A) Which of the above CMOS pulldown circuits would implement F if the corresponding complementary pullup circuit was also provided? For each pulldown, select Yes if it is a valid pulldown for F, and No if it is not a valid pulldown for F.

F is Ø when
- A=1 and
- D=1 and
- either B=1 or C=1

PD1 (Yes/No): YES

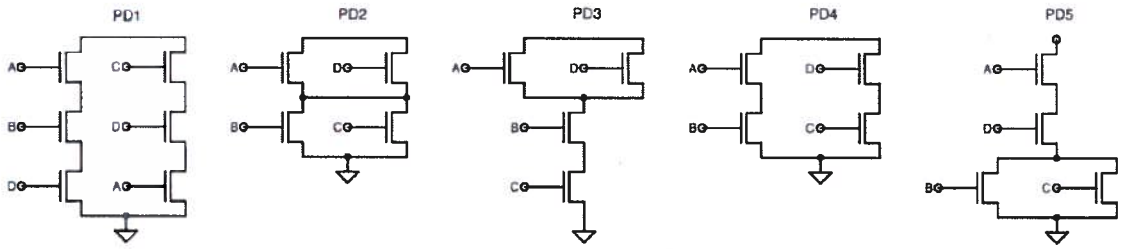
PD2 (Yes/No): NO

PD3 (Yes/No): NO

PD4 (Yes/No): NO

PD5 (Yes/No): YES

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0



(B) Are all the implementations you selected for part (A) lenient?

single CMOS gate is always lenient. **All lenient (Yes/No):** YES

Problem 2.

(A) A single CMOS gate, consisting of an output node connected to a single PFET-based pullup circuit and a single NFET-based pulldown circuit (as described in lecture) computes F(A, B, C, D). It is observed that F(1, 0, 1, 0) = 1. What can you say about the following values?

nfet off, pfet on ⇒ no change

(circle one) F(0, 0, 1, 0) = : 0 ... 1 ... (can't say)

(circle one) F(1, 1, 1, 0) = : 0 ... 1 ... (can't say)

all nfets on

(circle one) F(1, 1, 1, 1) = : 0 ... 1 ... (can't say)

(B) The Boolean function F(A,B,C) can be implemented using a *single* CMOS gate operating as a combinational device that obeys the static discipline. It's known that F(1,1,0) = 1 and F(0,1,1) = 0. What can be determined about the value of F in the following cases? Please circle one of "0", "1" or "Can't tell".

if F(1,1,0)=1
then F(?,?,0)=1

(circle one) F(1,0,0) = 0 ... (1) ... Can't tell

(circle one) F(1,0,1) = 0 ... 1 ... (Can't tell)

(circle one) F(1,1,1) = (0) ... 1 ... Can't tell

(C) A single CMOS gate, consisting of an output node connected to a single pullup circuit containing one or more PFETs and a single pulldown circuit containing one or more NFETs (as described in lecture), computes $F(A,B)$. F has the property that for all A, $F(A,0) = \overline{F(A,1)}$. What can you say about the value of $F(1,0)$?

$F(1,0) = \overline{F(1,1)} = \overline{0} = 1$
↑ all NFETs on

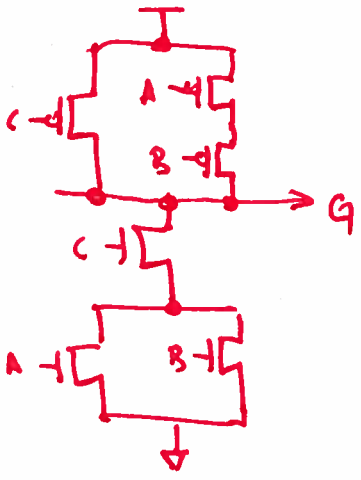
(circle one): F(1,0) = (1) ... 0 ... can't tell

Problem 3.

For each of the functions F and G, if the function can be implemented using a **single CMOS gate**, please draw the corresponding single CMOS gate. If it cannot be implemented using a single CMOS gate, then write NONE. **For full credit use a minimum number of FETs.**

A	B	C	F	G
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0

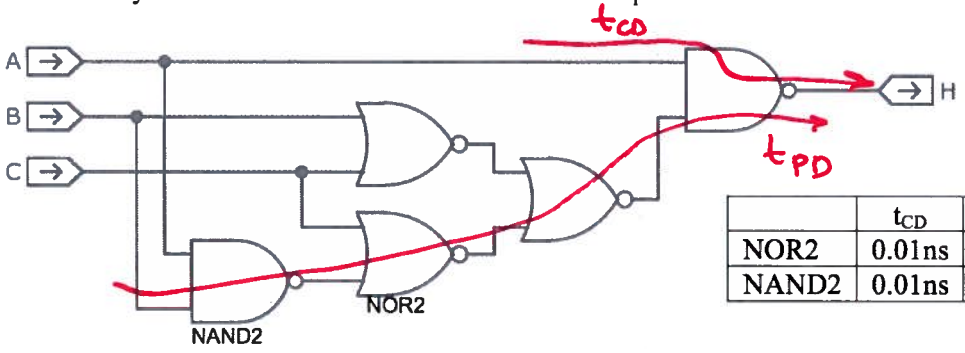
G is Φ when
- C is 1 and
- either A=1 or B=1

Draw CMOS implementation of F(A,B,C) below or write NONE if F cannot be implemented as single CMOS gate.	**Draw CMOS implementation of G(A,B,C) below or write NONE if G cannot be implemented as single CMOS gate.**
F(1,1,1) = 1 ⇒ not a CMOS gate since all NFETs on should produce a zero.	

Problem 4.

Consider the Boolean function that has the truth table shown to the right; a possible implementation as a combinational circuit is shown in the schematic below. You may assume that the NOR2 and NAND2 components are combinational.

A	B	C	H
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



	t_{CD}	t_{PD}
NOR2	0.01ns	0.05ns
NAND2	0.01ns	0.03ns

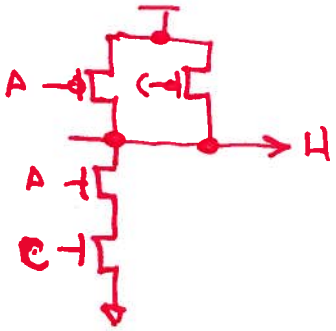
(A) Using the timing specifications shown above for NOR2 and NAND2, compute the contamination and propagation delay for the implementation of H shown above.

timing for H (ns): t_{CD}= 0.01 t_{PD}= 0.16

(B) Can H be implemented as a single CMOS gate (only PFETs in the pullup circuit, only NFETs in the pulldown circuit)? If so draw the MOSFET schematic for H to the right, otherwise write "NO".

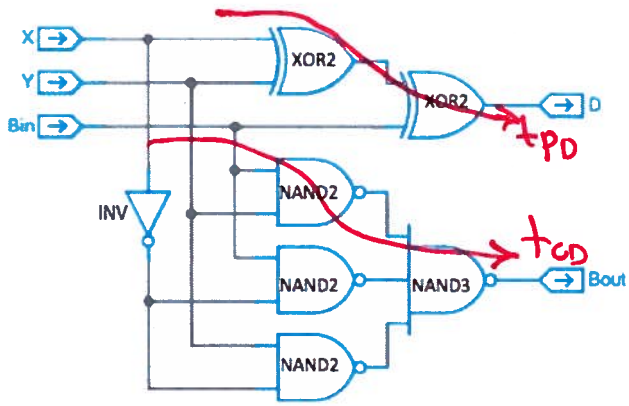
Draw schematic or write "NO"

H=0 when A=1 and C=1



Problem 5.

A gate-level schematic is shown below. Using the t_{CD} and t_{PD} information for the gate components shown in the table below, compute t_{CD} and t_{PD} for the circuit.



Compute timing specs:

t_{CD} = 0.5 ns

t_{PD} = 5 ns

Gate	t_{CD}	t_{PD}
INV	0.1ns	1.0ns
NAND2	0.2ns	1.5ns
NAND3	0.3ns	1.8ns
XOR2	0.6ns	2.5ns

Problem 6.

A minority gate has three inputs (call them A, B, C) and one output (call it Y). The output will be 0 if two or more of the inputs are 1, and 1 if two or more of the inputs are 0.

In the space below, draw the *pulldown* circuit for a single CMOS gate that implements the minority function, using the minimum number of NFETs. You needn't draw the *pullup* circuit.

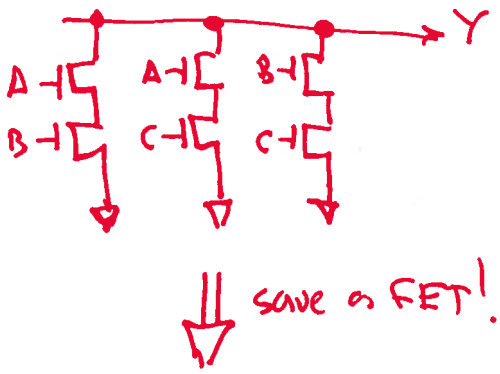
If you're convinced that the function cannot be implemented as a single CMOS gate, give a brief, convincing explanation.

Can it be implemented as single CMOS gate? Circle one: (YES) can't tell NO

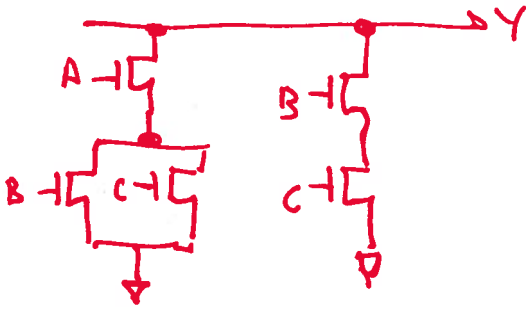
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Y is 0 when

- A=1 and B=1, or
- A=1 and C=1 or
- B=1 and C=1

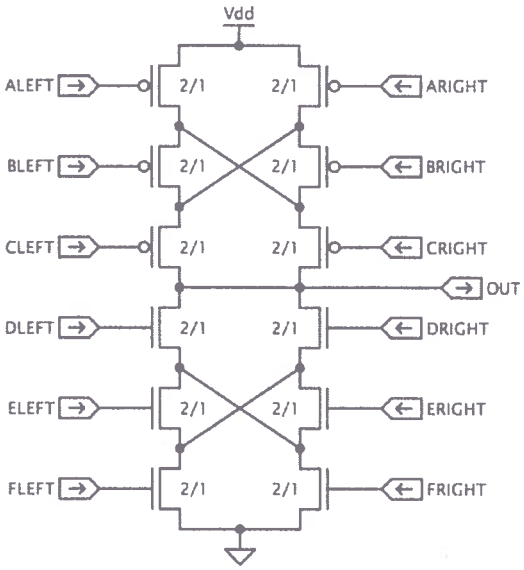


save a FET!



Problem 7.

In his bid for the Lemelson Prize, Ben Bitdiddle has invented the "flexible gate," a single CMOS gate that implements different functions depending how its inputs are wired up. The FlexGate® (see figure at right) uses 6 PFETs in its pullup circuit and 6 NFETs in its pulldown circuit..



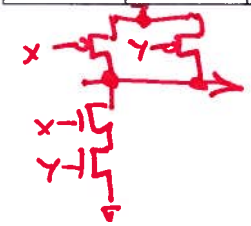
Each of the FlexGate's twelve inputs can connected to an input signal (X, Y, …), GND (logical "0") or VDD (logical "1"). To show off its versatility, Ben has asked you to show how to hook up the inputs so the FlexGate computes several different functions whose Boolean equations are given below. Associated with each equation is a table with 12 entries; in each cell of the table please write an input name, GND or VDD as appropriate. Note that there may be several possible implementations for each of the three functions – any correct answer will be acceptable. Hint: there should be an entry in each cell, i.e., a connection should be specified each input!

If the desired function cannot be implemented, please draw a big "X" through the table.

Note : other answers possible! **Fill in tables below or mark with "X"**

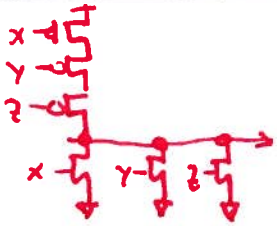
$$OUT = \overline{X \cdot Y}$$

input	LEFT	RIGHT
A	GND	GND
B	VDD	VDD
C	X	Y
D	X	GND
E	GND	GND
F	GND	Y



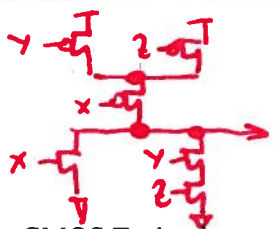
$$OUT = \overline{X + Y + Z}$$

input	LEFT	RIGHT
A	Z	VDD
B	Y	VDD
C	X	VDD
D	X	VDD
E	GND	Y
F	Z	VDD



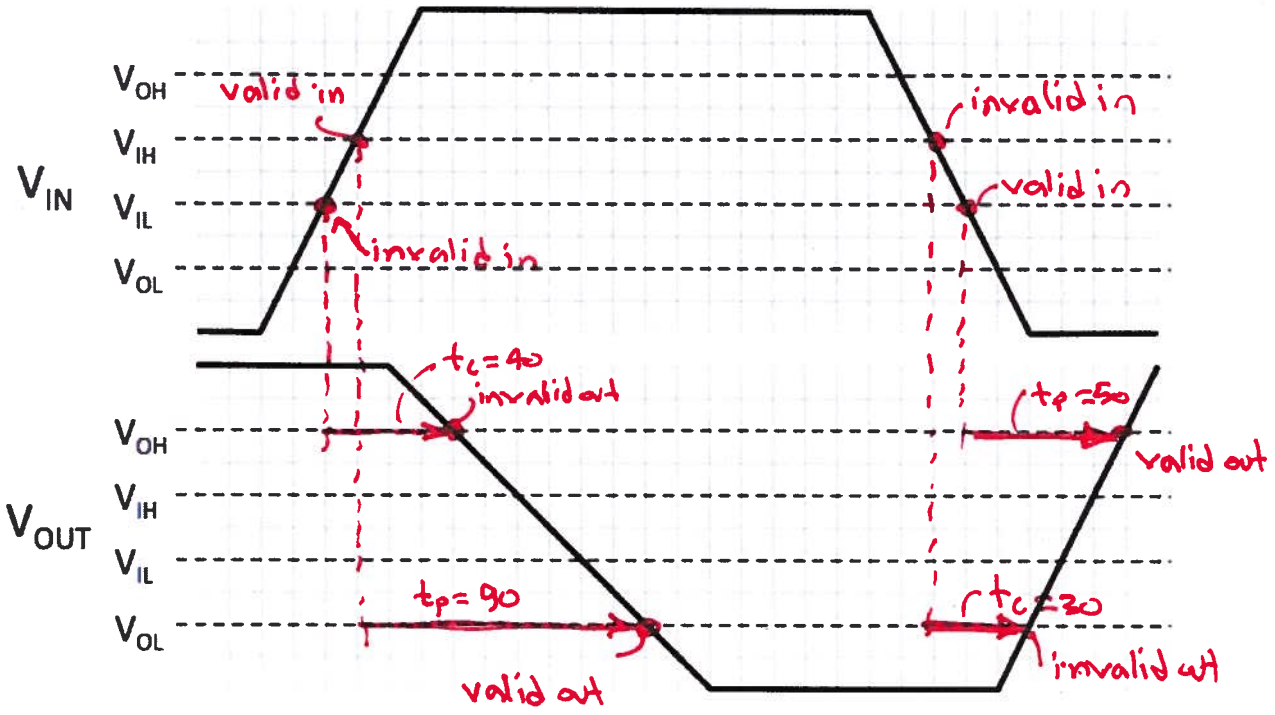
$$OUT = \overline{X + Y \cdot Z}$$

input	LEFT	RIGHT
A	Y	Z
B	GND	VDD
C	X	VDD
D	X	Y
E	GND	Z
F	GND	VDD



Problem 8.

The response of a combinational gate to a test input waveform is shown below. Each horizontal division of the plot represents 10 ps.



(A) Based on the figure below, what is an appropriate choice for the contamination delay of the gate?

$$\min(t_c) = \min(40, 30) = 30\text{ps} = t_{CD}$$

t_{CD} is lower bound on all t_c

(B) Based on the figure below, what is an appropriate choice for the propagation delay of the gate?

$$\max(t_p) = \max(90, 50) = 90\text{ps} = t_{PD}$$

t_{PD} is upper bound on all t_p.

6.004 Computation Structures
Spring 2017

Computation Structures

Combinational Logic Worksheet

Concept Inventory:

- Truth tables ↔ sum-of-products equations
- implementation using NOT/AND/OR
- Demorgan's Law, implementation using NAND/NOR
- Simplification, truth tables w/ don't cares
- Karnaugh maps
- Implementation using MUXes and ROMs

Here's a Design Approach

Truth Table

C	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

1. Write out our functional spec as a truth table
2. Write down a Boolean expression with terms covering each '1' in the output:

$$Y = \bar{C}B\bar{A} + \bar{C}BA + C\bar{B}\bar{A} + CBA$$

3. We'll show how to build a circuit using this equation in the next two slides.

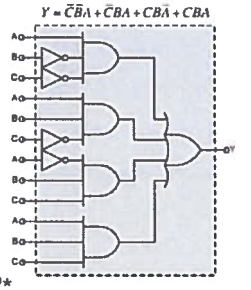
This approach will always give us Boolean expressions in a particular form: SUM-OF-PRODUCTS

Straightforward Synthesis

We can implement SUM-OF-PRODUCTS with just three levels of logic:

1. Inverters
2. ANDs
3. OR

Propagation delay -- No more than 3 gate delays?*



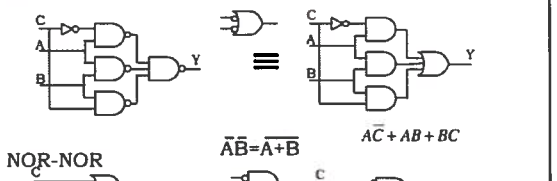
*assuming gates with an arbitrary number of inputs, which, as we'll see, isn't a good assumption!

CMOS Sum-of-products Implementation

NAND-NAND

$\overline{AB} = \bar{A} + \bar{B}$

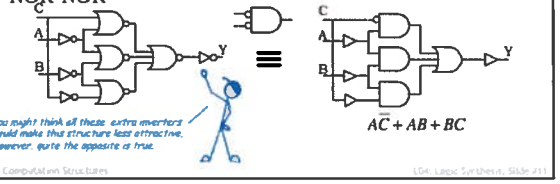
"Pushing Bubbles"



$A\bar{C} + AB + BC$

NOR-NOR

$\overline{A+B} = \bar{A}\bar{B}$



$A\bar{C} + AB + BC$

You might think all these extra inverters would make this structure less attractive. However, quite the opposite is true.

Finding Implicants

An implicant

- is a rectangular region of the K-map where the function has the value 1 (i.e., a region that will need to be described by one or more product terms in the sum-of-products)
- has a width and length that must be a power of 2: 1, 2, 4
- can overlap other implicants
- is a prime implicant if it is not completely contained in any other implicant.

C\AB	00	01	11	10
0	0	0	1	1
1	1	0	0	0

$A\bar{C}$

$\bar{A}BC$

C\AB	00	01	11	10
0	1	0	0	1
1	1	1	0	1

$\bar{A}\bar{C}$

$\bar{B}$

- can be uniquely identified by a single product term. The larger the implicant, the smaller the product term.

Write Down Equations

Picking just enough prime implicants to cover all the 1's in the KMap, combine equations to form minimal sum-of-products.

C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$Y = A\bar{C} + BC$

We're done!

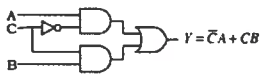
\AB C|	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

$Y = D + B\bar{C} + A\bar{C} + \bar{B}C$

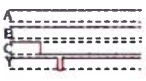
Minimal SOP is not necessarily unique!

Prime Implicants, Glitches & Leniency

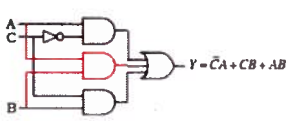
This circuit produces a glitch on Y when A=1, B=1, C: 1→0



C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0



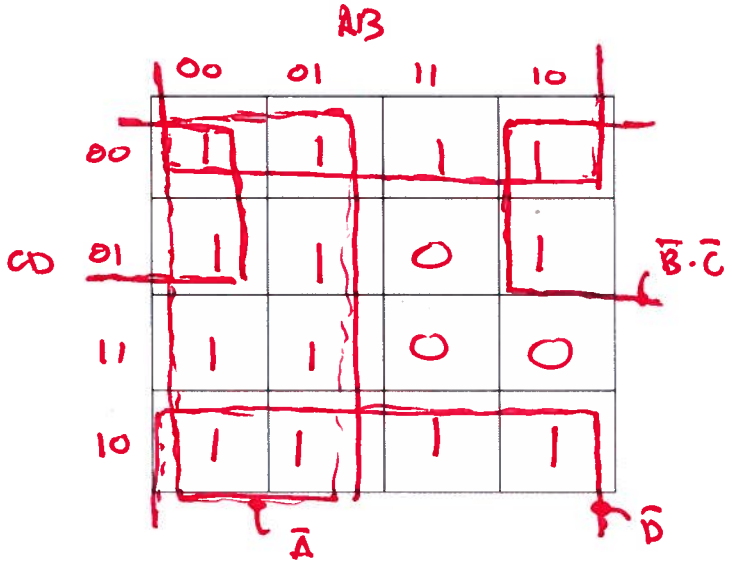
To make the circuit lenient, include product terms for ALL prime implicants.



Problem 1.

Given a function F defined by the truth table to the right, provide a minimal sum-of-products expression for F. Hint: Use a Karnaugh Map.

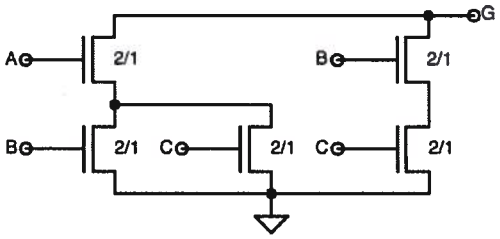
A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0



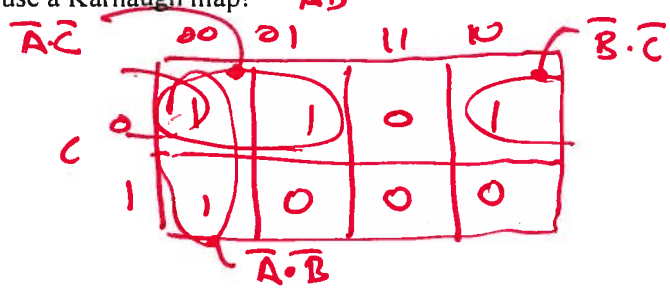
Minimal Sum-of-products Expression for F: $\bar{A} + \bar{D} + \bar{B}\cdot\bar{C}$

Problem 2.

(A)A correctly-formed CMOS gate implementing G(A,B,C) uses the pulldown circuit shown on the right. Please give a *minimal sum-of-products expression* for G(A,B,C). Hint: use a Karnaugh map!



FILL IN 0's USING PULLDOWNS; REST ARE 1's



Minimal sum-of-products expression for G(A,B,C): $\bar{A}\cdot\bar{B} + \bar{A}\cdot\bar{C} + \bar{B}\cdot\bar{C}$

(B) Is the function G(A,B,C) from part (B) universal? In other words, can we implement any Boolean function using combinational circuits built only from G gates and the Boolean constants 0 and 1?

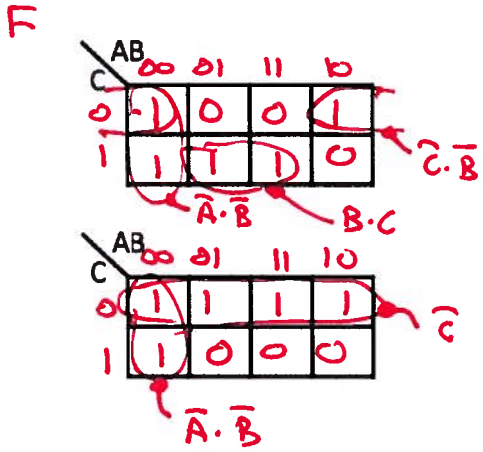
YES

G(0,B,C) == NAND(B,C)

Problem 3.

Consider the following truth table which defines two functions F and G of three input variables (A, B, and C).

A	B	C	F	G
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	1
1	1	1	1	0



Give the **minimal sum of products** (minimal SOP) logic equation for each of the two functions. Then determine if the minimum sum of products expression would result in a **lenient** implementation of the function. If it does, then enter "**SAME**" for the lenient SOP expression. **If not**, specify what sum of products expression would result in a lenient implementation. Hint: Use Karnaugh maps above to determine the minimal sum of products.

Minimal sum of products F(A,B,C) = $\bar{A}\cdot\bar{B} + B\cdot C + \bar{B}\cdot\bar{C}$

Does minimal SOP for F result in a lenient circuit (circle one)? Yes (No)

consider A=0, C=0, B: 0→1.

If "No", give lenient SOP expression for F(A,B,C) = $\bar{A}\cdot\bar{B} + B\cdot C + \bar{B}\cdot\bar{C} + \bar{A}C$

extra implicant!

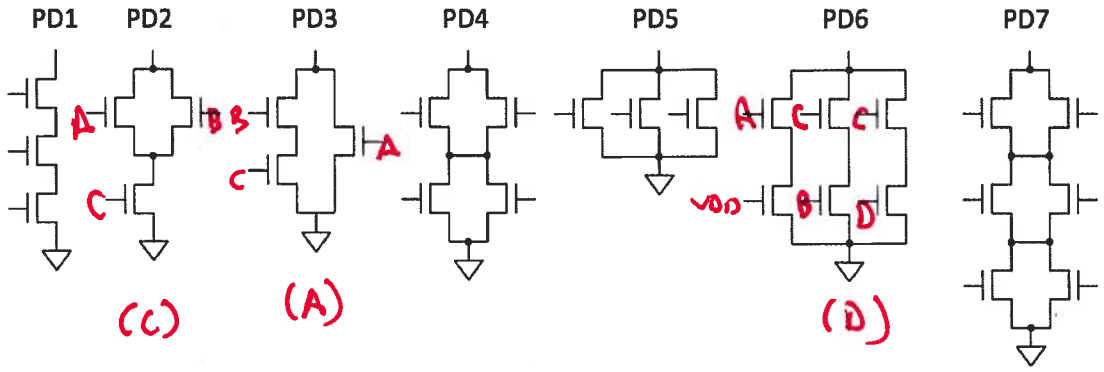
Minimal sum of products G(A,B,C) = $\bar{A}\cdot\bar{B} + \bar{C}$

Does minimal SOP for G result in a lenient circuit (circle one)? (Yes) No

If "No", give lenient SOP expression for G(A,B,C) = —

Problem 4.

You are trying to select pulldowns for several 3- and 4-input CMOS gate designs. The Pulldowns-R-Us website offers seven different pulldowns, given names PD1 through PD7, diagrammed below:



The web site explains that the customer can choose which inputs or constants (GND, VDD) are connected to each NFET, allowing their pulldowns to be used in various ways to build gates with various numbers of inputs. Since Pulldowns-R-Us charges by transistor, you are interested in selecting pulldowns using the minimum number of transistors for each of the 3-input gates you are designing.

For each of the following 3- and 4-input Boolean functions, choose the appropriate pulldown design, i.e., the one which, properly connected, implements that gate's pulldown using the *minimum number* of transistors. This may require applying Demorgan's Laws or minimizing the logic equation first. If none of the above pulldowns meets this goal, write "NONE".

(A) $F(A,B,C) = \overline{A + (B \cdot C)}$ **Choice or NONE:** PD3

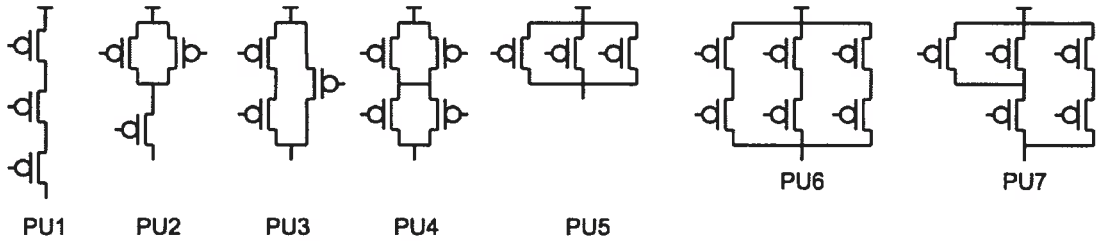
(B) $F(A,B,C) = A + \overline{B \cdot C}$ non inverting! **Choice or NONE:** NONE

(C) $F(A,B,C) = (\overline{A} \cdot \overline{B}) + \overline{C} = \overline{(A+B)} + \overline{C}$
$= \overline{(A+B) \cdot C}$ **Choice or NONE:** PD2

(D) $F(A,B,C,D) = \overline{A + C \cdot (B + D)}$ **Choice or NONE:** PD6

Problem 5.

You are trying to select pullups for several 3-input CMOS gate designs. The Pullups Galore web site offers seven different pullups, given names PU1 through PU7, diagrammed below:



The web site explains that the customer can choose which inputs are connected to each PFET, allowing their pullups to be used in various ways to build gates with various numbers of inputs. Since Pullups Galore charges by transistor, you are interested in selecting pullups using the minimum number of transistors for each of the 3-input gates you are designing.

For each of the following 3-input Boolean functions, choose the appropriate pullup design, i.e., the one which, properly connected, implements that gate's pullup using the *minimum number* of transistors. This may require minimizing the logic equation first. If none of the above pullups meets this goal, write "NONE".

(A) $F(A,B,C) = \overline{A} + \overline{B} + \overline{C}$ **Choice or NONE:** PU5

(B) $F(A,B,C) = \overline{A} + \overline{B \cdot C}$ same as (A)! **Choice or NONE:** PU5

(C) $F(A,B,C) = \overline{A + B \cdot C}$ $= \overline{A} \cdot (\overline{B} + \overline{C})$ **Choice or NONE:** PU2

(D) $F(A,B,C) = A + \overline{B \cdot C}$ not inverting! **Choice or NONE:** NONE

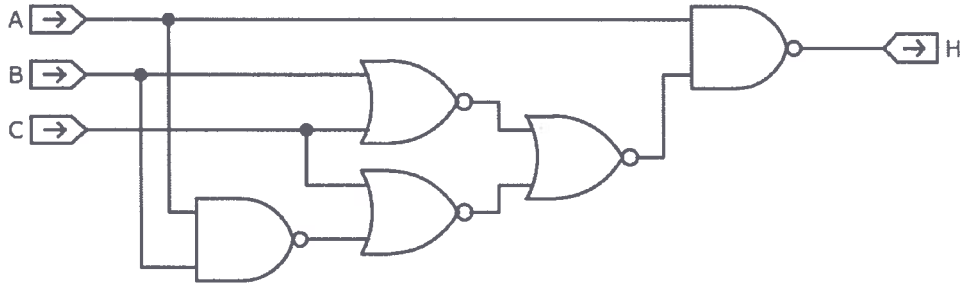
(E) $F(A,B,C) = \overline{(A+B)} + \overline{(B+C)} + \overline{(A+C)}$
$= \overline{A} \cdot \overline{B} + \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{C} = \overline{A} \cdot (\overline{B} + \overline{C}) + \overline{B} \cdot \overline{C}$ **Choice or NONE:** PU7 *

(F) $F(A,B,C) = \overline{(A+C) \cdot B}$
$= \overline{A} \cdot \overline{C} + \overline{B}$ **Choice or NONE:** PU3

* PU6 would also work, but uses more PFETs

Problem 6.

Consider the Boolean function $H(A,B,C) = \overline{A} + \overline{B}\cdot\overline{C} + A\cdot B\cdot\overline{C}$. Its truth table is shown to the right and a possible implementation is shown in the schematic below.



A	B	C	H
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(A) Give a minimal sum-of-products expression for H. A couple of scratch 3-input Karnaugh map templates are provided for your convenience.

minimal sum-of-products expression for H: $\overline{A} + \overline{C}$

AB

H	00	01	11	10
0	1	1	1	1
1	1	1	0	0

C

$\overline{C}$

$\overline{A}$

	00	01	11	10
0				
1				

(B) What is the largest number of product terms possible in a minimal sum-of-products expression for a 3-input, 1-output Boolean function?

Largest number of product terms possible: 4

	00	01	11	10
0				
1				

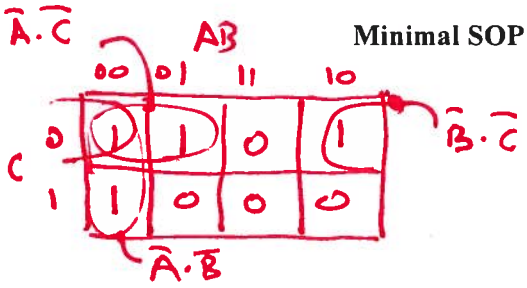
Think of a checker board pattern of 1's. ⇒ 4 product terms.

if you add another product term, k-map can be used to simplify

Problem 7.

A minority gate has three inputs (call them A, B, C) and one output (call it Y). The output will be 0 if two or more of the inputs are 1, and 1 if two or more of the inputs are 0.

(A) Give a *minimal* **sum-of-products** Boolean expression for the minority gates output Y, in terms of its three inputs A, B, and C.



Minimal SOP expression: Y = $\bar{A}\cdot\bar{B} + \bar{A}\cdot\bar{C} + \bar{B}\cdot\bar{C}$

(B) Is a minority gate *universal*, in the sense that using only minority gates (along with constants 0 and 1) its possible to implement arbitrary combinational logic functions?

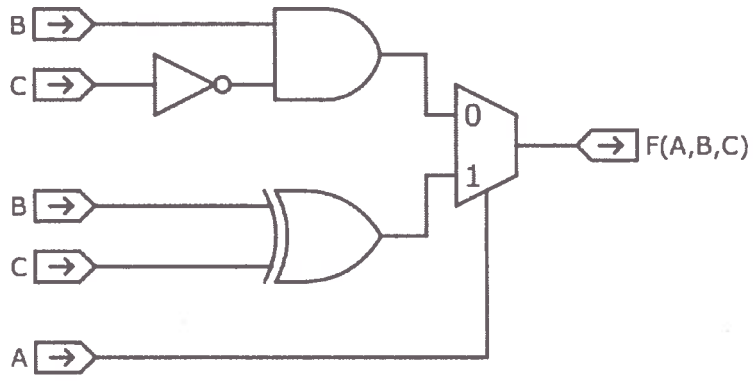
Universal? Circle one: **YES** **can't tell** **NO**

minority(A,B,1) == NOR(A,B)

↑ known to be universal

Problem 8.

A 6.004 intern at Intel has designed the combinational circuit shown below. His boss can't figure out what it does and has asked for your help.



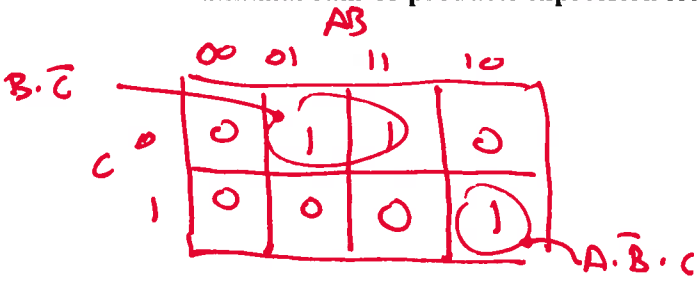
A	*B*	*C*	*F(A,B,C)*
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	0	0

(A) Please fill in the truth table for F(A,B,C) above.

Fill in truth table above

(B) Express F(A,B,C) in minimal sum-of-products form. Hint: use a Karnaugh map!

minimal sum-of-products expression for F(A,B,C) = $A \cdot \bar{B} \cdot C + B \cdot \bar{C}$

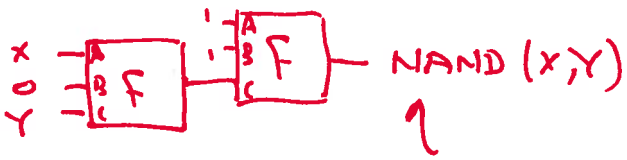


(C) The boss isn't quite sure what it means but he knows his engineers are always impressed if he asks "is the circuit universal?" Is it? Circle YES or NO.

F(A,B,C) universal? YES .. NO

$F(1,1,C) = \bar{C}$

$F(A,0,C) = A \cdot C$

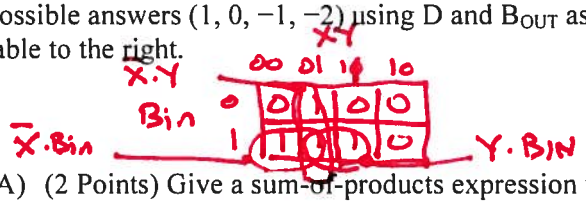


known to be universal

Problem 9.

The full subtractor (FS) implements a one-column binary subtraction of two bits (X and Y) producing their difference (D), accepting a borrow-in (B_{IN}) from the previous column and producing a borrow-out (B_{OUT}) for the next column. Numerically FS computes $X-Y-B_{IN}$ and encodes the possible answers (1, 0, −1, −2) using D and B_{OUT} as shown in the truth table to the right.

X	Y	B_{IN}	D	B_{OUT}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



(A) (2 Points) Give a sum-of-products expression for B_{OUT}.

Sum of products expression: B_{OUT} = $\bar{X}\cdot Bin + Y\cdot Bin + \bar{X}\cdot Y$

(B) (1 Point) Is the FS(X,Y,B_{IN}) circuit universal in the sense that 2-input NOR and 2-input NAND are universal? In other words, using only acyclic networks of FS circuits (perhaps with one or more of their inputs tied to "0" or "1"), can one implement any combinational logic function?

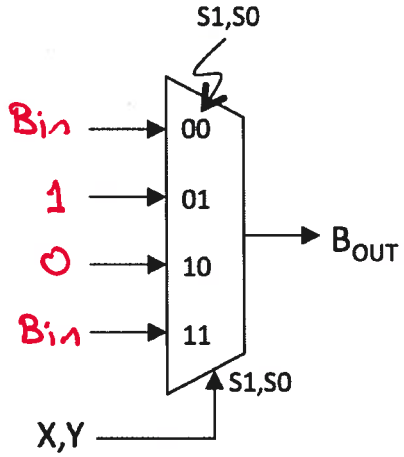
F(0,Y,Bin) = OR(Y,Bin) using B_{OUT}
F(1,Y,Bin) = AND(Y,Bin) using B_{OUT}
F(0,1,Bin) = NOT(Bin) using D

FS Universal: ... YES ... NO ...

← everything you need to build sum-of-products

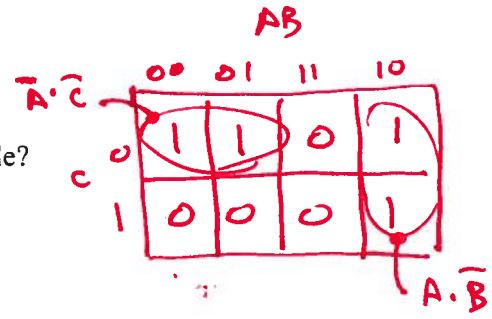
(C) (2 Points) You're trying to build an implementation for the B_{OUT} part of the FS circuit (see truth table above) but discover that the NITGFOC supply room only has 4-to-1 multiplexors in stock. In desperation, you call up your 6.004 TA who says "No problem! In fact, you can produce B_{OUT} with just a single 4-to-1 mux: connect X to S1 and Y to S0, then hook each of the data inputs to the appropriate choice of '0', '1' or B_{IN}." Using this hint, finish off the implementation shown below.

Show connections for data inputs using only '0', '1' or B_{IN}



Problem 10.

The 3-input Boolean function G(A,B,C) computes $\overline{A}\cdot\overline{C}+A\cdot\overline{B}+\overline{B}\cdot\overline{C}$.



(A) How many 1's are there in the output column of G's 8-row truth table?

4

(B) Give a minimal sum-of-products expression for G.

$\overline{A}\cdot\overline{C} + A\cdot\overline{B}$

(C) There's good news and bad news: the bad news is that the stockroom only has G gates. The good news is that it has as many as you need. Using only combinational circuits built from G gates, one can implement (choose the best response)

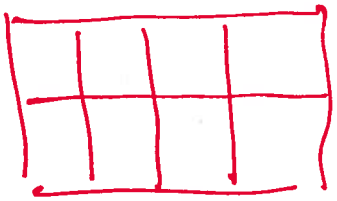
(A) only inverting functions
~~(B)~~ only non-inverting functions
(C) any function (G is universal)
(D) only functions with 3 inputs or less
(E) only functions with the same truth table as G

G(A,1,C) == NAND(A,C)

(D) Can a sum-of-products expression involving 3 input variables with greater than 4 product terms *always* be simplified to a sum-of-products expression using fewer product terms?

YES (see 6B)

Think of 3-input K-map



↑ if more than 4 1's, two must be adjacent ⇒ can use patch to cover both 1's.

6.004 Computation Structures
Spring 2017

Computation Structures

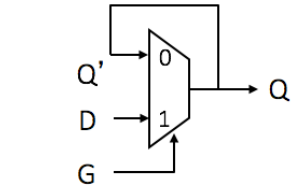
Sequential Logic Worksheet

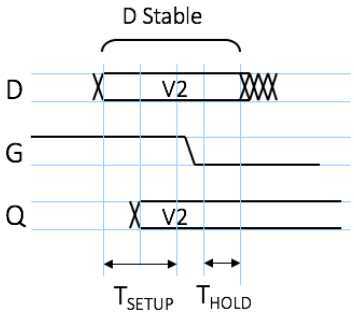
Concept Inventory:

- D-latch & the Dynamic Discipline
- D-register
- Timing constraints for sequential circuits
- Set-up and hold times for sequential circuits

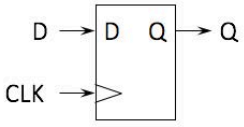
Notes:

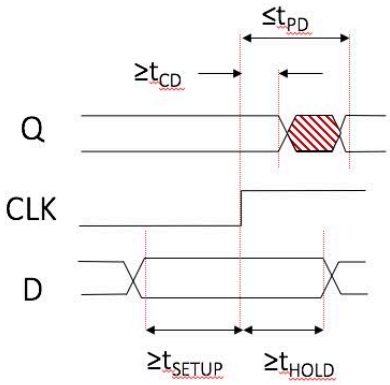
D latch

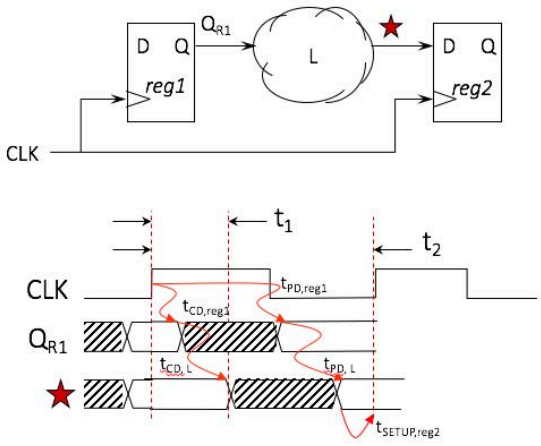




D register

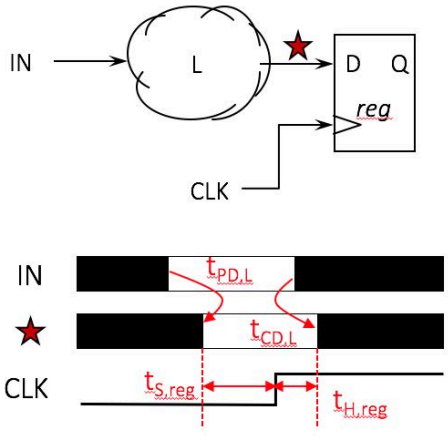






$t_1 = t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$

$t_2 = t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2} \leq t_{CLK}$

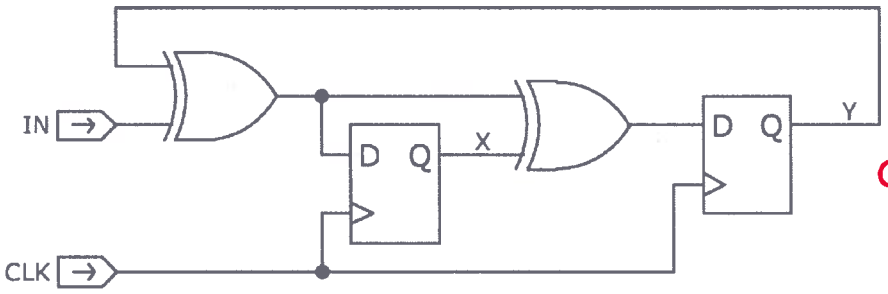


$t_{S,INPUT} = t_{PD,L} + t_{S,R}$

$t_{H,INPUT} = t_{H,R} - t_{CD,L}$

Problem 1.

Consider the following sequential logic circuit. It consists of one input IN, a 2-bit register that stores the current state, and some combinational logic that determines the state (next value to load into the register) based on the current state and the input IN.



GOAL: Dynamic discipline obeyed at input to each register

(A) Using the timing specifications shown below for the XOR and DREG components, determine the shortest clock period, t_{CLK}, that will allow the circuit to operate correctly or write NONE if no choice for t_{CLK} will allow the circuit to operate correctly and briefly explain why.

Component	t_{CD}	t_{PD}	t_{SETUP}	t_{HOLD}
XOR2	0.15ns	2.1ns	–	–
DREG	0.1ns	1.6ns	0.4ns	0.2ns

Minimum value for t_{CLK} (ns): 6.2
or explain why none exists

$t_{CLK} \geq t_{PD,REG} + 2 * t_{PD,XOR} + t_{SETUP,REG} = 1.6 + 2*2.1 + 0.4$

(B) Using the same timing specifications as in (A), determine the setup and hold times for IN with respect to the rising edge of CLK.

$2 * t_{PD,XOR} + t_{SETUP,REG}$

t_{SETUP} for IN with respect to CLK↑ (ns): 4.6

$t_{HOLD,REG} - t_{CD,XOR}$

t_{HOLD} for IN with respect to CLK↑ (ns): 0.05

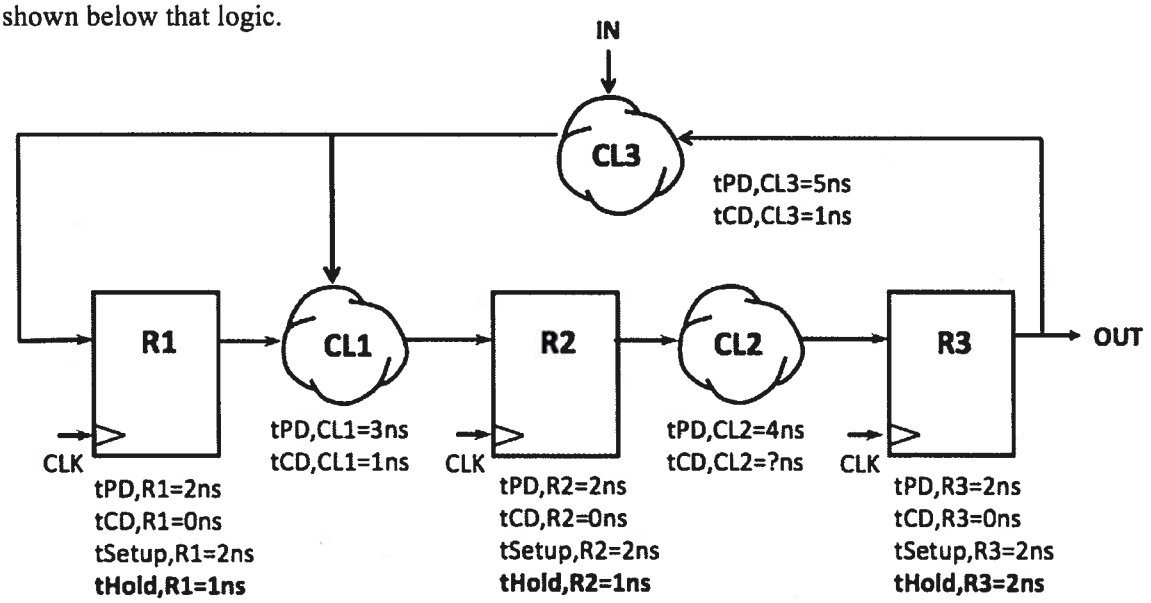
(C) One of the engineers on the team suggests using a new, faster XOR2 gate whose t_{CD} = 0.05ns and t_{PD} = 0.7ns. Determine a new minimum value for t_{CLK} or write NONE and explain why no such value exists.

Minimum value for t_{CLK} (ns): NONE
or explain why none exists

Now $t_{CD,REG} + t_{CD,XOR}$ is not greater than $t_{HOLD,REG}$

Problem 2.

Consider the following sequential logic circuit. It consists of three D registers, three different pieces of combinational logic (CL1, CL2, and CL3), one input IN, and one output OUT. The propagation delay, contamination delay, and setup time of the registers are all the same and are specified below each register. **The hold time for the registers is NOT the same** and is specified in bold below each register. The timing specification for each combinational logic block is shown below that logic.



(A) (1 point) What is the smallest value for the t_{CD} of CL2 that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

$t_{CD,R2} + t_{CD,CL2} \geq t_{HOLD,R3}$

$0 + ? \geq 2$

Smallest value for t_{CD} of CL2 (ns): 2

(B) (2 points) What is the smallest value for the period of CLK (i.e., t_{CLK}) that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

$(\sum t_{PD}) + t_{SETUP}$ R1→R2: 2+3+2 R3→R1: 2+5+2

R2→R3: 2+4+2 ~~R3→R2: 2+5+3+2~~

Smallest value for t_{CLK} (ns): 12

(C) (2 points) What are the smallest values for the setup and hold times for IN relative to the rising edge of CLK that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

longest path: $t_{PD,CL3} + t_{PD,CL1} + t_{S,R2} = 10$

Setup time for IN (ns): 10

shortest path: $t_{H,R1} - t_{CD,CL3} = 1 - 1 = 0$

Hold time for IN (ns): 0

(D) (2 points) What are the propagation delay and contamination delay of the output, OUT, of this circuit relative to the rising edge of the clock?

from R3

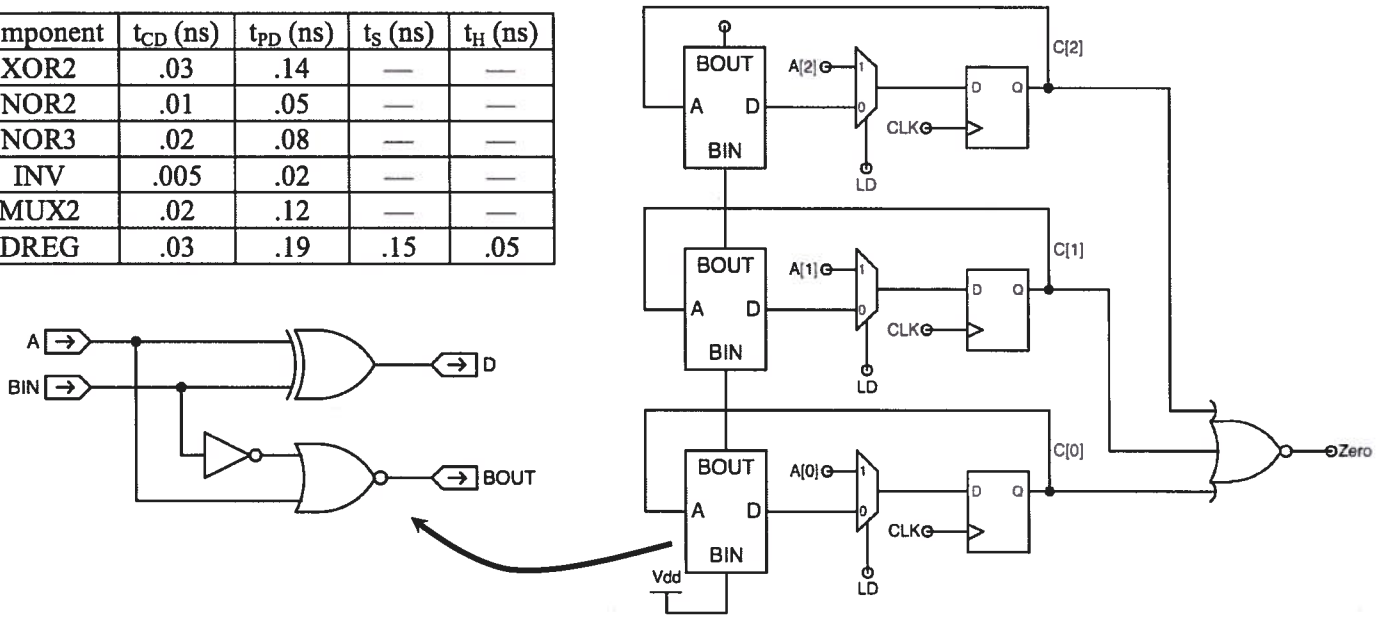
t_{PD} for OUT (ns): 2

t_{CD} for OUT (ns): 0

Problem 3.

Here's a schematic for a 3-bit loadable down-counter, which uses a ripple decrementer as a building block:

component	t_{CD} (ns)	t_{PD} (ns)	t_S (ns)	t_H (ns)
XOR2	.03	.14	—	—
NOR2	.01	.05	—	—
NOR3	.02	.08	—	—
INV	.005	.02	—	—
MUX2	.02	.12	—	—
DREG	.03	.19	.15	.05



(A) Using the contamination delays (t_{CD}), propagation delays (t_{PD}), setup times (t_S), and hold times (t_H) shown in the table above, please compute the minimum value for the clock period (t_{CLK}) for which the circuit will work correctly.

$(\Sigma t_{PD}) + t_{setup}$ along longest path **minimum value for t_{CLK} (ns):** 0.72

$t_{CLK} \geq t_{PD,REG} + t_{PD,NOR2} + t_{PD,INV} + t_{PD,NOR2} + t_{PD,XOR2} + t_{PD,MUX2} + t_{SETUP}$
$.19 \quad .05 \quad .02 \quad .05 \quad .14 \quad .12 \quad .15$

(B) What are the appropriate values for the setup (t_S) and hold (t_H) times for the **LD** input with respect to the rising edge of the clock?

$t_{S,LD} = t_{S,REG} + t_{PD,MUX2}$ $\quad .15 \quad .12$

setup time (t_S) for LD: 0.27

$t_{H,LD} = t_{H,REG} - t_{CD,MUX2}$ $\quad .05 \quad .02$

hold time (t_H) for LD: 0.03

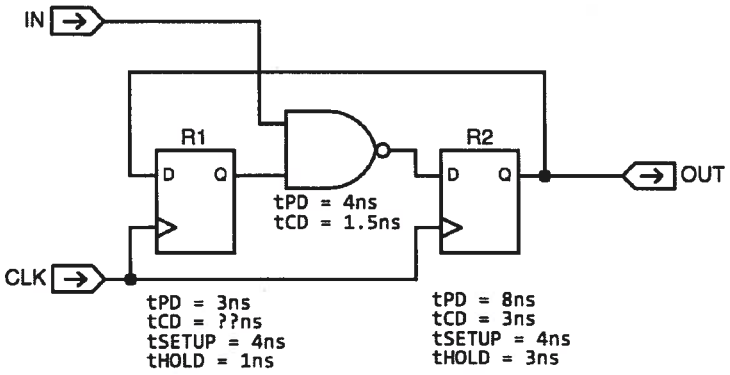
(C) What is the t_{PD} for the Zero output with respect to the rising edge of CLK?

t_{PD} for Zero (ns): 0.27

$t_{PD,Zero} = t_{PD,REG} + t_{PD,NOR3}$
$.19 \quad .08$

Problem 4.

Consider the following sequential logic circuit. The timing specifications are shown below each component. Note that the two registers do NOT have the same specifications.



(A) What are the smallest values for the setup and hold times for IN relative to the rising edge of CLK that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

$t_{S,IN} = t_{PD,NAND} + t_{SETUP,R2}$ (4 + 4)

Setup time for IN (ns): 8

$t_{H,IN} = t_{HOLD,R2} - t_{CD,NAND2}$ (3 − 1.5)

Hold time for IN (ns): 1.5

(B) What is the smallest value for the period of CLK (i.e., tCLK) that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

$t_{CLK} \geq$ { R1→R2: 3+4+4 = 11; R2→R1: 8+4=12 }

Smallest value for tCLK (ns): 12

(C) What is the smallest for the tCD of R1 that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

$t_{CD,R1} + t_{CD,NAND2} \geq t_{HOLD,R2}$ (? + 1.5 ≥ 3)

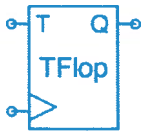
Smallest value for tCD of R1 (ns): 1.5

(D) Suppose two of these sequential circuits were connected in series, with the OUT signal of the first circuit connected to the IN signal of the second circuit. The same CLK signal is used for both circuits. Now what is the smallest value for the period of CLK (i.e., tCLK) that will guarantee the dynamic discipline is obeyed for all the registers in the circuit?

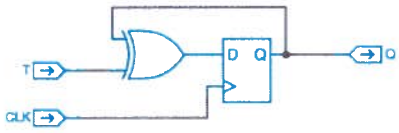
Smallest value for tCLK (ns): 16

$t_{CLK} \geq t_{PD,R2} + t_{PD,NAND}^{next} + t_{S,R2}^{next}$ (8 + 4 + 4)

Problem 5.



It is often useful to make clocked devices that count in binary, and a simple building block for such binary counters is the toggle flipflop whose symbol is shown on the right. It is a clocked device, hence the clock input indicated by the triangle on its lower-left edge. The other input, T (for *toggle*), may be set to one to cause the TFlop to flip its state (the *Q* output) from 0 to 1 or vice versa on the next active (positive) clock edge. If T is zero at an active clock edge, the state of the TFlop remains unchanged. We assume that the initial state of each TFlop at power-up is *Q=0*; more sophisticated versions might feature a *Reset* input to force a *Q=0* state.



A TFlop may be implemented using a D flipflop like the ones developed in lecture together with an XOR2 gate, as shown to the left.

As is our convention for clocked devices, we would like to specify timing specs for the TFlop as t_{CD}, t_{PD}, t_{SETUP}, and t_{HOLD}, all measured relative to the active (positive) clock edge.

(A) The timing specifications for the components are shown in the table below. Give appropriate values for the timing specifications of the TFlop implementation shown above.

Component	t_{CD}	t_{PD}	t_{SETUP}	t_{HOLD}
XOR2	40ps	400ps	–	–
DREG	100ps	300ps	80ps	40ps

$t_{CD,REG}$ **t_{CD}:** 100 **ps**

$t_{PD,REG}$ **t_{PD}:** 300 **ps**

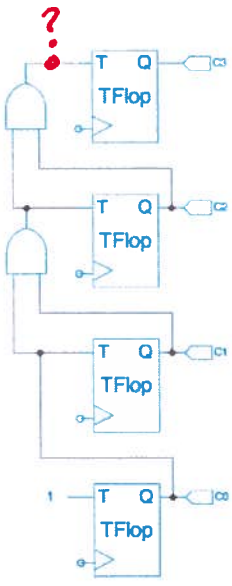
$t_{PD,XOR} + t_{S,REG}$ **t_{SETUP}:** 480 **ps**

$t_{H,REG} - t_{CD,XOR}$ **t_{HOLD}:** 0 **ps**

(B) Suppose we connect the T input of a single TFlop to 1 (i.e., V_{DD}) and try to clock it at its maximum rate. What is the minimum clock period we can use and expect the TFlop to perform properly?

$t_{CLK} \geq t_{PD,REG} + t_{CD,XOR} + t_{S,REG}$ (300 + 400 + 80)

***Minimum* clock period for correct operation:** 780 **ps**



We next consider the use of four TFlops to make a 4-bit ripple-carry counter as shown to the left. Assume that the TFlops share a common clock input (not shown) with an appropriate period, and that all TFlops have an initial Q=0 state.

(C) Suppose we run this circuit for a large number, N, of clock cycles. For approximately how many of the N active clock edges would you expect the T input to the topmost TFlop to be 1?

TOPMOST T=1 when C0 = C1 = C2 = 1 → every 8 cycles

***Topmost* T=1 occurrences in N cycles:** N/8

(D) If the AND2 gates have t_{PD} =200ps and t_{CD} =40ps, what is the minimum clock period we can use for the 4-bit counter?

***Minimum* clock period for correct operation:** 1180 **ps**

$t_{CLK} \geq t_{PD,REG} + 2 * t_{PD,AND} + t_{PD,XOR} + t_{S,REG}$ (300 + 2*200 + 400 + 80)

6.004 Computation Structures
Spring 2017

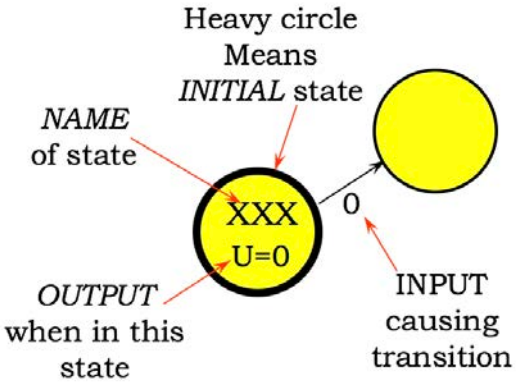
Computation Structures

Finite State Machines Worksheet

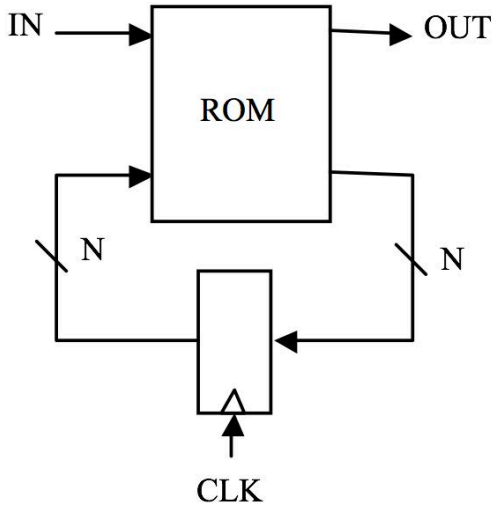
Concept Inventory:

- State transition diagrams & FSM truth tables
- Register & ROM implementation
- Equivalent FSMs; equivalent state reduction
- Metastability: causes and cures

Notes:



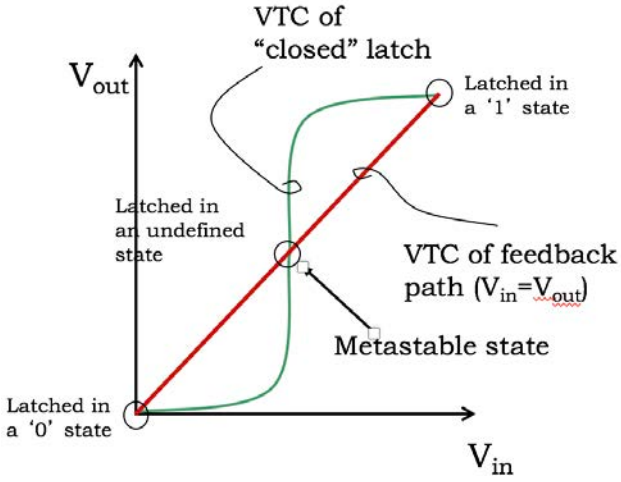
Arcs leaving a state must be (1) mutually exclusive, and (2) collectively exhaustive.



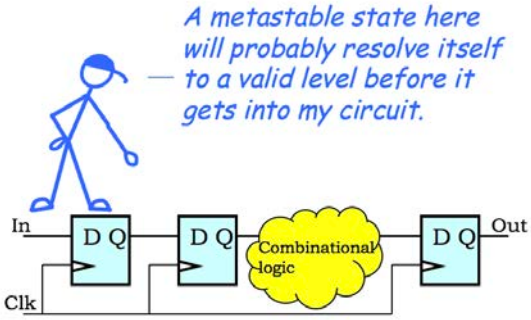
FSMs are EQUIVALENT if and only if every inputs sequence yields identical output sequences.

Two states are equivalent if
1. both states have identical outputs, AND
2. every input transitions to equivalent states.

Metastability:

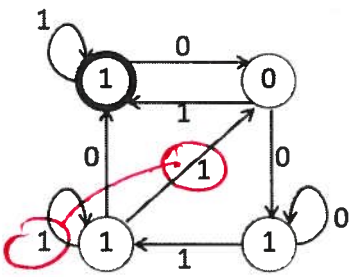


Quarantine time reduces p(metastable)

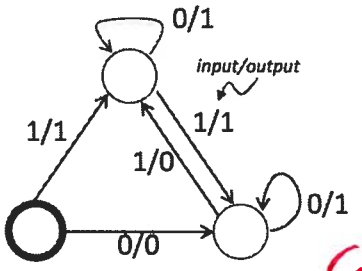


Problem 1.

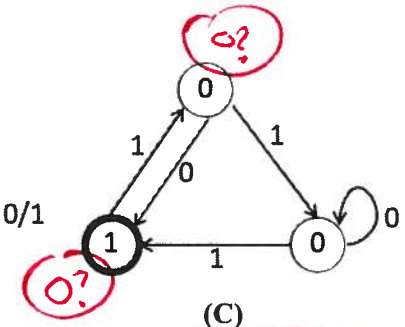
(A) For each of the following FSMs please indicate if they are or are not well formed. Note that the state names have been omitted for clarity; you may assume the state names are unique.



(A)



(B)



(C)

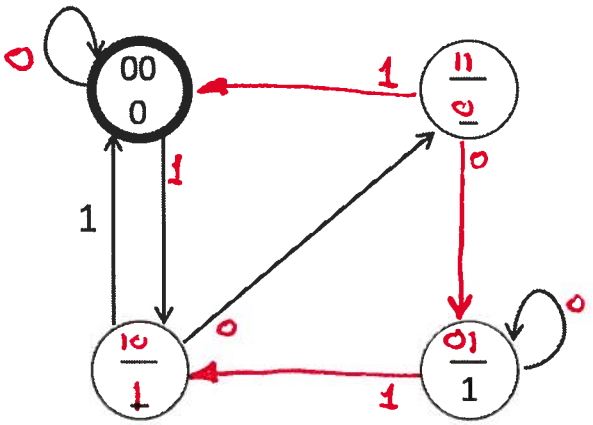
FSM A (circle one): Well Formed / Not Well Formed

FSM B (circle one): Well Formed / Not Well Formed

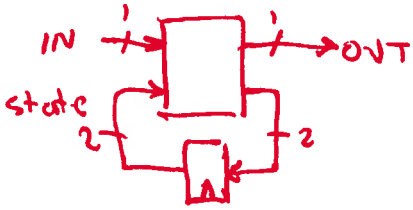
FSM C (circle one): Well Formed / Not Well Formed

(B) Given the partially completed truth table and FSM diagram below. **Complete all the missing entries** in the truth table and the FSM diagram. The FSM is a Moore machine, i.e., the Out signal is determined only by the current state. In each state circle, the top entry is S_1S_0 and the bottom entry is the value of Out. Make sure that you have labeled all missing states, inputs, and outputs, and that you have added and labeled any missing transitions in the FSM.

S1	S0	In	S1'	S0'	Out
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	1	1	1
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	0	0	0



(C) If this FSM is implemented using a 2-bit state register and a ROM, what size ROM would be needed? Please specify the number of locations (entries) of the ROM, and the width of each entry.

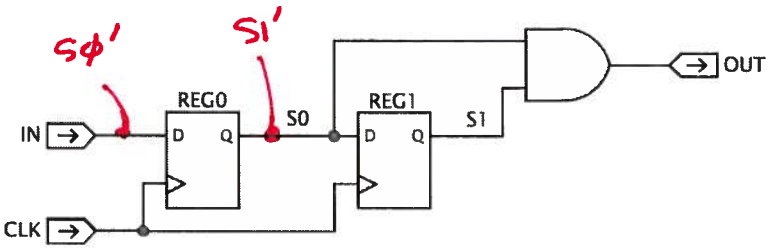


Number of locations in ROM: $2^3 = 8$

Width of each ROM entry (bits): $2+1 = 3$

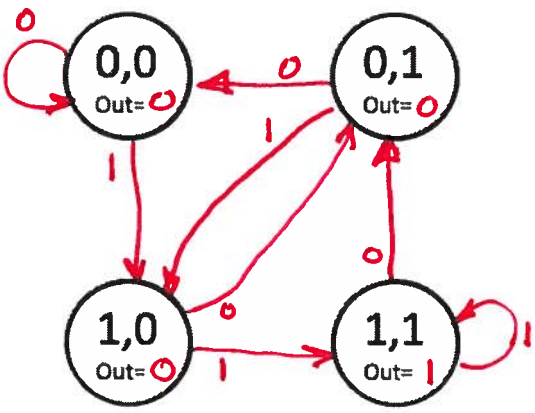
Problem 2.

Consider the sequential logic circuit to the right, which implements an FSM with a single data input IN and single data output OUT. Assume that all signal transitions are timed so that the dynamic discipline is satisfied at each register.



Please describe the operation of the FSM by filling in both the state transition diagram and the truth table shown below. The two-digit state names in the state transition diagram are S0,S1, the logic values present at the outputs of REG0 and REG1 after the rising edge of the clock. In the truth table, S0' and S1' are the values that will loaded into REG0 and REG1 at the next rising clock edge.

Fill in state transition diagram and truth table



S0	S1	IN	S0'	S1'	OUT
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	1	1
1	1	1	1	1	1

IN S0

Problem 3.

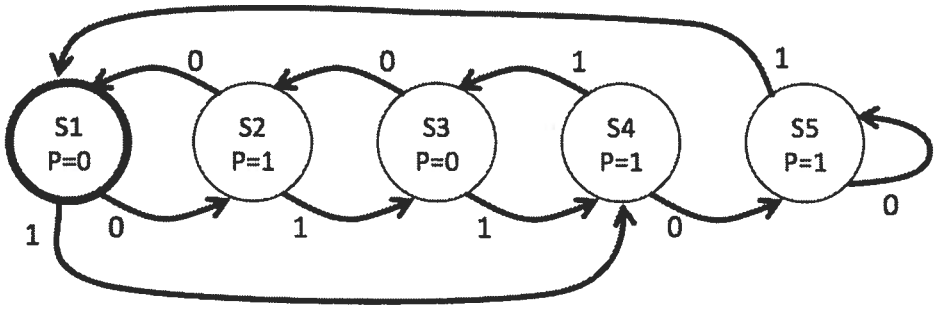
A "Thingee" is a clocked device built out of 3 interconnected components, each of which is known to be a 4-state FSM. What bound, if any, can you put on the number of states of a Thingee?

Each FSM might be in one of its 4 states.

Max # of states, or "Can't Tell": 4·4·4 = 64

Problem 4.

Consider the 1-input, 1-output finite state machine with the state transition diagram shown below. Note that the single output P only depends on the current state of the FSM.



(A) (1 Point) The FSM has been processing inputs for a while and we would like to determine its current state. After entering three additional inputs "000", we observe that we have reached a state where P=0. Please circle the possible values for the state *before* the additional three inputs were entered.

Possible values for state before: **S1** **S2** **S3** **S4** **S5**

(B) (2 Points) Assume that the states are represented by the 3-bit binary values given on the left below. Please fill in the appropriate entries for the partial truth table shown on the right where S is the current state, I is the input value, S' is the next state, P is the output value

State	*Encoding*
S1	001
S2	010
S3	011
S4	100
S5	101

Fill in partial truth table

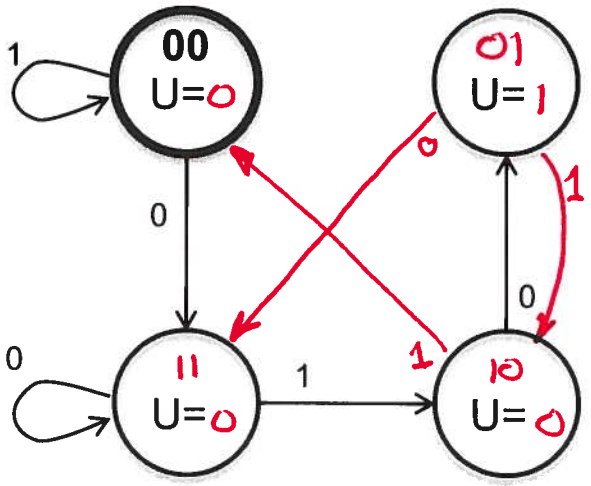
S	*I*	*S'*	*P*
011	0	010	0
011	1	100	0
100	0	101	1
100	1	011	1

(C) Please identify which, if any, states are equivalent. For example, if states S1, S2, and S4 are equivalent, please write "(S1,S2,S4)". You may need multiple parenthesized lists if more than one set of states is equivalent.

Equivalent states: (S1,S3) (S4,S5)

Problem 5.

Perfectly Perplexing Padlocks makes an entry-level electronic lock, the P3b, built from an FSM with two bits of state. The P3b has two buttons ("0" and "1") that when pressed cause the FSM controlling the lock to advance to a new state. In addition to advancing the FSM, each button press is encoded on the B signal (B=0 for button "0", B=1 for button "1"). The padlock unlocks when the FSM sets the UNLOCK output signal to 1, which it does **whenever—and only whenever—the last 3 button presses correspond to the 3-digit combination**. The combination is unique, and will open the lock independently of the starting state. Unfortunately the design notes for the P3b are incomplete.



S_1	S_0	B	S'_1	S'_0	U
0	0	0	1	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	1	1	0
1	1	1	1	0	0

(A) (1 Point) What is the 3-bit combination for the lock?

lock combination: 010

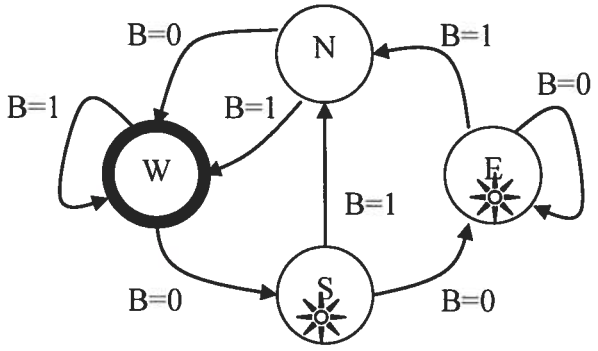
(B) (5 Points) Using the specification and clues from the partially completed diagrams above fill in the information that is missing from the state transition diagram and its accompanying truth table. When done:

- each state in the transition diagram should be assigned a 2-bit state name S_1S_0 (note that in this design the state name is *not* derived from the combination that opens the lock),
- the arcs leaving each state should be mutually exclusive and collectively exhaustive,
- the value for U should be specified for each state, and
- the truth table should be completed.

(complete above transition diagram and table)

Problem 6.

Below is a state transition diagram for a 4-state FSM with a single binary input B. The FSM has single output – a light that is "on" when the FSM is in states "E" or "S". The starting state, "W", is marked by the heavy circle.



(A) (1 Point) Does this FSM have a set or sets of equivalent states that can be merged to yield an equivalent FSM with fewer states?

List set(s) of states that can be merged or write NONE: S, E

(B) (5 Points) Please fill in as many entries as possible in the following truth table for the FSM. The *light* output is a function of the current state and should be 1 when the light is "on" and 0 when it's "off."

	S1	S0	B	S1'	S0'	light
E	0	0	0	0	0	1
	0	0	1	1	0	1
S	0	1	0	0	0	1
	0	1	1	1	0	1
N	1	0	0	1	1	0
	1	0	1	1	1	0
W	1	1	0	0	1	0
	1	1	1	1	1	0

LIGHT=1 implies either S or E. only S transitions to two different states.

by process of elimination

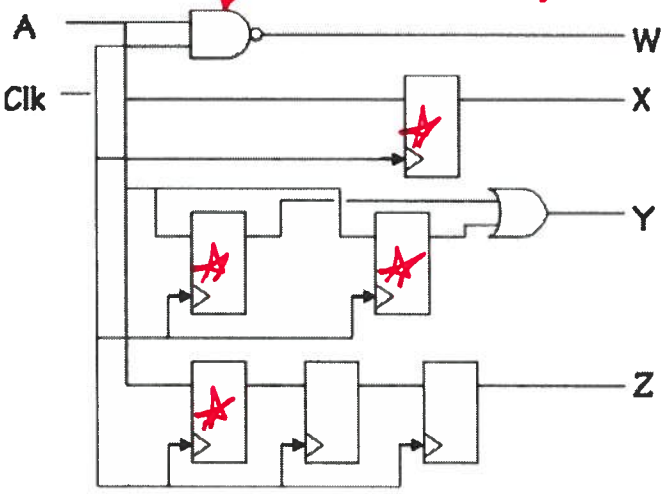
Problem 7.

The following circuit has two inputs (A, CLK) and four outputs (W, X, Y Z). The CLK signal is square wave with a period t_{CLK}=1us. The A signal makes a single 0→1 transition but the timing of the transition is close to (within a few ns of) the active CLK edge, ignoring dynamic discipline. All the devices are lenient and have the same propagation delay t_{PD}=10ns.

In a test involving a large number of trials, the **Z** output has been examined 100ns after an active CLK edge (and when *both CLK and A have been stable for many propagation delays*); at this time, **Z** was found to be invalid P times. In the same test, what would you expect to observe at the other outputs 100ns after the CLK edge? For each output, circle one of

LESS RELIABLE if you would expect the output to be **invalid** appreciably **more** than P times;
EQUALLY RELIABLE if you would expect the output to be **invalid** about P times; or
MORE RELIABLE if you would expect the output to be **invalid** appreciably **less** than P

combinational logic never goes metastable!



W reliability: LESS EQUAL MORE

X reliability: LESS EQUAL MORE

Y reliability: LESS EQUAL MORE

Z was found to be invalid P times.....

★ These registers are particularly susceptible to metastability since they are connected to input A.

6.004 Computation Structures
Spring 2017

Computation Structures

Pipelined Circuits Worksheet

Concept Inventory:

- Latency & throughput
- Pipelined circuits & conventions
- Pipeline diagrams
- Pipelining methodology: contours
- Pipelined components; interleaving

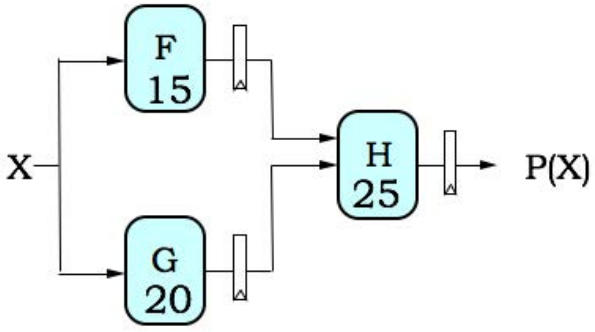
Notes:

Latency: the delay from when an input is established until the output associated with the input becomes valid.

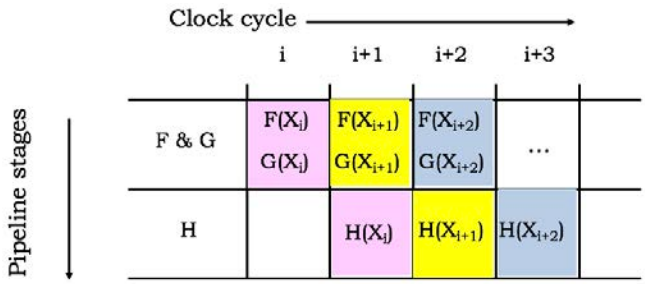
- Combinational circuits: $L = t_{PD}$
- K-pipeline: $L = K*t_{CLK}$

Throughput: the rate at which inputs or outputs are processed.

- Combinational circuits: $T = 1/L$
- K-pipeline: $T = 1/t_{CLK}$

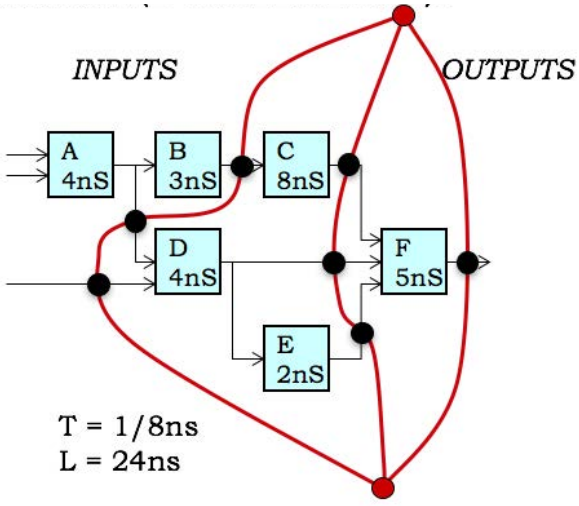


Unpipelined:
L = 45ns, T = 1/L = 1/(45ns)
2-stage pipeline [t_{CLK}=25ns]:
L = 2*25 = 50 ns, T = 1/(25ns)



Pipelining methodology:

- Form 1-pipeline by adding registers to all outputs
- To add a pipeline stage, draw contour across all paths from inputs to outputs such that it doesn't cross other contours and all input-output paths cross the contour in the same direction. This ensures the pipeline is well-formed (same # of registers on all input-output paths). A K-pipeline has K registers on all input-output paths.
- Contours must take into account pipelined or interleaved components. An N-way interleaved component behaves like N-pipeline.



Problem 1.

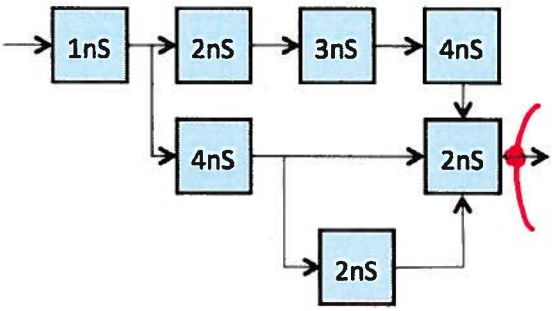
A simple combinational circuit is to be pipelined for **maximum throughput using a minimal number of registers**. For each of the questions below, please create a valid K-stage pipeline. **Show your pipelining contours** and place large black circles (●) on the signal arrows to **indicate the placement of ideal pipeline registers** (tPD=0, tSETUP=0). Give the latency and throughput for each design. Remember that our convention is to place a pipeline register on each output.

(A) (1 point). Show the maximum-throughput 1-stage pipeline.

K=1
t_{clk}=12
1 register

Latency (ns): 12

Throughput (ns-1): 1/12

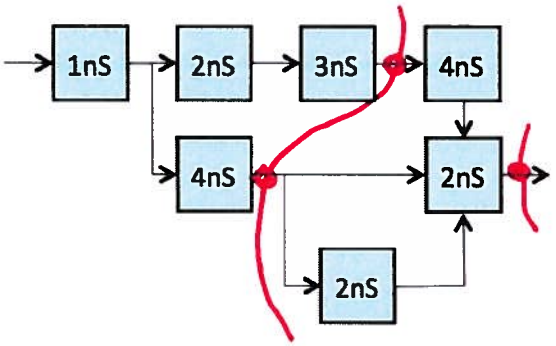


(B) (2 points). Show the maximum-throughput 2-stage pipeline using a minimal number of registers.

K=2
t_{clk}=6
3 registers

Latency (ns): 2×6=12

Throughput (ns-1): 1/6

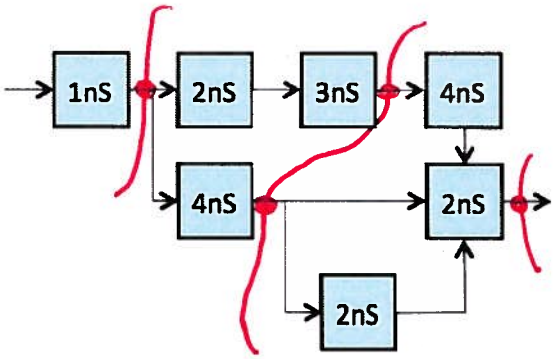


(C) (2 points). Show the maximum-throughput 3-stage pipeline using a minimal number of registers.

K=3
t_{clk}=6
4 registers

Latency (ns): 3×6=18

Throughput (ns-1): 1/6

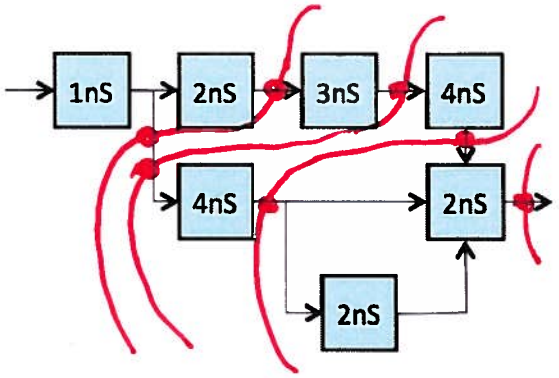


(D) (2 points). Show the maximum-throughput 4-stage pipeline using a minimal number of registers.

K=4
t_{clk}=4
7 registers

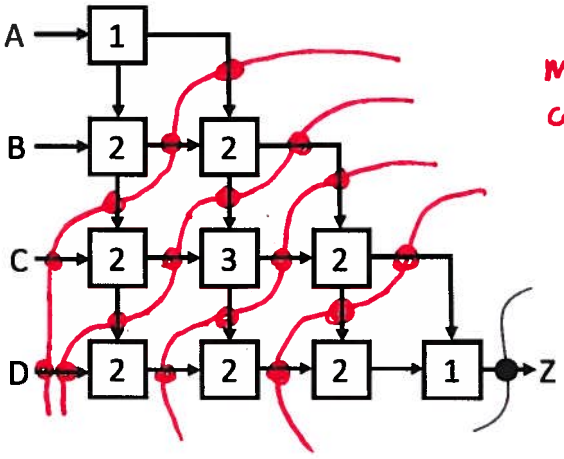
Latency (ns): 4×4=16

Throughput (ns-1): 1/4



Problem 2.

The following 1-stage pipelined circuit computes Z from the four inputs A, B, C, and D. Each component is annotated with its propagation delay in ns.



max t_{PD} for all components is 3 ⇒ $t_{CLK} = 3$?

(A) Please pipeline the circuit above for maximum throughput with the minimum possible latency using ideal pipeline registers ($t_{PD} = 0$, $t_{SETUP} = 0$). Show the location of pipeline registers in the diagram above using filled-in circles, like the one shown on the Z output. Please give the latency and throughput of the resulting pipelined circuit.

$k = 5$

$t_{CLK} = 3$

$k \cdot t_{CLK}$ **Latency (ns)?** $5 * 3 = 15$

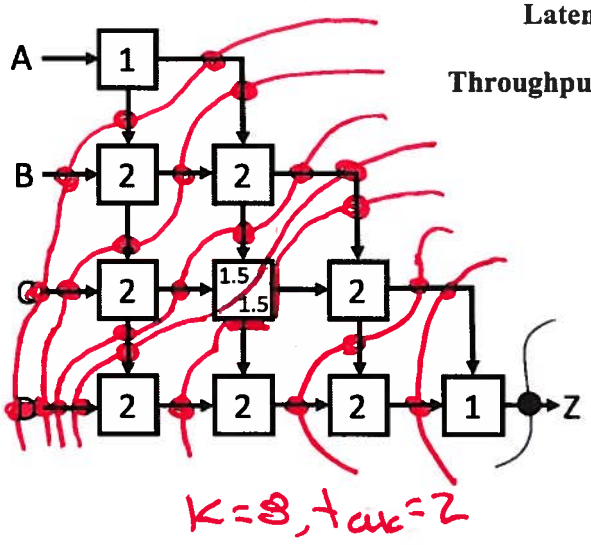
$1/t_{CLK}$ **Throughput (1/ns)?** $1/3$

(B) Now suppose the "3" component is replaced by a two-way interleaved component with a minimum t_{CLK} of 1.5ns. Recall that a two-way interleaved component behaves like a 2-stage pipelined component. Again, please pipeline the circuit below for maximum throughput with the minimum possible latency using ideal pipeline registers. Show the location of pipeline registers in the diagram below using filled-in circles, like the one shown on the Z output. Please give the latency and throughput of the resulting pipelined circuit.

Latency (ns)? $8 * 2 = 16$

Throughput (1/ns)? $1/2$

- we want $t_{CLK} = 2$
- interleaved component has pipeline regs built-in.

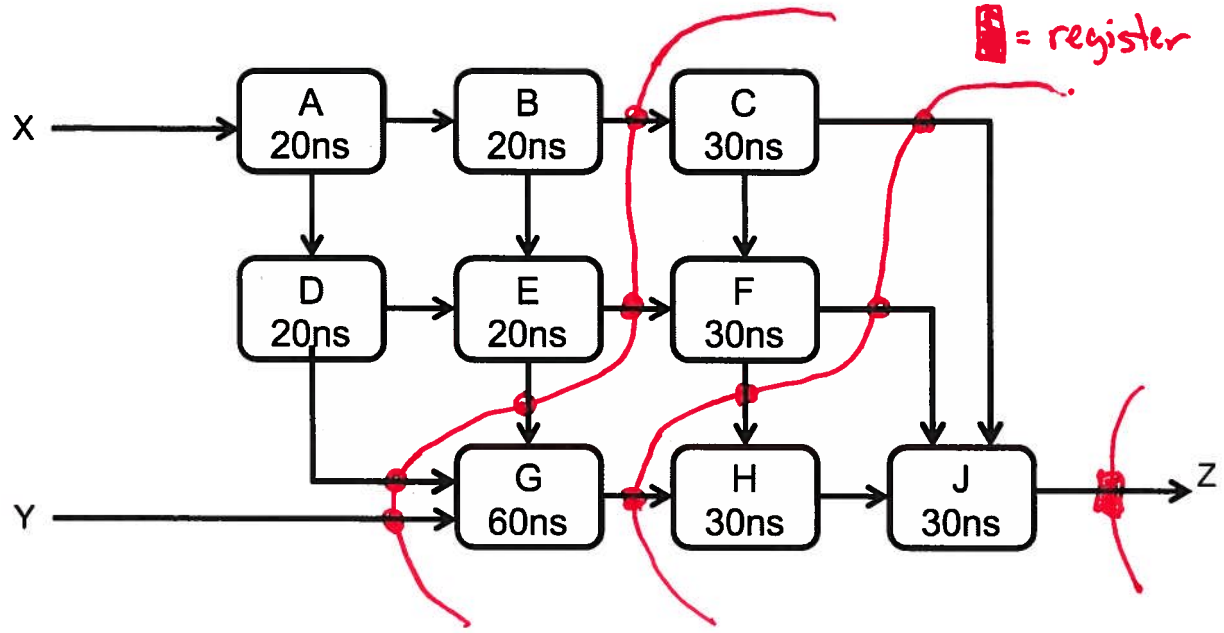


other diagrams possible

$k = 8, t_{CLK} = 2$

Problem 3.

An unidentified government agency has a design for a combinational device depicted below:



Although you don't know the function of each of the component modules, they are each combinational and marked with their respective propagation delays. You have been hired to analyze and improve the performance of this device.

(A) (1 Point) What are the throughput and latency for the unpipelined combinational device?

longest path: A B E G H J

Latency: 180 **ns; Throughput:** 1/180 **ns^{-1}**

(B) (4 Points) Show how to pipeline the above circuit for maximum throughput, by marking locations in the diagram where registers are to be inserted. **Use a minimum number of registers, but be sure to include one on the output.** Assume that the registers have 0 t_{PD} and t_{SETUP}.

$t_{ax} = 60$ $k = 3$ **(mark register locations in diagram above)**

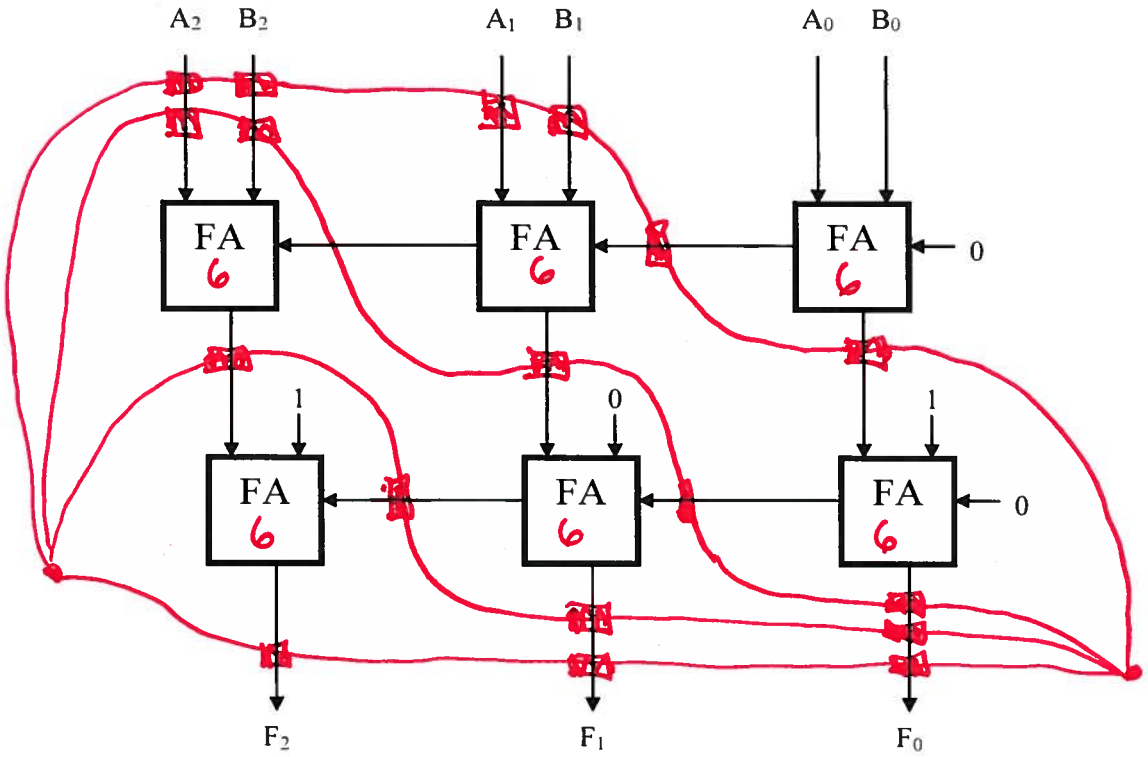
(C) (1 Point) What are the latency and throughput of your pipelined circuit?

Latency: 180 **ns; Throughput:** 1/60 **ns^{-1}**

$k \cdot t_{ax}$ $1/t_{ax}$

Problem 4.

The following circuit uses six full adder modules (as you've seen in lecture and lab) arranged in a combinational circuit that computes a 3-bit value **F=A+B+5** for 3-bit inputs **A** and **B**:



The full adders have a t_{PD} of 6ns.

(A) Give the latency and throughput of the combinational circuit.

4 FA modules on longest path **Latency:** 24 **ns; Throughput:** 1/24

(B) Indicate, on the above diagram, appropriate locations to place ideal (zero-delay) registers to pipeline the circuit for *maximum throughput* using a *minimum number of registers*. Be sure to include a register on each output.

(mark circuit above)

$t_{clk} = 6$ ns $k = 4$

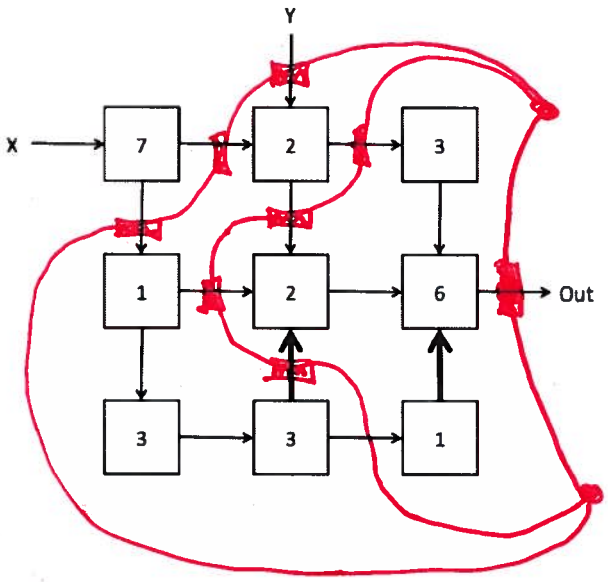
(C) Give the latency and throughput of your pipelined circuit.

Latency: 24 **ns; Throughput:** 1/6

$k \cdot t_{clk}$

Problem 5.

(A) You are provided with the circuit shown below. Each box represents some combinational logic. The number in each box is the t_{PD} of that combinational logic. The circuit has two inputs, X and Y, and one output Out. Pay close attention to the direction of the arrows especially the arrows shown in **bold**. What is the latency and throughput of this combinational circuit?



= pipeline register

longest path:
7+1+3+3+2+6 = 22

Latency (ns): 22

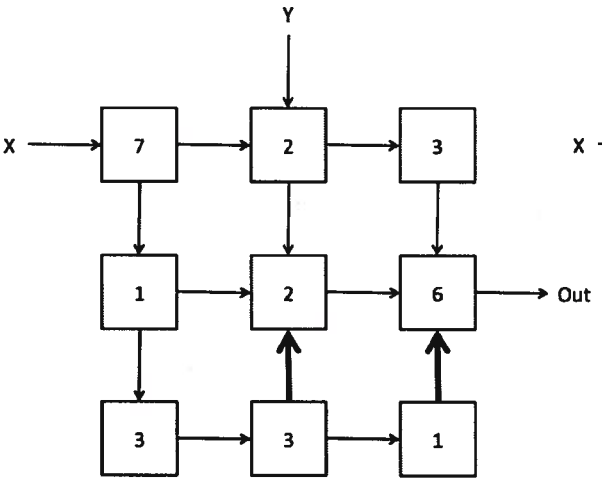
Throughput (1/ns): 1/L = 1/22

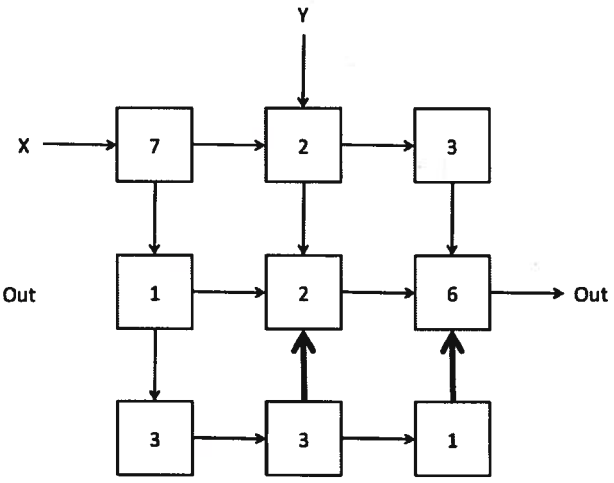
(B) Draw contours through the circuit above to produce a valid pipelined circuit **whose t_{CLK} = 9ns with minimum latency**. Extra copies of the diagram are included below. Please use a large dot to **indicate the location of each pipeline register**. Assume that you have ideal pipeline registers (t_{PD}=t_{CD}=t_{Setup}=t_{Hold}=0 ns). Pay close attention to the direction of each arrow to ensure that you produce a valid pipeline. What is the latency and throughput of this pipelined circuit?

see above: K=3

Latency (ns): $K \cdot t_{clk} = 3 \cdot 9 = 27$

Throughput (1/ns): $1/t_{clk} = 1/9$





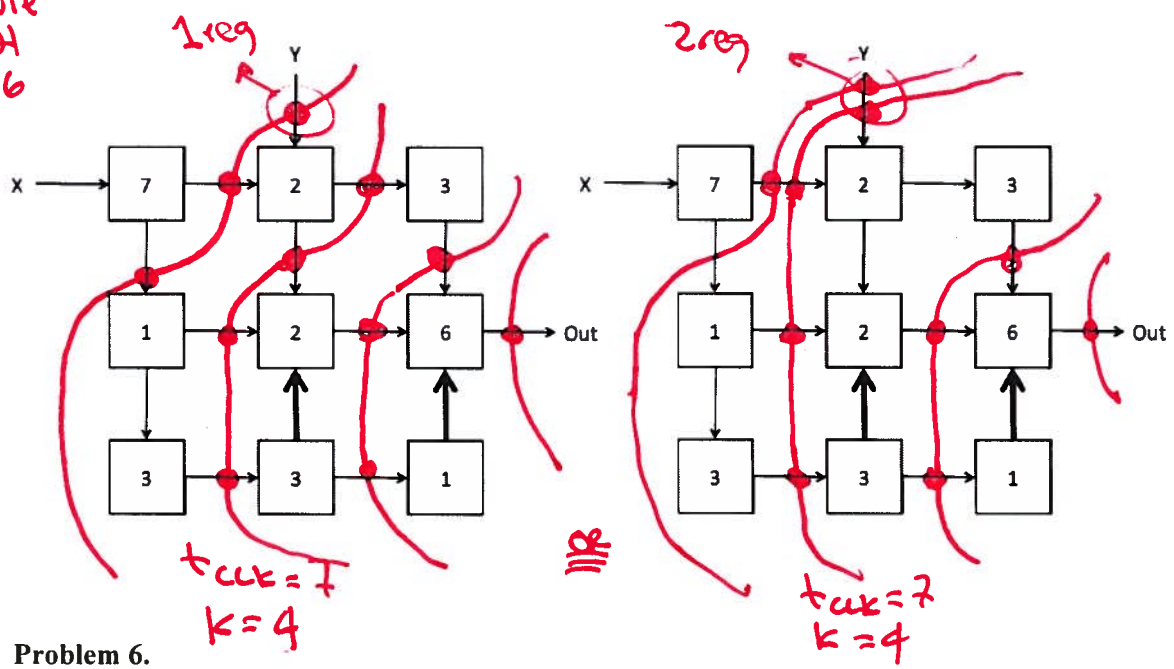
(C) You are now asked to consider the performance of this circuit using different clock periods while achieving the minimum latency. For each suggested t_{CLK}, specify whether or not you can create a **valid** pipelined circuit using that clock period. If you can, then provide the latency and throughput of the resulting circuit and specify the number of registers at each input. If it results in an invalid pipeline, enter NA for the rest of the row.

Extra copies of the circuit diagram are provided below.

t_{CLK}	Valid/Invalid	Latency (ns)	Throughput (1/ns)	Pipeline registers at input X	Pipeline registers at input Y
6 ns	INVALID	NA	NA	NA	NA
7 ns	VALID	$2k$	$1/7$	0	1 or 2

$k \cdot t_{clk}$ $1/t_{clk}$

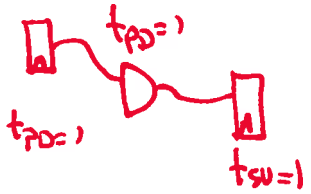
7 module rules out $t_{clk} = 6$



Problem 6.

A complex combinational circuit is constructed entirely from 2-input NAND gates having a propagation delay of 1 ns. If this circuit is pipelined for maximal throughput by adding (non-ideal) registers whose setup time and propagation delay are each 1 ns, what is the throughput of the resulting pipeline? Enter a number, a formula, or "CAN'T TELL".

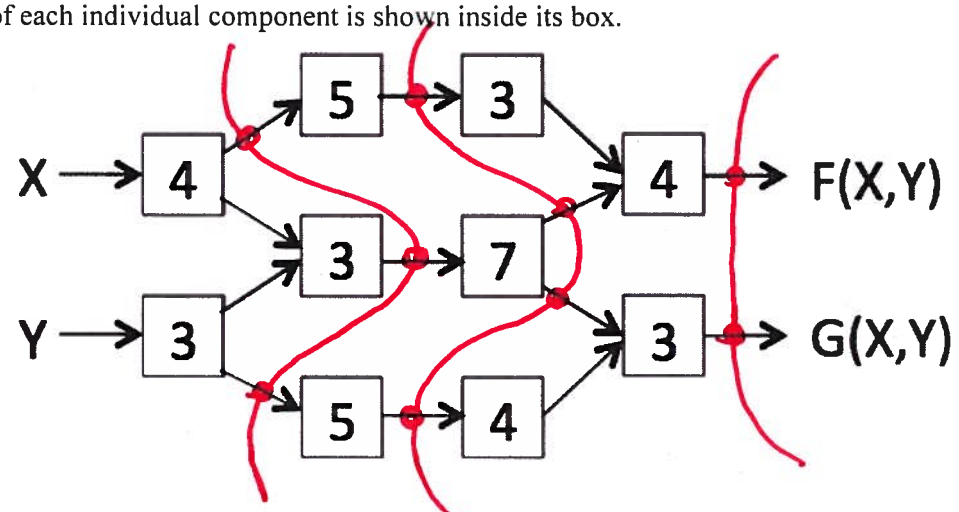
Throughput (ns^{-1}): $\frac{1}{t_{clk}} = \frac{1}{3}$



$t_{clk} = t_{PD,REG} + t_{PD,NAND} + t_{SU,REG}$
$= 1 + 1 + 1$

Problem 7.

The following combinational circuit computes F(X,Y) and G(X,Y) from inputs X and Y. The t_{PD} (in ns) of each individual component is shown inside its box.



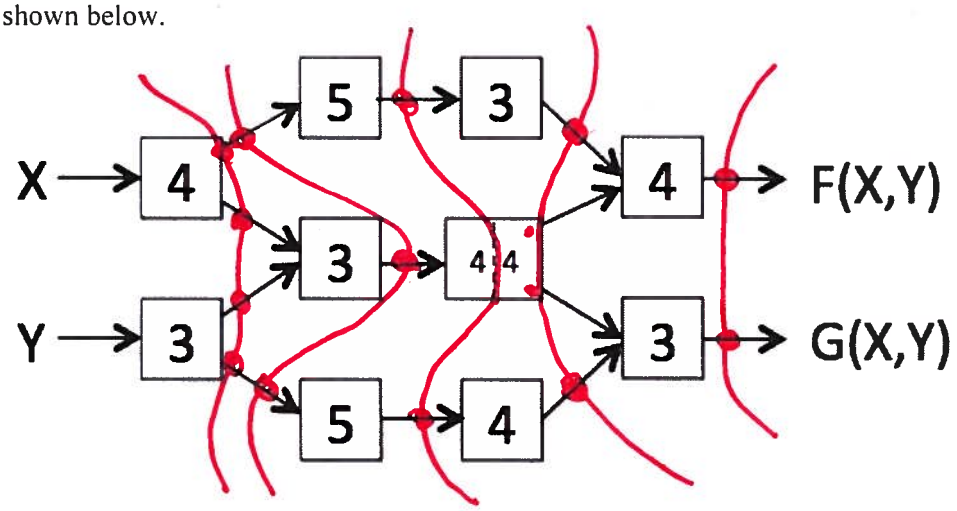
(A) Using ideal zero-delay registers, mark the location of the minimal number of registers necessary to achieve maximum throughput. Give the latency and throughput of your pipelined circuit.

mark diagram above

$t_{CLK} = 7$
$k = 3$

Latency: 21 **(ns); Throughput:** 1/7 **(1/ns)**

Rummaging through the stockroom you find a pipelined component with two pipeline stages that can replace the "7" module. The minimum t_{CLK} for the new component is 4ns. The updated circuit is shown below.



(B) Using ideal zero-delay registers, mark the location of the minimal number of registers necessary to achieve maximum throughput. Give the latency and throughput of your pipelined circuit.

mark diagram above

$t_{CLK} = 5$
$k = 5$

Latency: 25 **(ns); Throughput:** 1/5 **(1/ns)**

6.004 Computation Structures
Spring 2017

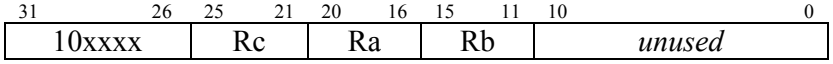
Computation Structures

Instruction Set Architecture Worksheet

Summary of β Instruction Formats

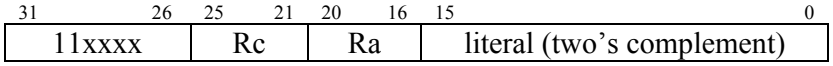
Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

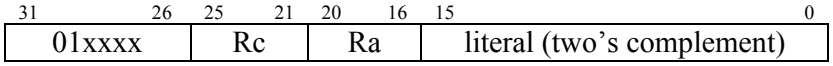
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or), **XNOR** (bitwise exclusive nor),
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or), **XNORC** (bitwise exclusive nor)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/**BF**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/**BT**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**	**BEQ**	**BNE**		**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**	**XNOR**	**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**	**XNORC**	**SHLC**	**SHRC**	**SRAC**	

Problem 1.

An unnamed associate of yours has broken into the computer (a Beta of course!) that 6.004 uses for course administration. He has managed to grab the contents of the memory locations he believes holds the Beta code responsible for checking access passwords and would like you to help discover how the password code works. The memory contents are shown in the table below:

	Addr	Contents	Opcode	Rc	Ra	Rb	Assembly
	0x100	0xC05F0008	110000	00010	11111	_____	ADDC(R31, 0x8, R2)
	0x104	0xC03F0000	110000	00001	11111	_____	ADDC(R31, 0x0, R1)
L2:	0x108	0xE060000F	111000	00011	00000	_____	ANDC(R0, 0xF, R3)
	0x10C	0xF0210004	111100	00001	00001	_____	SHLC(R1, 0x4, R1)
	0x110	0xA4230800	101001	00001	00011	00001	OR(R3, R1, R1)
	0x114	0xF4000004	111101	00000	00000	_____	SHRC(R0, 0x4, R0)
	0x118	0xC4420001	110001	00010	00010	_____	SUBC(R2, 0x1, R2)
	0x11C	0x73E20002	011100	11111	00010	_____	BEQ(R2, L1, R31)
		+2					
	0x120	0x73FFFFF9	011100	11111	11111	_____	BEQ(R31, L2, R31)
		-7					
	0x124	0xA4230800	101001	00001	00011	_____	not executed!
L1:	0x128	0x605F0124	011000	00010	11111	_____	LD(R31, 0x124, R2)
	0x12C	0x90211000	100100	00001	00001	_____	CMPEQ(R1, R2, R1)

Further investigation reveals that the password is just a 32-bit integer which is in R0 when the code above is executed and that the system will grant access if R1 = 1 after the code has been executed. What "passnumber" will gain entry to the system?

The loop reverses the order of the nibbles (4-bit chunks) of the value in R0, eg., 0x12345678 becomes 0x87654321.

So the "passnumber" is the nibble reverse of 0xA4230800 which is 0x00830324A.

Problem 2.

(A) What assembly instruction could a compiler use to implement y = x * 8 on the Beta assuming that MUL and MULC are not available? Assume x is in R0 and y is in R1.

Equivalent assembly instruction: SHLC(R0,3,R1)

(B) Assume that the registers are initialized to: R0=8, R1=10, R2=12, R3=0x1234, R4=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. SHL(R3, R4, R5) **Value of R5:** 0x3400 0000
2. ADD(R2, R1, R6) **Value of R6:** 22
3. ADD(R0, 2, R7) **Value of R7:** 20
4. ST(R1, 4, R3) **Value stored:** 10 **at address:** 4 + 0x1234 = 0x1238

op = ADDC so interpreted as R2!

(C) A student tries to optimize his Beta assembly program by replacing a line containing

ADDC(R0, 3*4+5, R1)

by

ADDC(R0, 17, R1)

expression value computed at assembly time

Is the resulting binary program smaller? Does it run faster?

(circle one) Binary program is SMALLER? yes ... **no**

(circle one) FASTER? yes ... **no**

(D) A BR instruction at location 0x1000 branches to 0x2000. If the binary representation for that BR were moved to location 0x1400 and executed there, where will the relocated instruction branch to?

Original branch offset (0x1000) now relative to 0x1400

Branch target for relocated BR (in hex): 0x 2400

(E) A line in an assembly-language program containing "ADDC(R1,2,R3)" is changed to "ADDC(R1,R2,R3)". Will the modified program behave differently when executed?

Interpret 2nd operand as a constant expression.

Value of symbol R2 is 2.

Circle best answer: YES ... NO ... CAN'T TELL

Problem 3

Each of the following programs is loaded into a Beta's main memory starting at location 0 and execution is started with the Beta's PC set to 0. Assume that all registers have been initialized to 0 before execution begins. Please determine the specified values after execution reaches the HALT() instruction and the Beta stops. Write "CAN'T TELL" if the value cannot be determined. **Please write all values in hex.**

(A)

```
        . = 0
        LD(R31,X+4,R1)
        SHLC(R1,2,R1)
        LD(R1,X,R2)
        HALT()
   X:   LONG(4)
        LONG(3)
        LONG(2)
        LONG(1)
        LONG(0)
```

R1 ← 3
R1 ← 12
R2 ← Mem[x+12]

Value left in R1: 0x C

Value left in R2: 0x 1

(B)

```
        . = 0
        LD(R31,X,R0)
        CMOVE(0,R1)
   L:   CMPLTC(R0,0,R2)
        BNE(R2,DONE)
        ADDC(R1,1,R1)
        SHLC(R0,1,R0)
        BR(L)
DONE:   HALT()
X:      LONG(0x08306352)
```

Value left in R0: 0x 83063520

Value left in R1: 0x 4

Value left in R2: 0x 1

Value assembler assigns to symbol X: 0x 20

↑ counts # of left shifts needed until MSB of R0 is 1.

(C)

```
        . = 0
        LD(R31,Z,R1)
        SHRC(R1,26,R1)
   Z:   CMPLTC(R1,0x3C,R2)
        HALT()
```

R1 ← binary for CMPLTC inst.
R1 ← opcode field

Value left in R1: 0x 35 (CMPLTC opcode)

Value left in R2: 0x 1

(D)

```
        . = 0
        LD(R31,X,R0)
        CMOVE(0,R1)
   L:   ADDC(R1,1,R1)
        SHRC(R0,1,R0)
        BNE(R0,L,R2)
        HALT()

        . = 0x100
   X:   LONG(5)
```

R0 ← 5

Value left in R0: 0x 0

Value left in R1: 0x 3

Value left in R2: 0x 14

Value assembler assigns to symbol X: 0x 100

↑ count # of right shifts until R0 == 0.

(E)

```
        . = 0
0       LD(r31, X, r0)
4       CMPLE(r0, r31, r1)
8       BNE(r1, L1, r1)
C       ADDC(r31, 17, r2)
10      BEQ(r31, L2, r31)
14  L1: SRAC(r0, 4, r2)
18  L2: HALT()

        . = 0x1CE8
    X:  LONG(0x87654321)
```

R0 ← 0x87654321 (negative!)

R1 ← 1

R1 ← 0xC, branch taken

R2 ← 0xF8765432

Value left in R0? 0x 87654321

Value left in R1? 0x C

Value left in R2? 0x F8765432

Value assembler assigns to L1: 0x 14

(F)

```
        . = 0
        LD(R31, i, R0)
        SHLC(R0, 2, R0)
        LD(R0, a-4, R1)
        HALT()
    a:  LONG(0xBADBABE)
+4      LONG(0xDEADBEEF)
+8      LONG(0xC0FFEE)
        LONG(0x8BADF00D)
    i:  LONG(3)
```

R0 ← 3

R0 ← 12

R1 ← Mem[12 + a - 4]

= i or 8

Contents of R0 (in hex): 0x C

Contents of R1 (in hex): 0x C0FFEE

(G)

```
        . = 0
0       LD(R31,Z,R1)
4       SHRC(R1,16,R2)
8   Z:  SUBC(R2,0x3C,R3)
C       HALT()
```

R1 ← binary for SUBC

R2 ← top half R1

SUBC 3 2

110001	00011	00010	0x3C
op	R2	R0	

Value left in R1: 0x C4620003C

Value left in R3: 0x C426

Value assembler assigns to symbol Z: 0x 8

(H)

```
        . = 0
0       LD(R31,X,R0)
4       CMOVE(0,R1)

8   L:  ADDC(R1,1,R1)
C       SHRC(R0,1,R0)
10      BNE(R0,L,R2)
14      HALT()

    X:  LONG(0xDECAF)
```

R0 ← DECAF

↑ count # of right shifts to make R0 equal to zero.

Value left in R0: 0x 0

Value left in R1: 0x 14

Value left in R2: 0x 14

6.004 Computation Structures
Spring 2017

Computation Structures

Compilation Worksheet

compile_expr(expr) ⇒ Rx

- Constants: `1234` ⇒ Rx
  - `CMOVE(1234,Rx)`
  - `LD(c1,Rx)`
    …
    `c1: LONG(123456)`
- Variables: `a` ⇒ Rx
  - `LD(a,Rx)`
    `...`
    `a: LONG(0)`
- Variables: `b[expr]` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `MULC(Rx,bsize,Rx)`
    `LD(Rx,b,Rx)`
    …
    `// reserve array space`
    `b: . = . + bsize*blen`
- Operations: $expr_1$ + $expr_2$ ⇒ Rx
  - `compile_expr(`$expr_1$`)⇒Rx`
    `compile_expr(`$expr_2$`)⇒Ry`
    `ADD(Rx,Ry,Rx)`
- Procedure call: `f(`e_1, e_2, …`)` ⇒ Rx
  next lecture!
- Assignment: `a=expr` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `ST(Rx,a)`

compile_statement(…)

- Unconditional: *expr*;
  - `compile_expr(expr)`
- Compound: { *s1; s2;* … }
  - `compile_statement(s1)`
    `compile_statement(s2)`
    …
- Conditional: `if (expr) s1;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lendif)`
    `compile_statement(s1)`
    `Lendif:`
- Conditional: `if (expr) s1; else s2;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lelse)`
    `compile_statement(s1)`
    `BR(Lendif)`
    `Lelse:`
    `compile_statement(s1)`
    `Lendif:`
- Iteration: while (*expr*) *s1*;
  - `BR(Ltest)`
    `Lwhile:`
    `compile_statement(s1)`
    `Ltest:`
    `compile_expr(expr)⇒Rx`
    `BT(Rx, Lwhile)`
- Iteration: `for (init; test; incr) s1;`
  init;
  while (*test*) { *s1; incr;* }

Problem 1.

Please hand-compile the following snippets of C code into equivalent Beta assembly language statements. Assume that memory locations have been allocated for the all C variables with labels that corresponds to the variable names. So to load the value of the C variable `a` into register R3, the appropriate assembly language statement would be `LD(R31,a,R3)`. And to store the value in R17 to the C variable `b`, the appropriate assembly language statement would be `ST(R17,b,R31)`. Similarly, assume that memory locations have been allocated for each C array, with a label defined whose value is the address of the 0th element of the array.

(A)
```
a = 42;
```

```
CMOVE(42,R0)
ST(R0,a,R31)
```

(B)
```
c = 5*x - 13;
```

```
CMOVE(5,R0)
LD(x,R1)
MUL(R0,R1,R0)
CMOVE(13,R1)
SUB(R0,R1,R0)
ST(R0,c)
```

Optimized:

```
LD(x,R1)
MULC(R0,5,R0)
SUBC(R0,13,R0)
ST(R0,c)
```

(C)
```
y = (x - 3)*(y + 123456);
```

```
LD(x,R0)
SUBC(R0,3,R0)
LD(y,R1)
LD(const,R2)
ADD(R1,R2,R1)
MUL(R0,R1,R0)
ST(R0,y)
```

```
// mem locn to hold
// large constant
const: LONG(123456)
```

(D)
```
if (a == 3) b = b + 1;
```

```
LD(R31,a,R0)
CMPEQC(R0,3,R0S)
BF(R0,L1)
LD(R31,b,R0)
ADDC(R0,1,R0)
ST(R0,b,R31)
L1:
```

(E)
```
a[i] = a[i-1];
```

```
LD(i,R0)
MULC(R0,4,R0) // 4 bytes/element
LD(R0,a-4,R1) // sub 4 at assy time!
ST(R1,a,R0)
```

(F)
```
x = y[3] + y[12];
```

```
LD(y+4*3,R0)
LD(y+4*12,R1)
ADD(R0,R1,R0)
ST(R0,x)
```

(G)
```
if (b == 0 || b < min) {
    min = b;
} else {
    too_big += 1;
}
```

```
LD(B,R0)
BEQ(R0,L2)
LD(min,R1)
CMPLT(R0,R1,R1)
BF(R1,L3)
L2:
ST(R0,min)
BR(L4)
L3:
LD(too_big,R0)
ADDC(R0,1,R0)
ST(R0,too_big)
L4:
```

(H)
```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i
    i = i + 1;
}
```

Unoptimized:

```
ST(R31,sum)
ST(R31,i)
L5:
LD(sum,R0)
LD(i,R1)
ADD(R0,R1,R0)
ST(R0,sum)
LD(i,R0)
ADDC(R0,1,R0)
ST(R0,i)
L6:
LD(i,R0)
CMPLTC(R0,10,R1)
BT(R2,L5)
```

Problem 2.

In block-structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```

A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

B. Define a *memory access* as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part H of the preceding problem).

```
Part (H): each iteration
  10 instruction fetches
  6 data memory accesses
10 iterations => 160 accesses
+ 4 from first two insts => 164
```

```
Part (A): each iteration
  4 instruction fetches
  0 data memory accesses
10 iterations => 40 accesses
+ 6 from before/after loop => 46
```

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) {
      sum += i; sum += i+1;
  }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

```
Part (C): each iteration
  6 instruction fetches
  0 data memory accesses
5 iterations => 30 accesses
+ 6 from before/after loop => 36
```

Problem 3.

Which of the following Beta instruction sequences might have resulted from compiling the following C statement? For each sequence describe the value that does end up as the value of y.

```
int x[20], y;
y = x[1] + 4;
```

A.
```
LD (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. If x[0] is stored a location x, x[1] is stored at location x+4 since array element takes one word (4 bytes). Program exception trying to read from a location whose address is not a multiple of 4.

B.
```
CMOVE (4, R0)
ADDC (R0, x + 4, R0)
ST (R0, y, R31)
```

Not this one. The second instruction adds the *address* of x[1] to R0, not the contents of x[1].

C.
```
LD (R31, x + 4, R0)
ST (R0, y + 4, R31)
```

Not this one. This stores x[1] in the location *following* the one word of storage allocated for y.

D.
```
CMOVE (4, R0)
LD (R0, x, R1)
ST (R1, y, R0)
```

Not this one. It's equivalent to (C).

E.
```
LD (R31, x + 4, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Yes!!! This one…

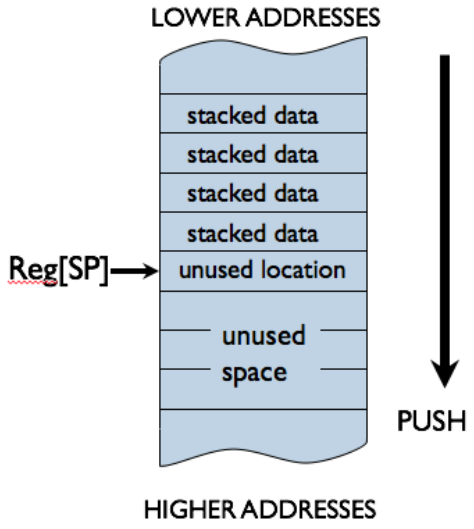
F.
```
ADDC (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. The ADDC instruction loads the address of x plus 1 into R0, then increments it by 4. So y gets the value "x+5" where x is address of the 0^{th} element of the array.

6.004 Computation Structures
Spring 2017

Computation Structures

Procedures & Stacks Worksheet



PUSH(X): Push Reg[x] onto stack

```
ADDC(SP,4,SP)
ST(Rx,-4,SP)
```

POP(X): Pop value at top of stack into Reg[x]

```
LD(SP,-4,RX)
SUBC(SP,4,SP)
```

ALLOCATE(k): Reserve k words of stack

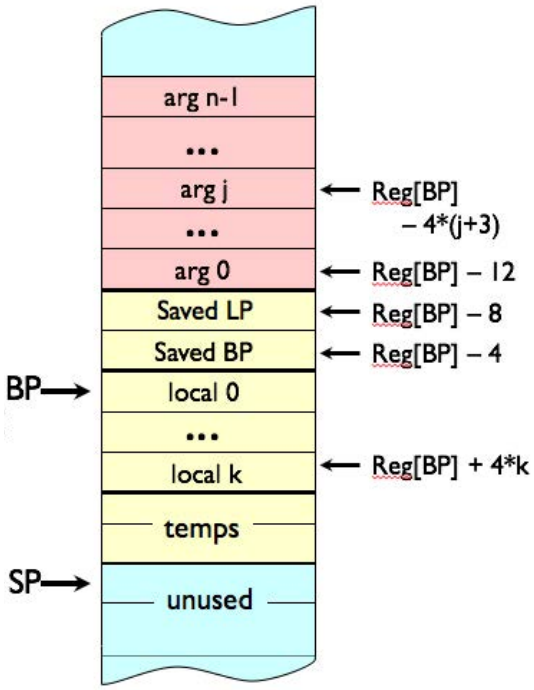
```
ADDC(SP,4*k,SP)
```

DEALLOCATE(k): Release k words of stack

```
SUBC(SP,4*k,SP)
```

Stack discipline: leave stack the way you found it => for every PUSH(), there's a corresponding POP() or DEALLOCATE()

Activation record layout on the stack (aka stack frame):



CALLING SEQUENCE

```
   PUSH(argn)    // push args, last arg first
   …
   PUSH(arg1)
   BR(f, LP)     // call f, return addr in LP
   DEALLOCATE(n) // remove args from stack
```

ENTRY SEQUENCE

```
f: PUSH(LP)      // save return addr
   PUSH(BP)      // save old frame pointer
   MOVE(SP,BP)   // initialize new frame pointer
   ALLOCATE(nlocals)  // make room for locals
   (push other regs)  // preserve old reg vals
```

EXIT SEQUENCE

```
   // return value in R0
   MOVE(BP,SP)   // remove locals
   POP(BP)       // restore old frame pointer
   POP(LP)       // recover return address
   JMP(LP)       // resume execution in caller
```

Problem 1.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int fn(int x) {
  int lowbit = x & 1;
  int rest = x >> 1;
  if (x == 0) return 0;
  else return ???;
}
```

```
fn: PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(2)
    PUSH(R1)
    LD(BP,-12,R0)
    ANDC(R0,1,R1)
xx: ST(R1,0,BP)
    SHRC(R0,1,R1)
    ST(R1,4,BP)
yy: BEQ(R0,rtn)
    LD(BP,4,R1)
    PUSH(R1)
    BR(fn,LP)
    DEALLOCATE(1)
    LD(BP,0,R1)
    ADD(R1,R0,R0)
rtn:POP(R1)
zz: MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: fn(rest) + lowbit

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure (circle one)?

Work the same? YES ... (NO)

The MOVE(BP,SP) is deallocating the local variables lowbit and rest

(C) In the space below, fill in the binary representation for the instruction stored at the location tagged '**xx:**' in the above program.

R27

0 1 1 0 0 1	0 0 0 0 1	1 1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ST	R1	BP	0

(fill in missing 1s and 0s for instruction at xx:)

The procedure **fn** is called from an external procedure and its execution is interrupted just prior to the execution of the instruction tagged '**yy:**'. The contents of a region of memory are shown on the left below.

NB: All addresses and data values are shown in hex. The contents of **BP** are 0x1C8 and **SP** contains 0x1D4.

(D) What was the argument to the most recent call to **fn**?

```
184:   4
188:   7
18C:  47   X
190:  C4   saved LP
194: 170   saved BP
198:   1   lowbit
19C:  23   rest
1A0:  22   saved R1
1A4:  23   X
1A8:  4C   saved LP
1AC: 198   saved BP
1B0:   1   lowbit
1B4:  11   rest
1B8:  23   saved R1
1BC:  11   X
1C0:  4C   saved LP
1C4: 1B0   saved BP
1C8:   1   ←BP  lowbit
1CC:   8   rest
1D0: ???   saved R1
1D4:   0   ←SP
```

from current stack frame **Most recent argument (HEX): x=** 11

(E) What is the missing value marked **???** for the contents of location 1D0?

looking at code: R1 held value of argument before the call

Contents of 1D0 (HEX): 11

(F) What is the hex address of the instruction tagged **rtn:?**

saved LP is the address of mem location holding DEALLOCATE inst.

Address of rtn (HEX): 58

(G) What was the argument to the *original* call to **fn**?

find frame with saved LP ≠ 0x4C.

Original argument (HEX): x= 47

(H) What is the hex address of the BR instruction that called **fn** *originally*?

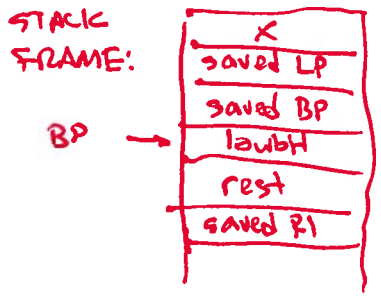
(saved LP on oldest stack frame) − 4.

Address of original call (HEX): C0

(I) What were the contents of R1 at the time of the *original* call?

Original R1 contents (HEX): 22

(J) What value will be returned to the *original* caller?



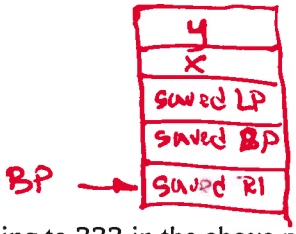
Return value for original call (HEX): 4

fn counts number of 1 bits in argument.

Problem 2.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int f(int x, int y) {
  x = (x >> 1) + y;
  if (y == 0) return x;
  else return ???;
}
```




```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP,BP)
      PUSH(R1)
      LD(BP,-12,R0)
      SHRC(R0,1,R0)
      LD(BP,-16,R1)
      ADD(R0,R1,R0)
      BEQ(R1,rtn)
      SUBC(R1,1,R1)
      PUSH(R1)
      PUSH(R0)
      BR(f,LP)
      DEALLOCATE(2)
rtn:  POP(R1)
zz:   MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: f(x, y-1)

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure?

Work the same (circle one)? (YES) ... NO

The procedure **f** is called from an external procedure and then execution is stopped just prior to one of the executions of the instruction labeled '**rtn:**'. The addresses and contents of a region of memory are shown in the table on the right; all addresses and data values in the table are in hex. When execution is stopped **BP contains the value 0x14C and SP contains the value 0x150.**

(C) What are the arguments to the **currently active call** to **f**?

Most recent arguments (in hex): x = 0x 6 , y = 0x 1

(D) If you can tell from the information provided, specify the arguments to the **original** call to **f**, otherwise select **CAN'T TELL.**

Original arguments (in hex) : x = 0x A , y = 0x 3 , **or CAN'T TELL**

(E) What is the missing value in location 0x12C?

x = (0xA >> 1) + y
  = 5   3

Contents of location 0x12C (in hex): 0x 8

(F) What is the hex address of the instruction labeled **rtn:**?

Address of instruction labeled rtn: (in hex): 0x 34C

(G) What is the hex address of the BR instruction that called **f** *originally*?

Address of original call (in hex): 0x 2C0 , **or CAN'T TELL**

(H) What value will be returned to the *original* caller?

Return value for original call (in hex): 0x 2

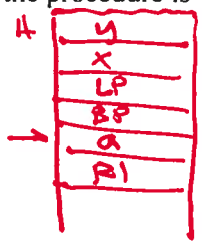
f(A,3) → f(8,2) → f(6,1) → f(9,0) → returns 2

108	7	
10C	320	
110	104	
114	3	y
118	A	x
11C	2C4	LP
120	104	BP
124	3	R1
128	2	y
12C	8	x
130	348	LP
134	124	BP
138	2	R1
13C	1	y
140	6	x
144	348	LP
148	138	BP
BP→14C	1	R1
SP→150	0	
154	4	
158	348	
15C	14C	
160	0	

Problem 3.

The following C program implements a function H(x,y) of two arguments, which returns an integer result. The assembly code for the procedure is shown on the right.

```
int H(int x, int y) {
    int a = x - y;
    if (a < 0) return x;
    else return ???;
}
```




```
H:    PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      ALLOCATE(1)
      PUSH(R1)

      LD(BP,-12,R0)
      LD(BP,-16,R1)
      SUB(R0,R1,R1)
      ST(R1,0,BP)

      CMPLTC(R1,0,R1)
      BT(R1,rtn)

      LD(BP,-16,R1)
      PUSH(R1)
      LD(BP,0,R0)
      PUSH(R0)
      BR(H,LP)
      DEALLOCATE(2)

rtn:  POP(R1)
      MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

The execution of the procedure call **H(0x68,0x20)** has been suspended just as the Beta is about to execute the instruction labeled "rtn:" during one of the recursive calls to H. A *partial* trace of the stack at the time execution was suspended is shown to the right below.

(A) Examining the assembly language for H, what is the appropriate C code for ??? in the C representation for H?

C code for ???: H(a,y)

(B) Please fill in the values for the blank locations in the stack dump shown on the right. Express the values in hex or write "---" if value can't be determined. Hint: Figure out the layout of H's activation record and use it to identify and label the stack frames in the stack dump.

Fill in the blank locations with values (in hex!) or "---"

	0x0024	LP	
	0x0070	BP	
	0x0048	a	
	0x0068	R1	
	0x20	y	
	0x68	x	
	0x7C	LP	
	0x80	BP	
C8	0x28	a	
CC	0x0020	R1	
D0	0x0020	y	
D4	0x0028	x	
D8	0x007C	LP	
DC	0x00C8	BP	
E0	BP→ 0x0008	a	
E4	0x0020	R1	
SP→ E8	0x0020		

(C) Determine the specified values at the time execution was suspended. Please express each value in hex or write "CAN'T TELL" if the value cannot be determined.

result of H(8,20) **Value in R0 or "CANT TELL": 0x** 8

y **Value in R1 or "CANT TELL": 0x** 20

Value in BP or "CANT TELL": 0x E0

Value in LP or "CANT TELL": 0x 7C

Value in SP or "CANT TELL": 0x E8

Problem 4.

The following C program computes the log base 2 of its argument. The assembly code for the procedure is shown on the right, along with a stack trace showing the execution of ilog2(10). The execution has been halted just as it's about to execute the instruction labeled "rtn:"

```
/* compute log base 2 of arg */
int ilog2(unsigned x) {
    unsigned y;
    if (x == 0) return 0;
    else {
        /* shift x right by 1 bit */
        y = x >> 1;
        return ilog2(y) + 1;
    }
}
```


```
ilog2: PUSH(LP)
       PUSH(BP)
       MOVE(SP,BP)
       ALLOCATE(1)
       PUSH(R1)

       LD(BP,-12,R0)
       BEQ(R0,rtn,R31)

       LD(BP,-12,R1)
       SHRC(R1,1,R1)
       ST(R1,0,BP)

       LD(BP,0,R1)
       PUSH(R1)
       BR(ilog2,LP)
1A8    DEALLOCATE(1)
1AC    ADDC(R0,1,R0)
1B0,1B4
rtn:   POP(R1)        <-- STOP
1B8 xxx: DEALLOCATE(1)
       MOVE(BP,SP)
       POP(BP)
       POP(LP)
       JMP(LP)
```

(A) What are the values in R0, SP, BP and LP at the time execution was halted? Please express the values in hex or write "CAN'T TELL".

Value in R0: 0x 1 = ilog2(0)+1 **in SP: 0x** 24C

Value in BP: 0x 244 **in LP: 0x** 1A8

(B) Please fill in the values for the five blank locations in the stack trace shown on the right. Please express the values in hex.

Fill in values (in hex!) for 5 blank locations

(C) In the assembly language code for ilog2 there is the instruction "LD(BP,-12,R0)". If this instruction were rewritten as "LD(SP,NNN,R0)" what is correct value to use for NNN?

Correct value for NNN: -20

(D) In the assembly language code for ilog2, what is the address of the memory location labeled "xxx:"? Please express the value in hex.

Address of location labeled "xxx:": 0x 1B8

Values are in hex!

	5	x
	1A8	LP
	208	BP
	2	y
	5	R1
	2	x
	1A8	LP
	21C	BP
230	1	y
234	2	R1
238	1	x
23C	1A8	LP
240	230	BP
244 BP→	0	y
248	1	R1
24C SP→	0	

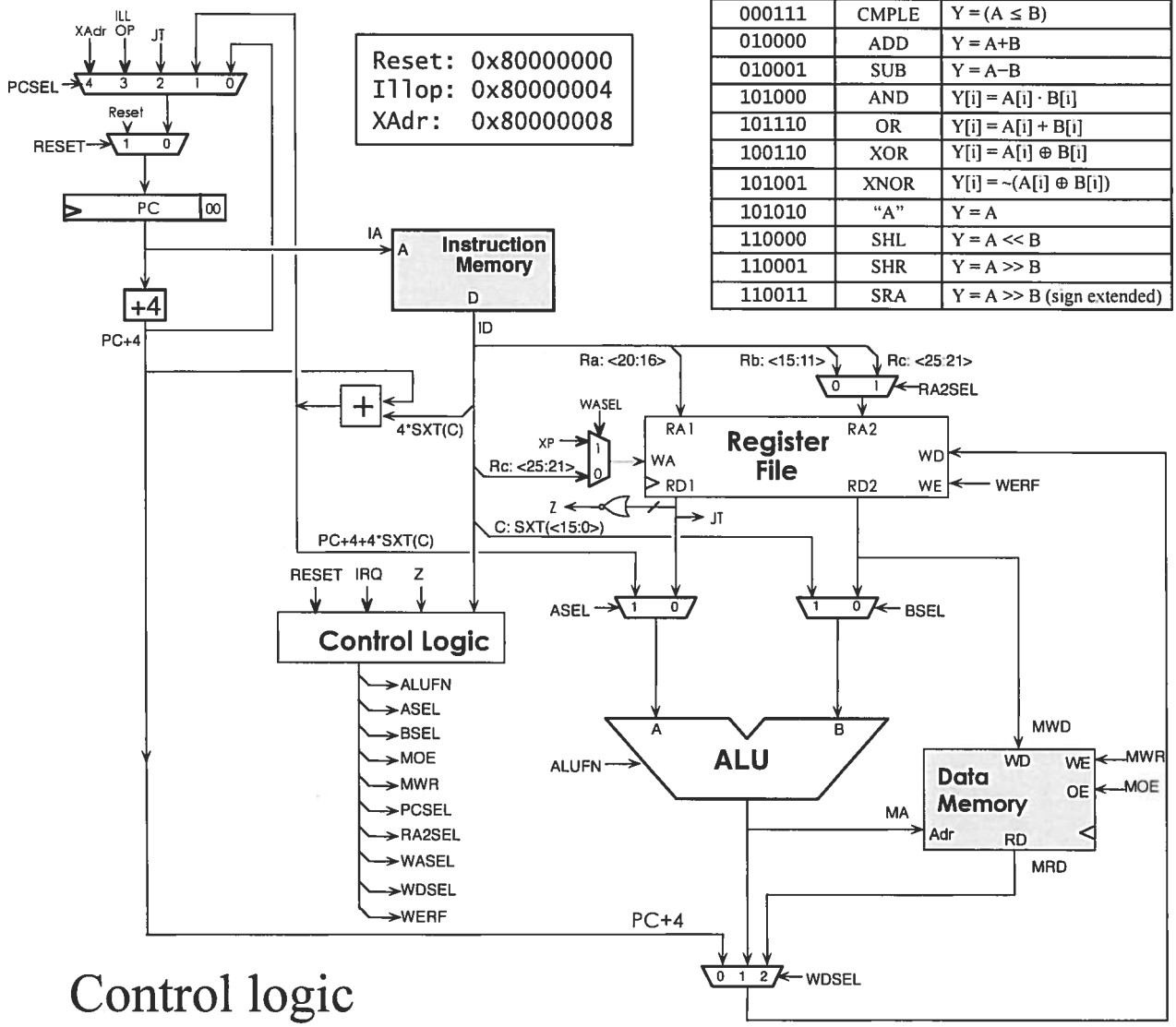
6.004 Computation Structures
Spring 2017

Computation Structures

Beta Implementation Worksheet

Unpipelined Beta

ALUFN[5:0]	Operation	Output value Y[31:0]
000011	CMPEQ	Y = (A = B)
000101	CMPLT	Y = (A < B)
000111	CMPLE	Y = (A ≤ B)
010000	ADD	Y = A+B
010001	SUB	Y = A−B
101000	AND	Y[i] = A[i] · B[i]
101110	OR	Y[i] = A[i] + B[i]
100110	XOR	Y[i] = A[i] ⊕ B[i]
101001	XNOR	Y[i] = ~(A[i] ⊕ B[i])
101010	"A"	Y = A
110000	SHL	Y = A << B
110001	SHR	Y = A >> B
110011	SRA	Y = A >> B (sign extended)



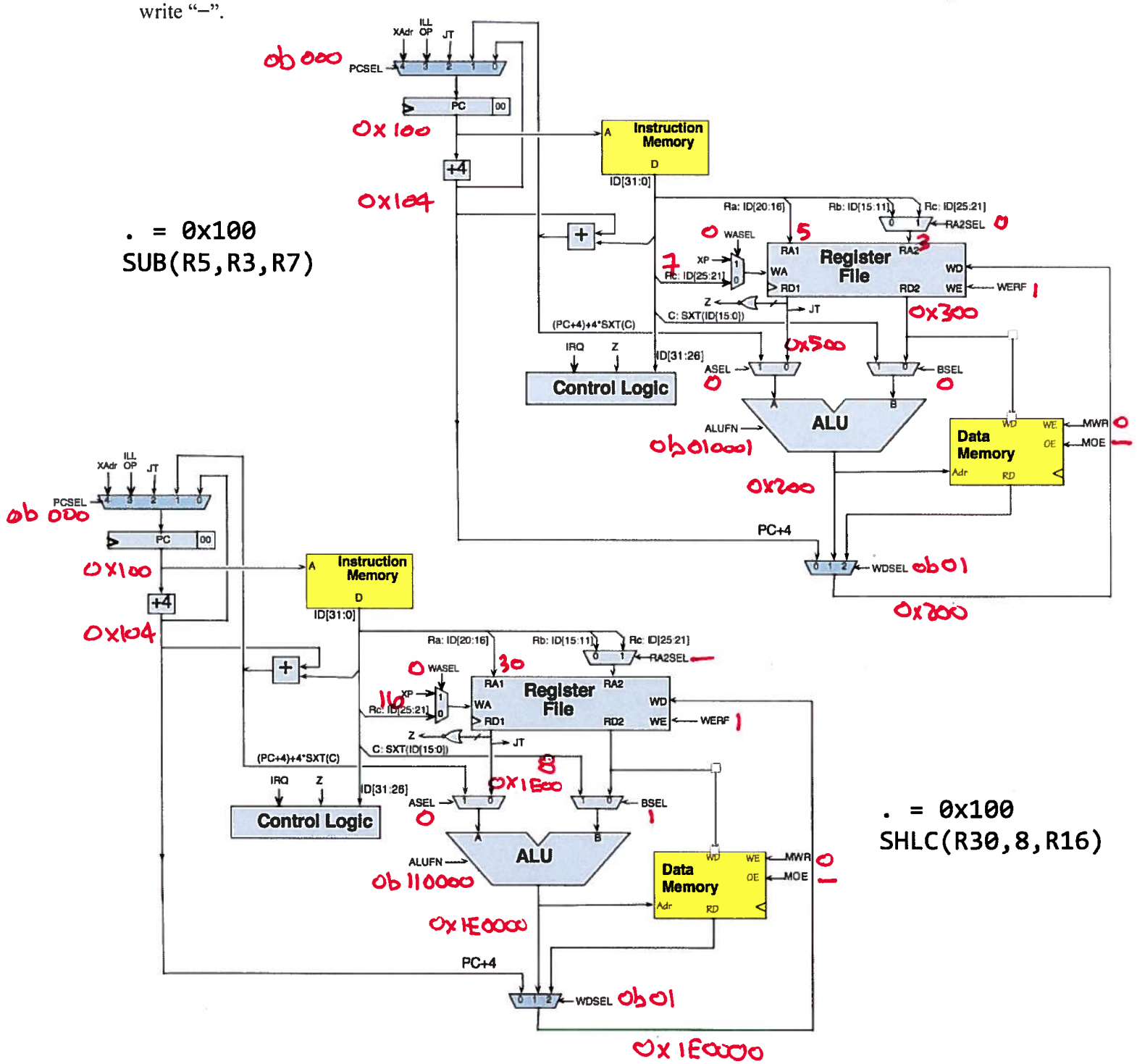
Control logic

	RESET	IRQ	OP	OPC	LD	LDR	ST	JMP	BEQ	BNE	ILLOP
ALUFN[5:0]	--	--	F(op)	F(op)	"+"	"A"	"+"	--	--	--	--
ASEL	--	--	0	0	0	1	0	--	--	--	--
BSEL	--	--	0	1	1	--	1	--	--	--	--
MOE	--	--	--	--	1	1	0	--	--	--	--
MWR	0	0	0	0	0	0	1	0	0	0	0
PCSEL[2:0]	--	4	0	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	3
RA2SEL	--	--	0	--	--	--	1	--	--	--	--
WASEL	--	1	0	0	0	0	--	0	0	0	1
WDSEL[1:0]	--	0	1	1	2	2	--	0	0	0	0
WERF	--	1	1	1	1	1	0	1	1	1	1

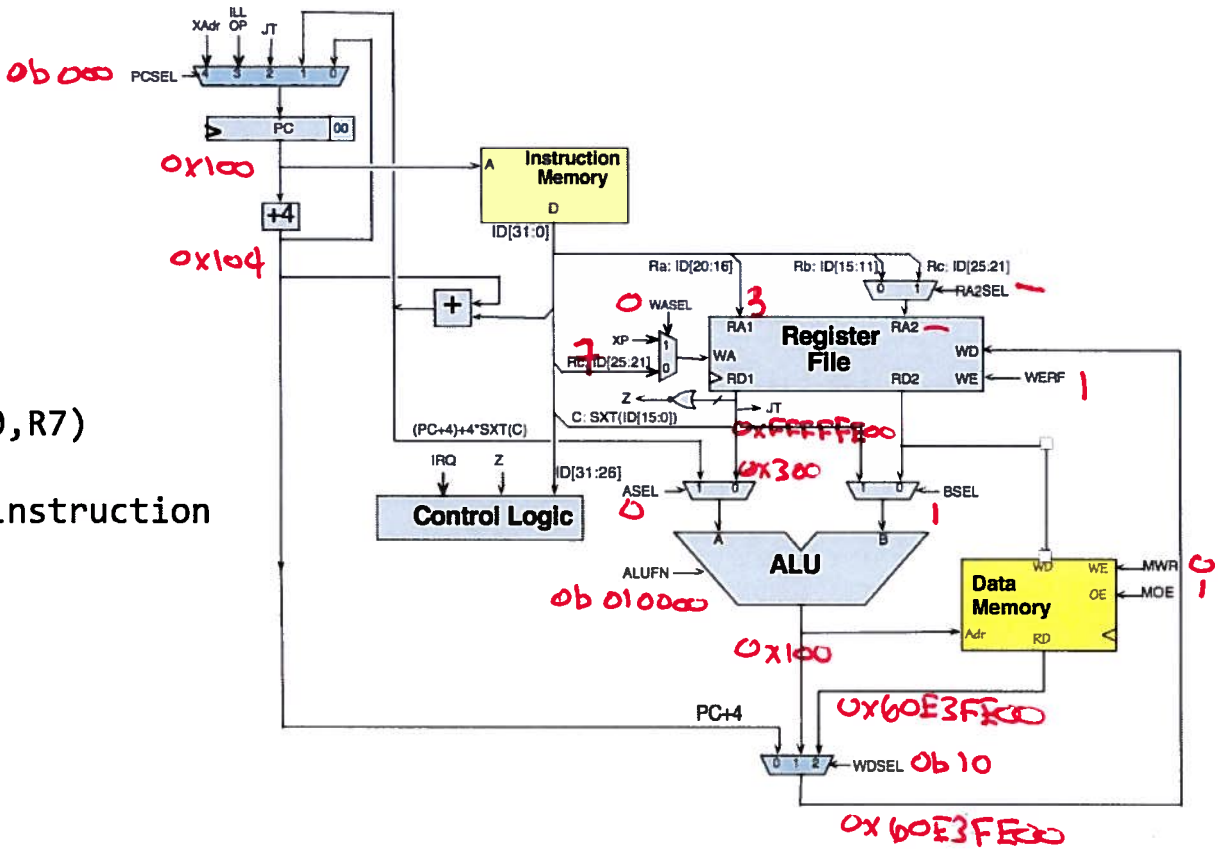
Problem 1.

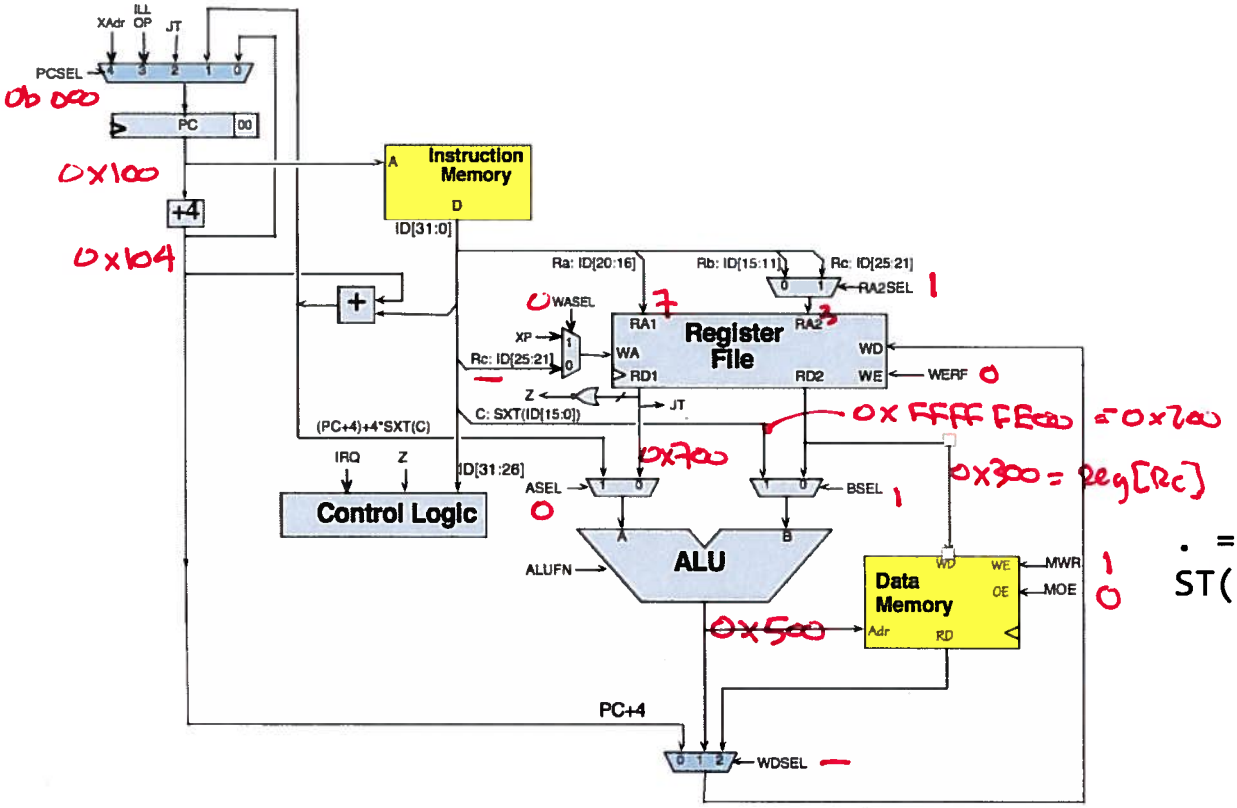
For this problem assume that each register has been initialized to the value 0x0000??00 where "??" is the register number as a two-digit hex number. So R0 is initialized to 0x00000000, R1 to 0x00000100, …, and R30 to 0x00001E00. R31 of course always reads as 0.

For each instruction below, please indicate the values that will be found in the unpipelined Beta datapath just before the end of the clock cycle in which the instruction is executed. If the value doesn't matter since it's not used during the execution of the instruction or can't be determined, write "–".

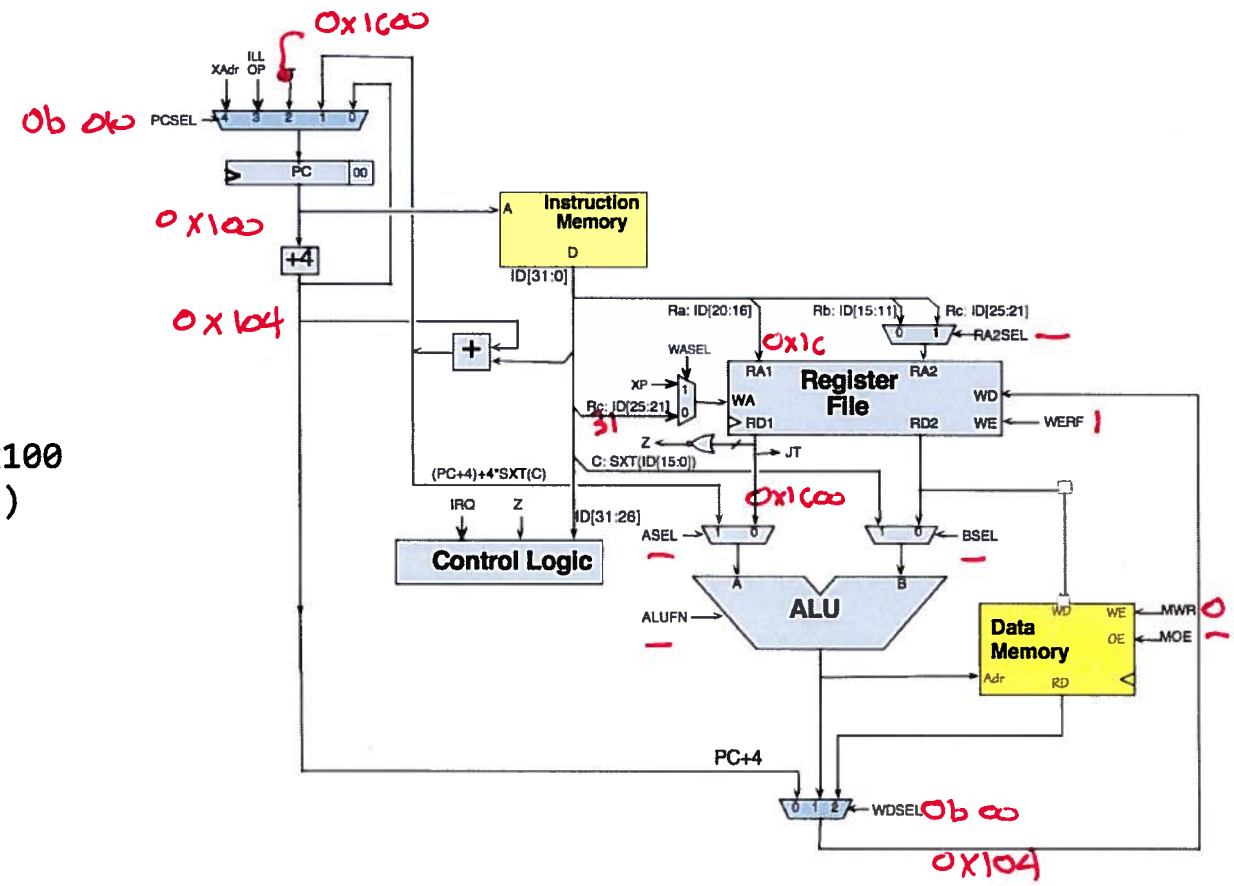


```
. = 0x100
LD(R3,-0x200,R7)

// hex for instruction
0x60E3FE00
```

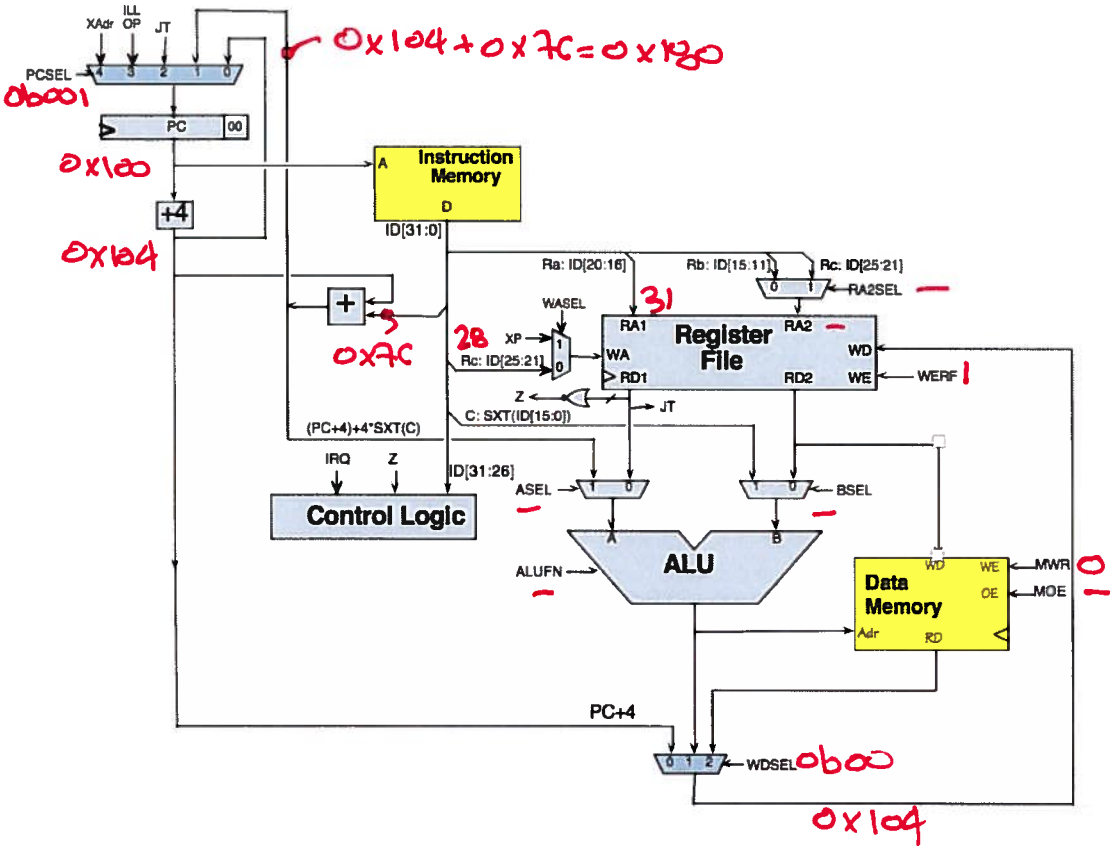





```
. = 0x100
ST(R3,-0x200,R7)
```



. = 0x100
JMP(LP)



. = 0x100
BEQ(R31,.+0x80,LP)

.+0x80 = 0x180

literal:

$$\frac{(0x180 - 0x104)}{4} = 0x1F$$

4*literal = 0x7C

Problem 2.

Consider adding the following instructions to the Beta instruction set, for implementation on the Beta hardware shown in lecture (see diagram included in the reference material at the end of this quiz). You're allowed to change how the control signals are generated but no modifications to the datapath are permitted.

For each instruction either fill in the appropriate values for the control signals in the table below or **put a line through the whole row if the instruction cannot be implemented** using the existing Beta datapath. Use "—" to indicate a "don't care" value for a control signal. The values can be a function of Z (which is 1 when Reg[Ra] is zero).

LDX(Ra, Rb, Rc) // Load indexed
EA ← Reg[Ra] + Reg[Rb]
Reg[Rc] ← Mem[EA]
PC ← PC + 4

like LD with BSEL=0

STX(Ra, Rb, Rc) // Store indexed
EA ← Reg[Ra] + Reg[Rb]
Mem[EA] ← Reg[Rc]
PC ← PC + 4

} need to read 3 regs!

MVZC(Ra, literal, Rc) // Move constant if zero
If Reg[Ra] == 0 then Reg[Rc] ← SXT(literal)
PC ← PC + 4

SOB(Ra, literal, Rc) // Subtract one and branch
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
tmp ← Reg[Ra]
Reg[Rc] ← Reg[Ra] – 1 ← not possible with our ALU
if tmp != 0 then PC ← EA

ARA(Ra, literal, Rc) // Add Relative Address
Reg[Rc] ← Reg[Rc] + PC + 4 + 4*SEXT(literal)
PC ← PC + 4

(FILL IN TABLE BELOW)

Instr	ALUFN	WERF	BSEL	WDSEL	MOE	MWR	RA2SEL	PCSEL	ASEL	WASEL
LDX	"+"	1	0	2	1	0	0	0	0	0
STX	—	—	—	—	—	—	—	—	—	—
MVZC	"B"	Z	1	1	?	0	?	0	?	0
SOB	—	—	—	—	—	—	—	—	—	—
ARA	"+"	1	0	1	?	0	1	0	1	0

Problem 3.

Ben Bitdiddle is proposing the short assembly language program shown to the right as a manufacturing test to ensure the correct operation of the Control ROM. He is assuming – and you may too – that the Beta datapath components (e.g., Memories, ALU, muxes, register file, adders) are working correctly and that any errors in execution are due to faulty signals from the Control ROM. Ben's plan is to run the program then look at the value in the memory location labeled ANS. If the value is 0x6004, the test passes, otherwise the Beta being tested is declared faulty and discarded.

```
        .=0
Test:   LD(R31,X,R0)
        ADDC(R0,1,R1)
        BNE(R1,L1,R31)
        ADDC(R1,1,R1)
L1:     ST(R1,ANS,R31)
        HALT()
X:      .LONG(0x6003)
ANS:    .LONG(0)
```

For each of the following faults, indicate the value that the faulty Beta will store into ANS.

(A) RA2SEL is stuck at the value 0.

Rb chosen during ST. inst[15:11] == 0, so ST writes Reg[Rb]

Value stored in ANS by faulty Beta: 0x6003

(B) WDSEL[1:0] is stuck at the binary value 00.

PC+4 chosen as value to store into Rc

Value stored in ANS by faulty Beta: 0x8

(C) PCSEL[2:0] is stuck at the binary value 000.

branches never taken, so second ADDC is executed

Value stored in ANS by faulty Beta: 0x6005

Problem 4. Beta Implementation

Consider the assembly language program shown to the right. Assume that all register values are initialized to 0, execution starts at PC=0 and halts when HALT() is executed.

This program is run on 4 different broken Betas, where each Beta has a specified control signal stuck at the specified value, i.e., the control signal value is fixed and is not affected by the value produced by the Beta's CTL module. For each broken Beta, please give the value in registers R1, R2, R3, and the location X: after the programs halts. **Assume that any don't care control signal values are 0.**

```
        . = 0
        LD(R31,X,R1)
        CMPLTC(R1,0,R2)
        BF(R2,end,R3)
        SUB(R31,R1,R1)
        ST(R1,X,R31)
END:    HALT()
X:      LONG(-42)
```

Broken control signal	*Final value in*			
	R1	*R2*	*R3*	*Location X:*
RA2SEL stuck at 0	42	1	0xC	0
WDSEL stuck at 0b00	0x10	8	0xC	0x10
WASEL stuck at 1	0	0	0	-42
WERF stuck at 1	0x14	1	0xC	42

select RB, not RC

contents of R0

Rc = PC+4

only write to R31

change RC during ST

Problem 5.

In this problem, you will consider a number of plausible hardware faults in an otherwise working Beta processor; you may want to consult the diagram and documentation on the backs of pages of this quiz. Each of the faults involves changing a particular output of the control logic to some new (incorrect) constant value. In each case, you are to evaluate the impact of the fault on each of the following Beta instructions:

```
I1:     ST(R0, 0x100, R1)
I2:     JMP(LP, R31)
I3:     BEQ(R31, .+4, R0)
I4:     SUB(R1, R0, R0)
```

For each of the following faults, identify which (if any) of the above instructions will fail to work properly – that is, if the fault might effect the processor state (register and PC values) after the execution of the instruction. Be careful: some of the these are tricky!

(A) ALUFN stuck at code for "-" (32-bit SUBTRACT)

Which instruction(s) fail? Circle all applicable, or NONE: (I1) I2 I3 I4 NONE

(B) RA2SEL stuck at 1 note for I4: Rb==Rc, so RA2SEL doesn't matter

Which instruction(s) fail? Circle all applicable, or NONE: I1 I2 I3 I4 (NONE)

(C) WERF stuck at 0

Which instruction(s) fail? Circle all applicable, or NONE: I1 I2 (I3) (I4) NONE

(D) BSEL stuck at 0

Which instruction(s) fail? Circle all applicable, or NONE: (I1) I2 I3 I4 NONE

Problem 6.

(A) The Beta executes the assembly program below starting at location 0 and stopping when it reaches the HALT() instruction. Please give the values in the indicated registers after the Beta stops. Write the values in hex or write "CAN'T TELL" if the values cannot be determined.

```
addr
 0      . = 0
 4      LD(r31, X, r0)
 8      CMPLE(r0, r31, r1)
 C      BNE(r1, L1, r2)
10      ADDC(r31, 1, r0)
10 L1:  HALT()

14 X:   LONG(0x87654321)
```

Value left in R0 or "CAN'T TELL": 0x 87654321

Value left in R1 or "CAN'T TELL": 0x 1

Value left in R2 or "CAN'T TELL": 0x C

(B) Redo part (A) but this time assume that all the control signals going to the datapath from the control logic are stuck at logic 0, *except for WERF* which operates as expected. Note that when ALUFN[4:0] = 0b00000, the ALU computes A+B.

PCSEL==0 ⇒ branch not taken
WDSEL=0 ⇒ PC+4 always written to Rc

```
addr
 0       . = 0
 4       LD(r31, X, r0)
 8       CMPLE(r0, r31, r1)
 C       BNE(r1, L1, r2)
10       ADDC(r31, 1, r0)
10  L1:  HALT()

14  X:   LONG(0x87654321)
```

Value left in R0 or "CAN'T TELL": 0x 10

Value left in R1 or "CAN'T TELL": 0x 8

Value left in R2 or "CAN'T TELL": 0x C

(C) Bettah Beta Inc. (you can tell they're based in Boston!) is proposing a new Beta instruction TCLR that sets Rc to the current value of a memory location whose address is in Ra and writes a zero to that location, all in a single cycle. They are assuming that main memory works as it does in JSim: its read ports are combinational and the write port takes a CLK signal and performs the write at the end of the current cycle – *so the same memory location can be read and written in the same clock cycle*.

Here's their draft entry for the Beta reference manual:

Usage: TCLR(Ra,Rc)

Opcode:

011010	Rc	Ra	11111	*unused*

Operation:

PC ← PC + 4
EA ← Reg[Ra]
Reg[Rc] ← Mem[EA]
Mem[EA] ← 0

The contents of register Rc are set to the contents of the memory location whose address is in Ra. Then, at the end of the cycle, that memory location is set to 0.

Please fill in the appropriate values for the control signals that will cause the datapath to implement the correct operations OR briefly explain why TCLR cannot be implemented with the existing Beta datapath in a single cycle.

Fill in table:

Instr	ALUFN	WERF	BSEL	WDSEL	MWR	RA2SEL	PCSEL	ASEL	WASEL
TCLR	"A"	1	–	2	1	0	0	0	0

ASEL=0, ALUFN="A" ⇒ memory address in Reg[RA]

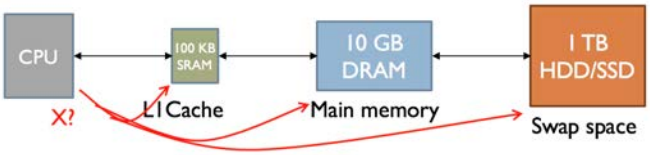
WDSEL=2, WERF=1, WASEL=0 ⇒ regfile written with Mem[Reg[Ra]]

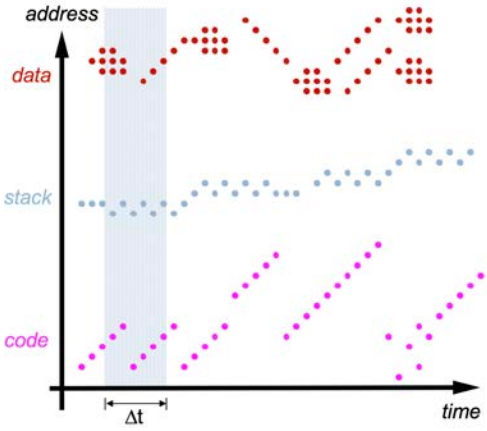
RA2SEL=0, WR=1 ⇒ memory write data is Reg[Rb]=Reg[31]=0 and main memory will write

6.004 Computation Structures
Spring 2017

Computation Structures

Memory Hierarchy & Caches Worksheet

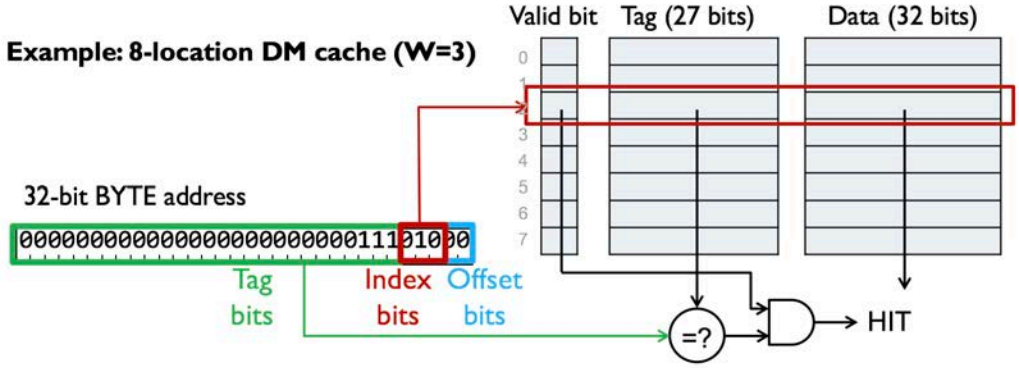




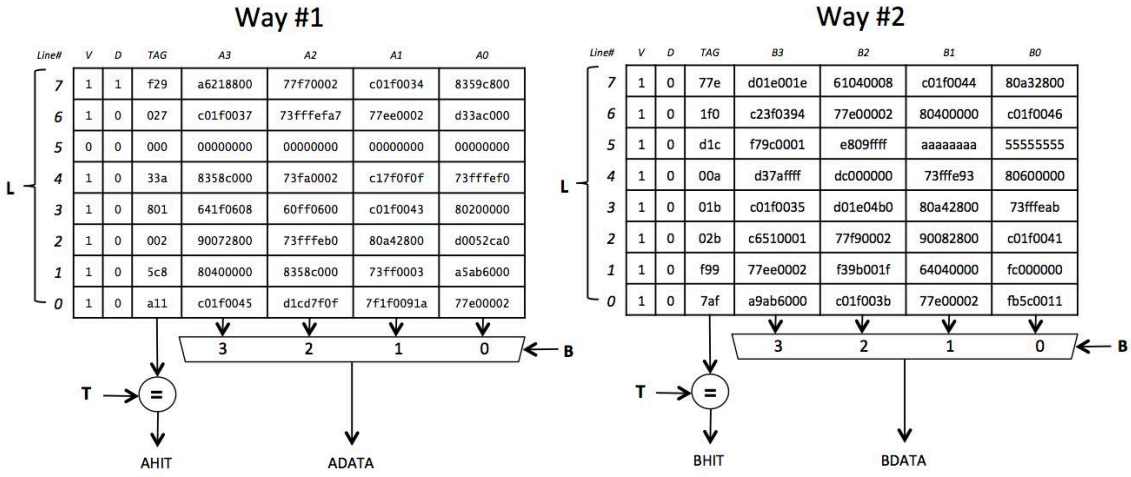
Keep the most often-used data in a small, fast SRAM (often local to CPU chip). The reason this strategy works: LOCALITY.

Locality of reference: Access to address X at time t implies that access to address X+ΔX at time t+Δt becomes more probable as ΔX and Δt approach zero.

AMAT = HitTime + MissRatio * MissPenalty



Example: 2-way set-associative cache, 8 sets, 4-word block size, write-back



Line#	V	D	TAG	A3	A2	A1	A0
7	1	1	f29	a6218800	77f70002	c01f0034	8359c800
6	1	0	027	c01f0037	73fffefa7	77ee0002	d33ac000
5	0	0	000	00000000	00000000	00000000	00000000
4	1	0	33a	8358c000	73fa0002	c17f0f0f	73fffef0
3	1	0	801	641f0608	60ff0600	c01f0043	80200000
2	1	0	002	90072800	73fffeb0	80a42800	d0052ca0
1	1	0	5c8	80400000	8358c000	73ff0003	a5ab6000
0	1	0	a11	c01f0045	d1cd7f0f	7f1f0091a	77e00002

Line#	V	D	TAG	B3	B2	B1	B0
7	1	0	77e	d01e001e	61040008	c01f0044	80a32800
6	1	0	1f0	c23f0394	77e00002	80400000	c01f0046
5	1	0	d1c	f79c0001	e809ffff	aaaaaaaa	55555555
4	1	0	00a	d37affff	dc000000	73fffe93	80600000
3	1	0	01b	c01f0035	d01e04b0	80a42800	73fffeab
2	1	0	02b	c6510001	77f90002	90082800	c01f0041
1	1	0	f99	77ee0002	f39b001f	64040000	fc000000
0	1	0	7af	a9ab6000	c01f003b	77e00002	fb5c0011

Replacement strategy choices: least-recently used (LRU); first in, first out (FIFO); random
Write-policy choices: write-through, write-behind, write-back

Problem 1.

(A) The timing for a particular cache is as follows: checking the cache takes 1 cycle. If there's a hit the data is returned to the CPU at the end of the first cycle. If there's a miss, it takes 10 *additional* cycles to retrieve the word from main memory, store it in the cache, and return it to the CPU. If we want an average memory access time of 1.4 cycles, what is the minimum possible value for the cache's hit ratio?

$1.4 = 1 + (1-\alpha)10$

$\therefore 0.4 = 1-\alpha$

Minimum possible value of hit ratio: 0.96

(B) If the cache block size, i.e., words/cache line, is doubled but the total number of data words in the cache is unchanged, how will the following cache parameters change? Please circle the best answer.

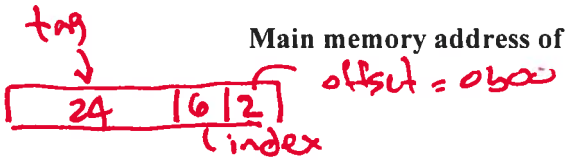
# of offset bits: UNCHANGED ... **+1** ... -1 ... 2x ... 0.5x ... CAN'T TELL

# of tag bits: **UNCHANGED** ... +1 ... -1 ... 2x ... 0.5x ... CAN'T TELL

# of cache lines: UNCHANGED ... +1 ... -1 ... 2x ... **0.5x** ... CAN'T TELL

Consider a direct-mapped cache with 64 total data words with 1 word/cache line, which uses a LRU replacement strategy and a write-back write strategy. This cache architecture is used for parts (C) through (F).

(C) If cache line number 5 is valid and its tag field has the value 0x1234, what is the address in main memory of the data word currently residing in cache line 5?



Main memory address of data word in cache line 5: 0x 0x123414

The program shown on the right repeatedly executes an inner loop that sums the 16 elements of an array that is stored starting in location 0x310.

The program is executed for many iterations, then a measurement of the cache statistics is made during one iteration through all the code, i.e., starting with the execution of the instruction labeled `outer_loop:` until just before the next time that instruction is executed.

```
  . = 0
outer_loop:
  CMOVE(16,R0)    // initialize loop index J
  CMOVE(0,R1)

loop:             // add up elements in array
  SUBC(R0,1,R0)   // decrement index
  MULC(R0,4,R2)   // convert to byte offset
  LD(R2,0x310,R3)// load value from A[J]
  ADD(R3,R1,R1)   // add to sum
  BNE(R0,loop)    // loop until all words are summed

  BR(outer_loop) // perform test again!
```

(D) In total, how many instruction fetches occur during one complete iteration of the outer loop? How many data reads?

ifetch = 2 + (5*16) + 1

dread = 1*16

Number of instruction fetches: 83

Number of data reads: 16

(E) How many instruction fetch misses occur during one complete iteration of the outer loop? How many data read misses? Hint: remember that the array starts at address 0x310.

4 locations in array conflict w/ 4 instructions
⇒ other insts. in cache
⇒ other data in cache

Number of instruction fetch misses: 4

Number of data read misses: 4

(F) What is the hit ratio measured after one complete iteration of the outer loop?

total accesses = 83 + 16 = 99

total hits = 99 - 8

Hit ratio: $\frac{99-8}{99} = \frac{91}{99}$

Problem 2.

The Beta Engineering Team is working on the design of a cache. They've decided that the cache will have **a total of 2^{10} = 1024 data words**, but are still thinking about the other aspects of the cache architecture.

First assume the team chooses to build a direct-mapped write-back cache with a block size of 4 words. ⇒ 4 bit offset

(A) Please answer the following questions:

$\frac{1024 \text{ words}}{4 \text{ wds/line}} = 256$ lines

Number of lines in the cache: 256

Number of bits in the tag field for each cache entry: 20

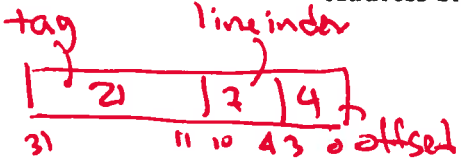
(B) This cache takes *2 clock cycles* to determine if a memory access is a hit or a miss and, if it's a hit, return data to the Beta. If the access is a miss, the cache takes *20 additional clock cycles* to fill the cache line and return the requested word to the Beta. If the hit rate is 90%, what is the Beta's average memory access time in clock cycles?

Average memory access time assuming 90% hit rate (clock cycles): 4

AMAT = 2 + (1 - .9)(20) = 2 + 2 = 4

Now assume the team chooses to build a 2-way set-associative write-back cache with a block size of 4 words. *The total number of data words in the entire cache is still 1024.* The cache uses a LRU replacement strategy. ⇒ 128 lines in each way (7 bits of index)

(C) Please answer the following questions:



Address bits used as offset (including byte offset): A[3 : 0]

Address bits used as cache line index: A[10 : 4]

Address bits used for tag comparison: A[31 : 11]

(D) To implement the LRU replacement strategy this cache requires some additional state for each set. How many state bits are required for each set?

Number of state bits needed for each set for LRU: 1

To test this set-associative cache, the team runs the benchmark code shown on the right. The code sums the elements of a 16-element array. The first instruction of the code is at location 0x0 and the first element of the array is at location 0x10000. Assume that the cache is empty when execution starts and remember *the cache has a block size of 4 words.*

```
. = 0x0
  CMOVE(0,R0)
  CMOVE(0,R1)
L: LD(R0,A,R2)
  ADD(R2,R1,R1)
  ADDC(R0,4,R0)
  CMPLTC(R0,64,R2)
  BT(R2,L)
  HALT()

. = 0x10000
A: LONG(1)
  LONG(2)
  ...
  LONG(15)
  LONG(16)
```

(E) How many instruction misses will occur when running the benchmark?

↳ 2 cache lines

Number of instruction misses when running the benchmark: 2

(F) How many data misses (i.e., misses caused by the memory access from the LD instruction) will occur when running the benchmark?

Number of data misses when running the benchmark: 4

16-element array ⇒ 4 cache lines

(g) What's the exact hit rate when the complete benchmark is executed?

Benchmark hit rate: $\frac{99-6}{99} = \frac{93}{99} \simeq 94\%$

• # inst. fetches = 2 + 16*5 + 1 = 83 (CMOVEs, HALT, ↑ loop iterations)

• # data fetches = 16 ⇒ total fetches = 99

Problem 3.

The program from the Cache performance lab is shown at the right. Assume the program is being run on a Beta with a cache with the following parameters:

- **2-way set-associative**
- **block size of 2**, i.e., 2 data words are stored in each cache line
- total number of data words in the cache is **32**
- **LRU** replacement strategy

```
I = 0x240          // location of program
A = 0x420          // location of array A
N = 16             // size of array (in words)

. = I              // start program here
test:
240 CMOVE(N,R0)    // initialize loop index J
244 CMOVE(0,R1)

loop:              // add up elements in array
248 SUBC(R0,1,R0)  // decrement index
24c MULC(R0,4,R2)  // convert to byte offset
    LD(R2,A,R3)    // load value from A[J]
    ADD(R3,R1,R1)  // add to sum
    BNE(R0,loop)   // loop N times

    BR(test)       // perform test again!

// allocate space to hold array
. = A
    STORAGE(N)     // N words
```

(A) The cache will divide the 32-bit address supplied by the Beta into three fields: B bits of block offset (including byte offset bits), L bits of cache line index, and T bits of tag field. Based on the cache parameters given above, what are the appropriate values for B, L, and T?

8 bytes/line **value for B:** 3

8 lines/way **value for L:** 3

32-L-B **value for T:** 26

(B) If the MULC instruction is resident in a cache line, what will be its cache line index? the value of the tag field for the cache?

0x24C = 0010 0100 1100
offset: 0b 100
line: 0b001
tag: 0b1001

Cache line index for MULC when resident in cache: 1

Tag field for MULC when resident in cache: 0x 9

(C) With the values of I, A, and N as shown, list *all* the values j ($0 \le j < N$) where the location holding the value A[j] will map to the same cache line index as the MULC instruction in the program.

List all j where A[j] have the same cache line index as MULC: 10, 11

A[0] @ 0x420 ⇒ cache line 4. 2 words/line ⇒ A[8], A[9] are in cache line 4.

(D) If the outer loop is run many times, give the steady-state hit ratio for the cache, i.e., assume that the number of compulsory misses as the cache is first filled are insignificant compared to the number of hits and misses during execution.

everything fits in the cache!

Steady-state hit ratio (%): 100%

Problem 4.

Consider a 2-way set-associative cache where each way has 4 cache lines with a **block size of 2 words**. Each cache line includes a valid bit (V) and a dirty bit (D), which is used to implement a write-back strategy. The replacement policy is least-recently-used (LRU). The cache is used for both instruction fetch and data (LD,ST) accesses. Please use this cache when answering questions (A) through (D).

(A) Using this cache, a particular benchmark program experiences an average memory access time (AMAT) of 1.3 cycles. The access time on a cache hit is 1 cycle; the miss penalty (i.e., additional access time) is 10 cycles. What is the hit ratio when running the benchmark program? You can express your answer as a formula if you wish:

Hit ratio for benchmark program: 0.97

$1.3 = 1 + (1 - \text{hit ratio})(10)$

(B) The circuitry for this cache uses various address bits as the block offset, cache line index and tag field. Please indicate which address bits A[31:0] are used for each purpose by placing a "B" in each address bit used for the block offset, "L" in each address bit used for the cache line index, and "T" in each address bit used for the tag field.

Fill in each box with "B", "L", or "T"

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T																										T	L	L	B	0	0

... all T's ...

(C) This cache needs room to store new data and based on the LRU replacement policy has chosen the cache line whose information is shown to the right for replacement. Since the current contents of that line are marked as dirty (D = 1), the cache must write some information back to main memory. What is the address of each memory location to be written? Please give each address in hex.

Way: 0
Cache line index: 3
Valid bit (V): 1
Dirty bit (D): 1
Tag field: 0x123

Addresses of each location to be written (in hex): 0x2478, 0x247C

~~0001 0010 0011~~

using fields from (B)

B: 100, 000

INDEX = 0b11

TAG: 0b 0001 0010 0011

ADDRESS = 0b 10 0100 0111 1X00

(D) This cache is used to run the following benchmark program. The code starts at memory address 0; the array referenced by the code has its first element at memory address 0x2000. First determine the number of memory accesses (both instruction and data) made during each iteration through the loop. Then estimate the steady-state average hit ratio for the program, i.e., the average hit ratio after many iterations through the loop.

```
. = 0
    CMOVE(0,R0)          // byte index into array
    CMOVE(0,R1)          // initialize checksum accumulator
loop:
    LD(R0,array,R2)      // load next element of array
    SHLC(R1,1,R1)        // shift checksum
    ADDC(1,R1,R1)        // increment checksum
    ADD(R2,R1,R1)        // include data value in checksum
    ADDC(R0,4,R0)        // byte index of next array element
    CMPLTC(R0,1000,R2)   // process 250 entries
    BT(R2,loop)
    HALT()

. = 0x2000
array:
    … array contents here …
```

each iteration: 7 ifetches, 1 data access
- instructions occupy 4 cache lines (2 words/line)
  - ↳ 2-way ⇒ 100% hit on ifetch
- data accesses: new word every cycle
  - ↳ 2 words/line ⇒ 50% hit

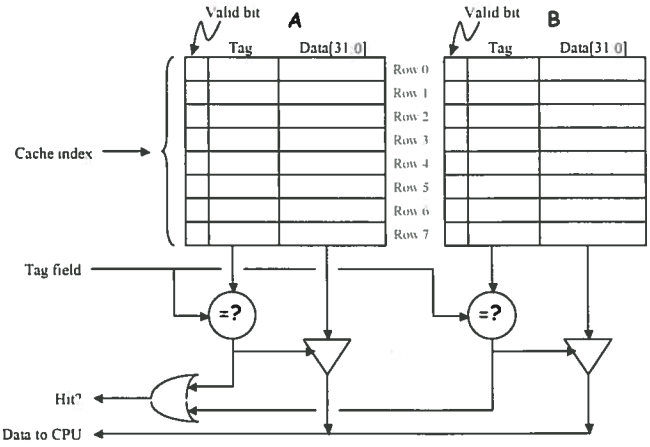
Number of memory accesses made during each iteration of the loop: 8

Estimated steady-state average hit ratio: 15/16 = 93.75%

1 miss every two iterations

Problem 5.

Consider the diagram to the right for a 2-way set associative cache to be used with our Beta design. Each cache line holds a single 32-bit word of data along with its associated tag and valid bit (0 when the cache line is invalid, 1 when the cache line is valid).



(A) The Beta produces 32-bit byte addresses, A[31:0]. To ensure the best cache performance, which address bits should be used for the cache index? For the tag field?

8 cache lines ↳

address bits used for cache index: A[4 : 2]

address bits used for tag field: A[31 : 5]

↑ remaining bits are tag.

(B) Suppose the Beta does a read of location 0x5678. Identify which cache location(s) would be checked to see if that location is in the cache. For each location specify the cache section (A or B) and row number (0 through 7). E.g., **3A** for row 3, section A. If there is a cache hit on this access what would be the contents of the tag data for the cache line that holds the data for this location?

0x5678
0b0101100111100 **cache location(s) checked on access to 0x5678:** 6A, 6B
2 B 3 (offset
tag line index = 0x6 **cache tag data on hit for location 0x5678 (hex): 0x** 2B3

(C) Assume that checking the cache on each read takes 1 cycle and that refilling the cache on a miss takes an *additional* 8 cycles. If we wanted the *average* access time over many reads to be 1.1 cycles, what is the minimum hit ratio the cache must achieve during that period of time? You needn't simplify your answer.

$1.1 = 1 + (1 - HR) \cdot (8)$

minimum hit ratio for 1.1 cycle average access time: 7.9/8

(D) Estimate the approximate cache hit ratio for the following program. Assume the cache is empty before execution begins (all the valid bits are 0) and that an LRU replacement strategy is used. Remember the cache is used for both instruction and data (LD) accesses.

LINE	ADDR	
0	0	`. = 0`
1	4	`CMOVE(source,R0)`
2	8	`CMOVE(0,R1)`
3	C	`CMOVE(0x1000,R2)`
4	10	`loop: LD(R0,0,R3)`
5	14	`ADDC(R0,4,R0)`
6	18	`ADD(R3,R1,R1)`
7	1C	`SUBC(R2,1,R2)`
0	20	`BNE(R2,loop)`
1	24	`ST(R1,source)`
		`HALT()`
		`. = 0x100`
		`source:`
		`. = . + 0x4000 // Set source to 0x100, reserve 1000 words`

each iteration
- 5 inst. fetches ≅ 100% hit
- 1 data fetch ≅ 0% hit
  - no data word fetched twice

6 total

approximate hit ratio: 5/6

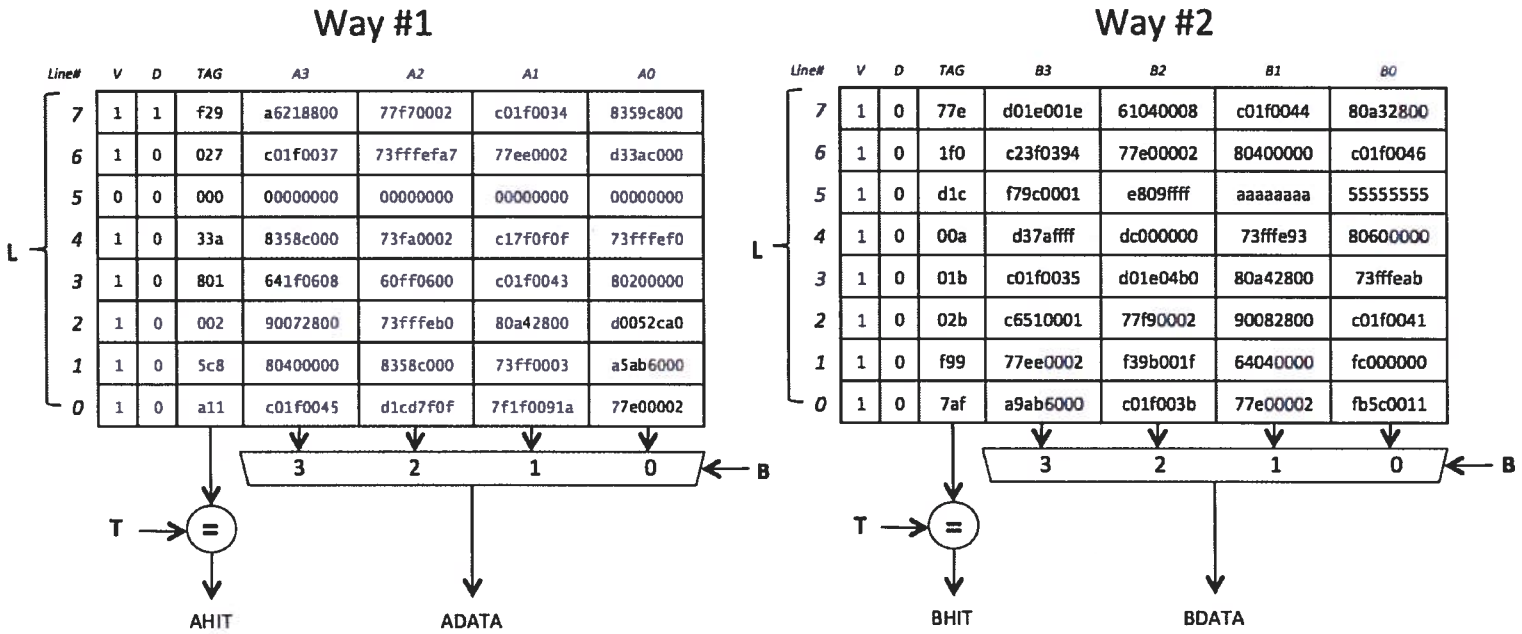
(E) After the program of part (D) has finished execution what information is stored in row 4 of the cache? Give the addresses for the two locations that are cached (one in each of the sections) or briefly explain why that information can't be determined.

Addresses whose data is cached in "Row 4": 0x10 **and** 0x40F0 (data)

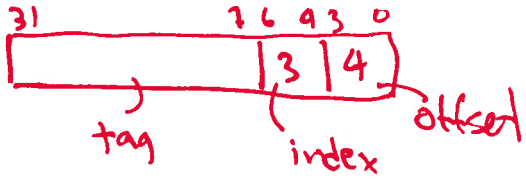
ADDR	40E0	40E4	40E8	40EC	40F0	40F4	40F8	40FC
LINE	0	1	2	3	4			

Problem 6.

A standard unpipelined Beta is connected to a 2-way set-associative cache containing 8 sets, with a block size of 4 32-bit words. The cache uses a LRU replacement strategy. At a particular point during execution, a snapshot is taken of the cache contents, which are shown below. All values are in hex; **assume that any hex digits not shown are 0.**



(A) The cache uses bits from the 32-bit byte address produced by the Beta to select the appropriate set (L), as input to the tag comparisons (T) and to select the appropriate word from the data block (B). For correct and optimal performance what are the appropriate portions of the address to use for L, T and B? Express your answer in the form "A[N:M]" for N and M in the range 0 to 31, or write "CAN'T TELL".



Address bits to use for L: A[6:4]

Address bits to use for T: A[31:7]

Address bits to use for B: A[3:2]

(B) For the following addresses, if the contents of the specified location appear in the cache, give the location's 32-bit contents in hex (determined by using the appropriate value from the cache). If the contents of the specified location are NOT in the cache, write "MISS".

B=0, L=0, T=0x1422 **Contents of location 0xA1100 (in hex) or "MISS": 0x** miss

B=2, L=4, T=0xA **Contents of location 0x548 (in hex) or "MISS": 0x** DC000000

(C) Ignoring the current contents of the cache, is it possible for the contents of locations 0x0 and 0x1000 to both be present in the cache simultaneously?

Locations 0x0 and 0x1000 present simultaneously (circle one): (YES) ... **NO**

both map to cache line 0, but it's a 2-way cache

(D) (Give a one-sentence explanation of how the D bit got set to 1 for Line #7 of Way #1.

One sentence explanation

ST changed a value of a word on that cache line but it hasn't yet been written back to memory.

(E) The following code snippet sums the elements of the 32-element integer array X. Assume this code is executing on a Beta with a cache architecture as described above and that, initially, the cache is empty, i.e., all the V bits have been set to 0. Compute the hit ratio as this program runs until it executes the HALT() instruction, a total of 2 + (6*32) + 1 = 195 instruction fetches and 32 data accesses.

Hit ratio: 216/227 = 95.29%

```
        . = 0
+2  {   CMOVE(0, R0)          // loop counter
    {   CMOVE(0, R1)          // accumulated sum

loop:
        SHLC(R0, 2, R2)       // convert loop counter to byte offset
        LD(R2, X, R3)         // load next value from array
        ADD(R3, R1, R1)       // add value to sum
        ADDC(R0, 1, R0)       // increment loop counter
        CMPLTC(R0, 32, R2)    // finished with all 32 elements?
        BT(R2,loop)           // nope, keep going

+1      HALT()                // all done, sum in R1

X:      LONG(1)               // the 32-element integer array X
        LONG(2)
        ...
        LONG(32)
```

6 ifetch + 1 dfetch × 32 iterations

11 misses total out of 227 accesses

- 2-way cache: 100% hit rate on ifetch
- block size = 4:
  - 3 misses to load instructions
  - 32 data accesses, but only 25% miss (each miss loads 4 words) ⇒ 8 data misses total

Problem 7.

After his geek hit single *I Hit the Line,* renegade singer Johnny Cache has decided he'd better actually learn how a cache works. He bought three Beta processors, identical except for their cache architectures:

- ***Beta1*** has a 64-line direct-mapped cache
- ***Beta2*** has a 2-way set associative cache, LRU, with a total of 64 lines
- ***Beta3*** has a 4-way set associative cache, LRU, with a total of 64 lines

Note that each cache has the same total capacity: 64 lines, each holding a single 32-bit **word** of data or instruction. All three machines use the same cache for data and instructions fetched from main memory.

Johnny has written a simple test program:

```
// Try a little cache benchmark
I = 0x1000                      // where program lives
A = 0x2000                      // data region 1
B = 0x3000                      // data region 2
N = 16                          // size of data regions (BYTES!)

. = I                           // start program here
P:    CMOVE(1000, R6)           // outer loop count
Q:    CMOVE(N, R0)              // Loop index I (array offset)
R:    SUBC(R0, 4, R0)           // I = I-1
      LD(R0, A, R1)             // read A[I]
      LD(R0, B, R2)             // read B[I]
      BNE(R0, R)
      SUBC(R6,1, R6)            // repeat many times
      BNE(R6, Q)
      HALT()
```

Johnny runs his program on each Beta, and finds that one Beta model outperforms the other two.

(A) (2 points) Which Beta gets the highest hit ratio on the above benchmark?

3 regions: 0x1000+, 0x2000+, 0x3000+ **Circle one:** **Beta1** **Beta2** **Beta3**

(B) (2 points) Johnny changes the value of **B** in his program to **0x2000** (same as **A**), and finds a substantial improvement in the hit rate attained by one of the Beta models (approaching 100%). Which model shows this marked improvement?

now just 2 regions: 0x1000+, 0x2000+ **Circle one:** **Beta1** **Beta2** **Beta3**

(C) (3 points) Finally, Johnny sets **I**, **A**, and **B** each to **0x0**, and sets **N** to **64**. What is the TOTAL number of cache misses that will occur executing this version of the program on each of the Beta models?

TOTAL cache misses running on Beta1: 16 ; **Beta2:** 16 ; **Beta3:** 16

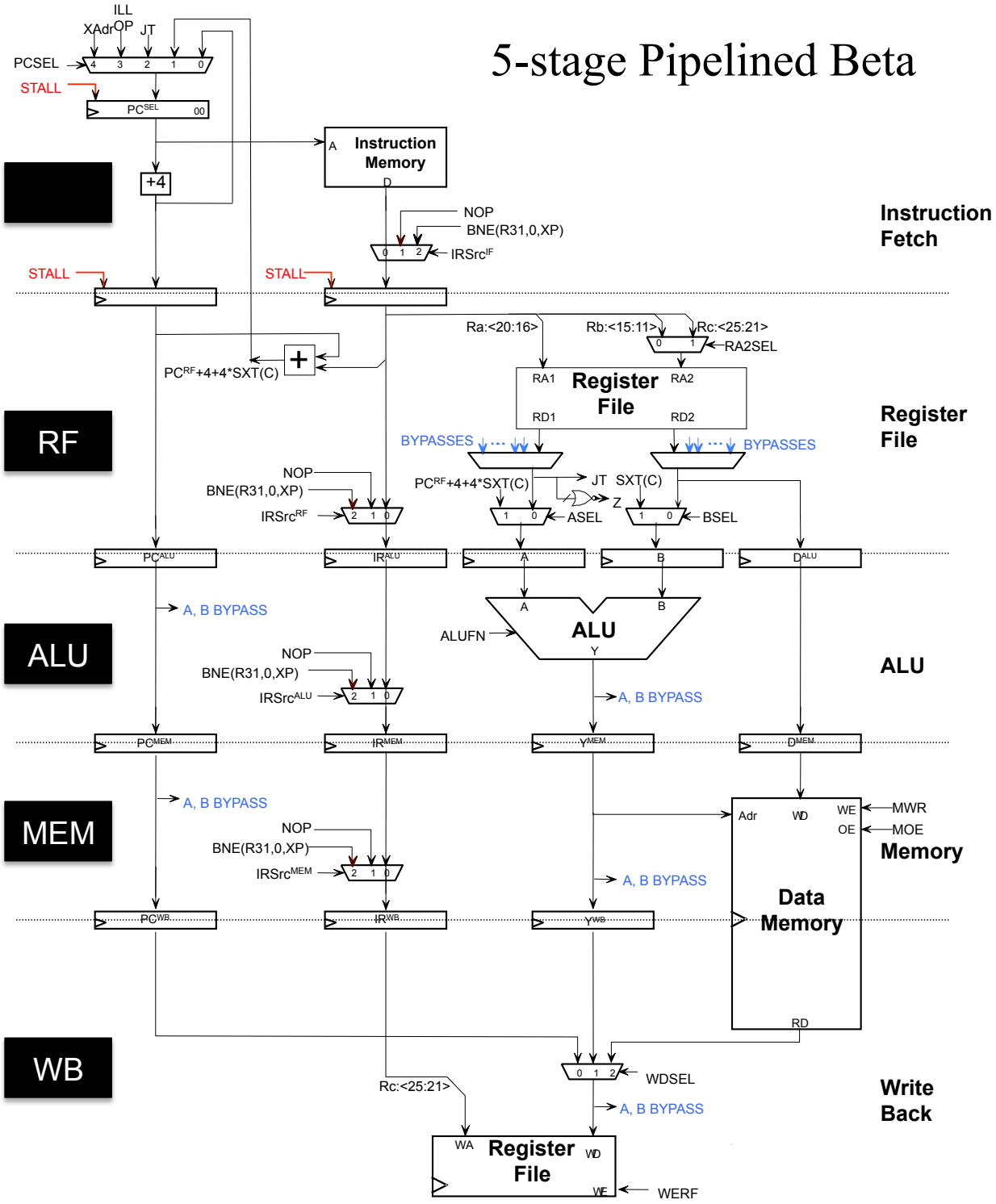
only compulsory misses for all these caches

6.004 Computation Structures
Spring 2017

Computation Structures

Pipelining the Beta Worksheet

Options for dealing with *data* and *control* hazards: **stall, bypass, speculate**



Problem 1.

The program shown on the right is executed on a 5-stage pipelined Beta with full bypassing and annulment of instructions following taken branches.

The program has been running for a while and execution is halted at the end of cycle 108.

The pipeline diagram shown below shows the history of execution at the time the program was halted.

```
   . = 0
outer_loop:
   CMOVE(16,R0)    // initialize loop index J
   CMOVE(0,R1)

loop:              // add up elements in array
   SUBC(R0,1,R0)   // decrement index
   MULC(R0,4,R2)   // convert to byte offset
   LD(R2,0x310,R3)// load value from A[J]
   ADD(R3,R1,R1)   // add to sum
   BNE(R0,loop)    // loop until all words are summed

   BR(outer_loop) // perform test again!
```

stall stall annul

cycle	100	101	102	103	104	105	106	107	108
IF	MULC	LD	ADD	BNE	BNE	BNE	BR	SUBC	MULC
RF	SUBC	MULC	LD	ADD	ADD	ADD	BNE	NOP	SUBC
ALU	NOP	SUBC	MULC	LD	NOP	NOP	ADD	BNE	NOP
MEM	BNE	NOP	SUBC	MULC	LD	NOP	NOP	ADD	BNE
WB	ADDC	BNE	NOP	SUBC	MULC	LD	NOP	NOP	ADD

Please indicate on which cycle(s), 100 through 108, each of the following actions occurred. If the action did not occur in any cycle, write "NONE". You may wish to refer to the signal names in the 5-stage Pipelined Beta Diagram included in the reference material.

Register value used from Register File: 100, 105, 106, 108

Register value bypassed from ALU stage to RF stage: 101, 102

Register value bypassed from MEM stage to RF stage: none

Register value bypassed from WB stage to RF stage: 105

IRSrc^{IF} was 1: 106

IRSrc^{IF} was 2: none

STALL was 1: 103, 104

PCSEL was 1: 106

WDSEL was 2: 105

Problem 2.

The following program fragments are being executed on the 5-stage pipelined Beta described in lecture with full bypassing, stall logic to deal with LD data hazards, and speculation for JMPs and taken branches (i.e., IF-stage instruction is replaced with a NOP if necessary). The execution pipeline diagram is shown for cycle 1000 of execution. Please fill in the diagram for cycle 1001; use "?" if you cannot tell what opcode to write into a stage. Then for **both** cycles use arrows to indicate any bypassing from the ALU/MEM/WB stages back to the RF stage (see example for cycle 1000 in part A).

(A) (2 points) **Assume BNE is taken.**

```
    ...
    ADDC(R1,5,R1)
L:  SUBC(R1,1,R1)
    SHRC(R0,1,R0)
    BNE(R1,L)
    ST(R1,data)
    ...
```

Cycle	*1000*	*1001*
IF	ST	SUBC
RF	BNE	NOP
ALU	SHRC	BNE
MEM	SUBC	SHRC
WB	NOP	SUBC

(B) (2 points)

```
    ...
    ST(R31,0,BP)
    LD(BP,-12,R17)
    ADDC(SP,4,SP)
    SHLC(R17,2,R1)
    ST(R1,-4,SP)
    BEQ(R31,fact,LP)
    ...
```

Cycle	*1000*	*1001*
IF	ST	ST
RF	SHLC	SHLC
ALU	ADDC	NOP
MEM	LD	ADDC
WB	ST	LD

(C) (2 points)

```
    ...
    XOR(R1,R2,R1)
    MULC(R2,3,R2)
    SUB(R2,R1,R3)
    AND(R3,R1,R2)
    ADD(R3,R2,R3)
    ST(R3,x)
    ...
```

Cycle	*1000*	*1001*
IF	ADD	ST
RF	AND	ADD
ALU	SUB	AND
MEM	MULC	SUB
WB	XOR	MULC

(D) (2 points) Assume during cycle 1000 the DIV instruction in the RF stage **triggers an ILLEGAL OPCODE (ILLOP) exception.**

```
    ...
    LD(x,R1)
    LD(y,R2)
    SHLC(R1,3,R1)
    DIV(R2,R1,R3)
    ADDC(R3,17,R3)
    ST(R3,z)
    ...
```

Cycle	*1000*	*1001*
IF	ADDC	?
RF	DIV	NOP
ALU	SHLC	BNE(R31,...,XP)
MEM	NOP	SHLC
WB	LD	NOP

inst @ 0x4

Problem 3.

In answering this question, you may wish to refer to the diagram of the 5-stage pipelined beta provided with the reference material.

The loop on the right has been executing for a while on our standard 5-stage pipelined Beta with branch annulment and full bypassing. The pipeline diagram below shows the opcode of the instruction in each pipeline stage during 10 consecutive cycles of execution.

```
    ...
L1: SUBC(R0,4,R0)
    CMPLTC(R0,10,R1)
    BF(R1,L2)
    LD(R0,A,R2)
    BR(L3)
L2: LD(R0,B,R2)
L3: ST(R2,C,R31)
    BNE(R0,L1)
    ADDC(R2,1,R2)
    ...
```

stall stall

Cycle #	300	301	302	303	304	305	306	307	308	309
IF	SUBC	CMPLTC	BF	LD	LD	ST	BNE	BNE	BNE	ADDC
RF	NOP	SUBC	CMPLTC	BF	NOP	LD	ST	ST	ST	BNE
ALU	BNE		SUBC	CMPLTC	BF	NOP	LD	NOP	NOP	ST
MEM	ST			SUBC	CMPLTC	BF	NOP	LD	NOP	NOP
WB	NOP				SUBC	CMPLTC	BF	NOP	LD	NOP

(A) (4 Points) Indicate which bypass/forwarding paths are active in each cycle by drawing a vertical arrow in the pipeline diagram from pipeline stage X in a column to the RF stage in the same column if an operand would be bypassed from stage X back to the RF stage that cycle. Note that there may be more than one vertical arrow in a column.

Draw bypass arrows in pipeline diagram above

(B) (2 Points) Assume that the previous iteration of the loop executed the same instructions as the iteration show here. Please complete the pipeline diagram for cycle 300 by filling in the OPCODEs for the instructions in the RF, ALU, MEM, and WB stages.

Fill in OPCODEs for Cycle 300

For the following questions *think carefully* about when a signal would be asserted in order to produce the effect you see in the pipeline diagram.

(C) (2 Points) During which cycle(s), if any, would the $IRSrc^{IF}$ signal be 1?

$IRSrc^{IF}$ == 1 when taken branches are in RF stage.

Cycle number(s) or NONE: 303, 309

(D) (2 Points) During which cycle(s), if any, would the $IRSrc^{RF}$ signal be 1?

$IRSrc^{RF}$ == 1 when register operand not available in the data path.

Cycle number(s) or NONE: 306, 307

(E) (2 Points) During which cycle(s), if any, would the STALL signal be 1, *i.e.*, cycle(s) when the IF and RF stages would be stalled?

STALL is 1 when $IRSrc^{RF}$ is 1.

Cycle number(s) or NONE: 306, 307

Problem 4.

You've discovered a secret room in the basement of the Stata center full of discarded 5-stage pipelined Betas. Unfortunately, many have certain defects. You discover that they fall into four categories:

C1: Completely functional 5-stage Betas with working bypass paths, annulment, and other components.
C2: Betas with a bad register file: all data read from the register file is zero.
C3: Betas without bypass paths: all source operands come from the register file.
C4: Betas without annulment of instructions following branches.

To help sort the Beta chips into the above classes, you write the following small test program:

```
. = 0x0
// Start at 0x0, with ZERO in all registers…
      ADDC(R31, 4, R0)
      BEQ(R31, X, R2)
      MULC(R2, 2, R2)
X:    SUBC(R2, 4, R2)
      ADD(R0, R2, R3)
      JMP(R3)
```

C1 C2 C3 C4

R0←4 R0←4 R0←4 R0←4
R2←8 R2←8 R2←8 R2←8
— — — R2←16
R2←4 R2←4 R2←-4 R2←12
R3←8 *R3←4 R3←4 R3←16

*R0 reads as 0

Your plan is to single-step through the program using each Beta chip, carefully noting the address the final JMP loads into the PC. Your goal is to determine which of the above four classes a chip falls into by this JMP address.

For each class of Beta processor described above, specify the value that will be loaded into the PC by the final JMP instruction.

Pipeline diagram showing first 7 cycles of test program executing on C1:

cycle	*0*	*1*	*2*	*3*	*4*	*5*	*6*
IF	ADDC	BEQ	MULC	SUBC	ADD	JMP	
RF		ADDC	BEQ	NOP	SUBC	ADD	JMP
ALU			ADDC	BEQ	NOP	SUBC	ADD
MEM				ADDC	BEQ	NOP	SUBC
WB					ADDC	BEQ	NOP

C1: JMP goes to address: 8

C2: JMP goes to address: 4

C3: JMP goes to address: 0

C4: JMP goes to address: 16

Problem 5.

Recall the code for gcd that we saw in lecture, and the assembly code for the while loop:

C code

```
int gcd(int x, int y) {
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}
```

Corresponding Beta assembly for while loop

```
        // x in R0, y in R1
        CMPEQ(R0, R1, R2)   // R2 ← (x == y)
        BT(R2, end)
loop:   CMPLT(R1, R0, R2)   // R2 ← (x > y)
        BF(R2, else)
        SUB(R0, R1, R0)     // x ← x - y
        BR(cond)
else:   SUB(R1, R0, R1)     // y ← y - x
cond:   CMPEQ(R1, R0, R2)   // R2 ← (x == y)
        BF(R2, loop)
end:    …
```

Assume a **5-stage pipelined Beta** as presented in lecture, with **full bypass paths**, and which **predicts branches by assuming they are not taken** to resolve control (i.e., the instruction following the branch is fetched in the IF stage on the cycle after the branch is in the IF stage).

First, find the number of cycles per iteration in steady state (do not worry about the first or last iterations). Note that the BF(R2, else) branch is not taken if x > y and taken if x < y, so you should consider these two cases separately.

(A) Fill in the following table:

	Iterations where x > y	Iterations where x < y
Instructions per iteration	6	5
+ Cycles lost to data hazards	0	0
+ Cycles lost to annulments	2	2
= Total cycles per iteration	8	7

x>y

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	CMPLT	BF	SUB	BR	SUB	CMPEQ	BF	…	CMPLT					
RF		CMPLT	BF	SUB	BR	NOP	CMPEQ	BF	NOP	CMPLT				
ALU			CMPLT	BF	SUB	BR	NOP	CMPEQ	BF	NOP	CMPLT			
MEM				CMPLT	BF	SUB	BR	NOP	CMPEQ	BF	NOP	CMPLT		
WB					CMPLT	BF	SUB	BR	NOP	CMPEQ	BF	NOP	CMPLT	

one iteration

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
IF	CMPLT	BF	SUB	SUB	CMPEQ	BF	…	CMPLT						
RF		CMPLT	BF	NOP	SUB	CMPEQ	BF	NOP	CMPLT					
ALU			CMPLT	BF	NOP	SUB	CMPEQ	BF	NOP	CMPLT				
MEM				CMPLT	BF	NOP	SUB	CMPEQ	BF	NOP	CMPLT			
WB					CMPLT	BF	NOP	SUB	CMPEQ	BF	NOP	CMPLT		

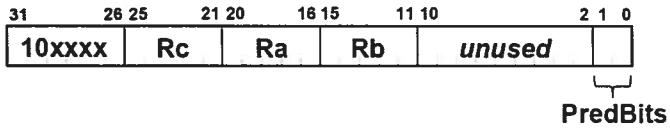
x<y

one iteration

To make this code faster, we modify the Beta ISA and pipeline to implement a technique called **predication** to reduce the number of branches.

First, all the compare instructions (CMPEQ, CMPLT, CMPLE, and their C variants) write their result into a special 1-bit register, called the **predicate register**, in addition to their normal destination register.

Second, we change the format of ALU instructions with two register source operands to use their lower two bits, which were previously unused:



- If PredBits == 10, the instruction only executes if the predicate register is false (0)
- If PredBits == 11, the instruction only executes if the predicate register is true (1)
- If PredBits == 0X, the instruction always executes and writes its result, as before

We say that instructions that depend on the predicate register are predicated. We denote predicated instructions in assembly as follows:

- If PredBits == 10, OP(Ra, Rb, Rc) [predFalse]
- If PredBits == 11, OP(Ra, Rb, Rc) [predTrue]
- If PredBits == 0X, OP(Ra, Rb, Rc), as before

For example, consider the following instruction sequence:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
ADD(R4, R5, R6) [predTrue]
SUB(R5, R6, R7)
```

If the CMPLT instruction evaluates to true (i.e., writes 1 to R3), this sequence is equivalent to:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
ADD(R4, R5, R6)
SUB(R5, R6, R7)
```

If the CMPLT instruction evaluates to false (i.e., writes 0 to R3), this sequence is equivalent to:

```
CMPLT(R1, R2, R3)
MUL(R3, R4, R5)
SUB(R5, R6, R7)
```

(B) Modify the code to use predication, minimizing the number of instructions per loop iteration.

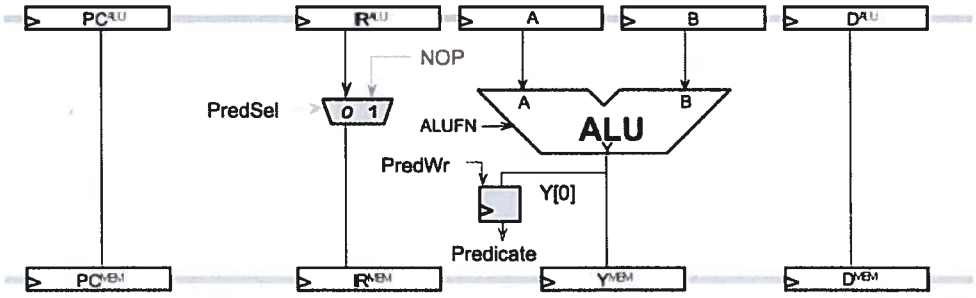
Original code

```
        // x in R0, y in R1
        CMPEQ(R0, R1, R2)
        BT(R2, end)
loop:   CMPLT(R1, R0, R2)
        BF(R2, else)
        SUB(R0, R1, R0)
        BR(cond)
else:   SUB(R1, R0, R1)
cond:   CMPEQ(R1, R0, R2)
        BF(R2, loop)
end:    …
```

Code with predication

```
        // x in R0, y in R1
        CMPEQ(R0, R1, R2)
        BT(R2, end)
loop:   CMPLT(R1, R0, R2)
        SUB(R0, R1, R0) [predTrue]
        SUB(R1, R0, R1) [predFalse]
        CMPEQ(R1, R0, R2)
        BF(R2, loop)
end:    …
```

We implement predication in the pipelined Beta with minor changes to the ALU stage:



Comparison instructions write the 1-bit predicate register (the PredWr control signal ensures that only comparison instructions update the register). The PredSel mux annuls ALU instructions if they are predicated and should not execute according to the value of the predicate register.

(C) Write the Boolean expression for the PredSel control signal. You can use AND, OR, NOT, Predicate, and comparisons with PredBits (e.g., PredBits == 0b10).

PredSel = (IR^{ALU}[31:30] == 0b10) AND predBit[1]==1 and predBit[0]!=Predicate

(D) How fast is this modified code? Fill in the following table:

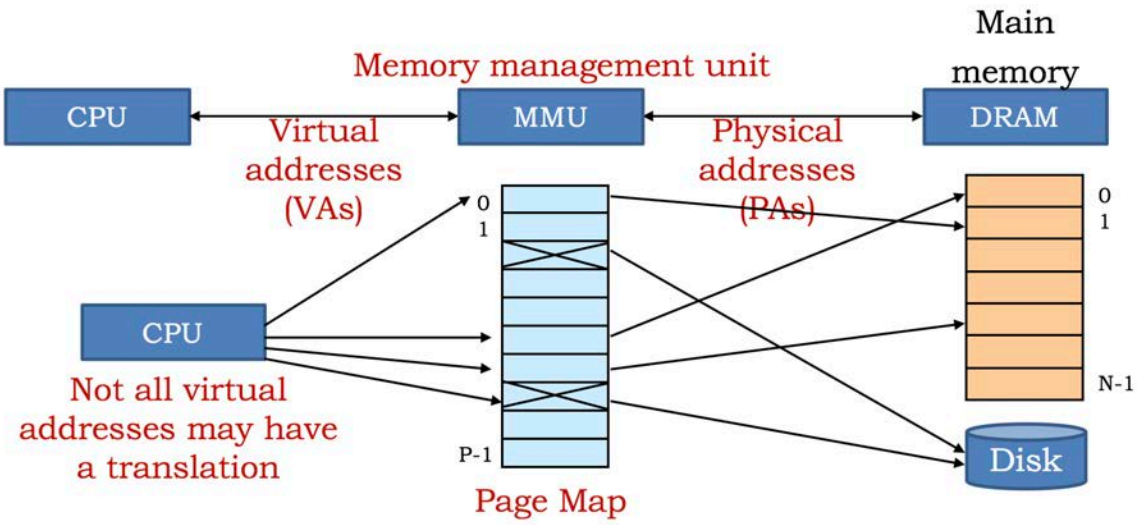
	Iterations where x > y	Iterations where x < y
Instructions per iteration	4	4
+ Cycles lost to data hazards	0	0
+ Cycles lost to annulments	2	2
= Total cycles per iteration	6	6

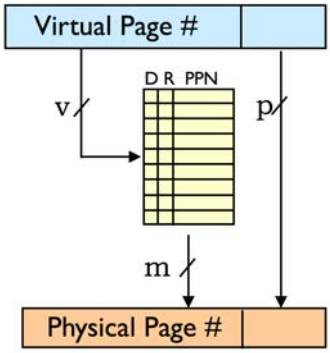
no cycles lost due to annulments after taken branches except for final BF.

6.004 Computation Structures
Spring 2017

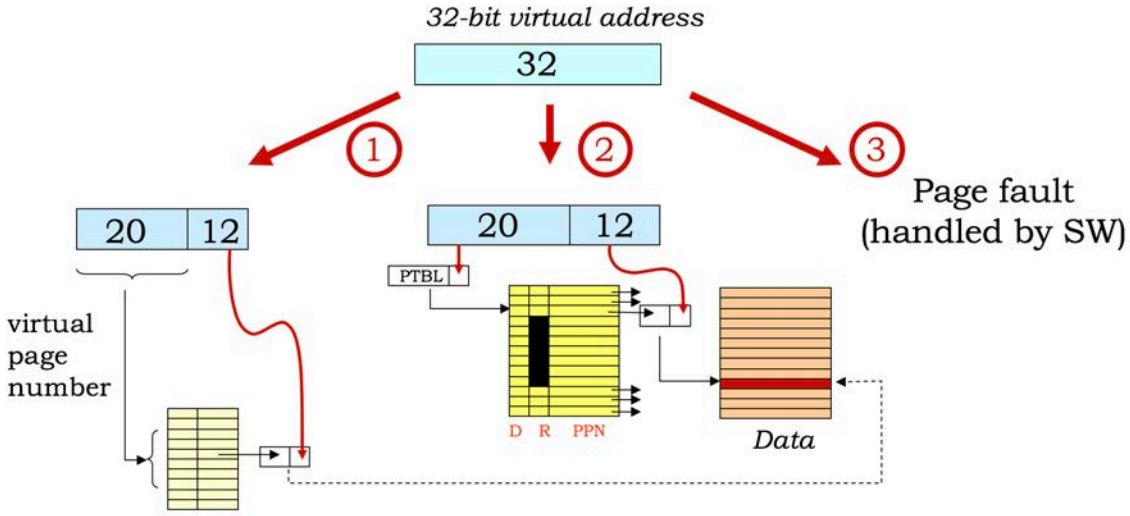
Computation Structures

Virtual Memory Worksheet





(v + p)	bits in virtual address
(m + p)	bits in physical address
2^v	number of *virtual* pages
2^m	number of *physical* pages
2^p	bytes per physical page
2^{v+p}	bytes in virtual memory
2^{m+p}	bytes in physical memory
$(m+2)2^v$	bits in the page map



Look in TLB: VPN→PPN cache
Usually implemented as a small fully-associative cache

Problem 1.

The micro-Beta has a 12-bit virtual address, an 11-bit physical address and uses a page size of 256 (= 2^8) bytes. The micro-Beta has been running for a while and at the current time the page map has the contents shown on the right.

VPN	D	R	PPN
0	0	1	2
1	—	0	—
2	0	1	4
3	—	0	—
4	1	1	0
5	1	1	1
6	~~1~~	~~0~~ 1	~~1~~ 5
7	—	0	—
8	—	0	—
9	—	0	—
A	—	0	—
B	—	0	—
C	1	1	7
D	1	1	6
LRU→E	~~1~~ —	~~1~~ 0	~~5~~ —
F	0	1	3

(A) Assuming each page map entry contains the usual dirty (D) and resident (R) bits, what it is the total size of the page map in bits?

PPN = 3 bits
entry = 5 bits
VPN = 4 bits

Size of page map (bits): $2^4 \cdot (2+3) = 80$

(B) (The following instruction, located at virtual address 0x0BA, is about to be executed.

0xBA ⇒ VPN 0 ⇒ PPN 2

```
LD(R31, 0x2C8, R0)
```

0x2C8 ⇒ VPN 2 ⇒ PPN 4

When the instruction is executed, what main memory locations are accessed by the instruction fetch and then the memory access initiated by the LD? Use the page map shown to the right. Assume the LRU page is virtual page 0xE.

Physical address for instruction fetch: 0x 2BA

Physical addr for data read by LD instruction: 0x 4C8

(C) A few instructions later, the following instruction, located at virtual address 0x0CC, is executed:

```
ST(BP, -4, SP)    // current value of SP = 0x604
```

Please mark up the page map to show its contents after the ST has been executed. Use the page map shown to the right. Assume the LRU page is virtual page 0xE.

Remember to show any changes to the dirty and resident control bits as well as updates to the physical page numbers. If an entry in the page map no longer matters, please indicate that by replacing it with "— 0 —" for the D, R and PPN entries.

Show updated contents of page map

0xCC ⇒ VPN 0 ⇒ PPN 2

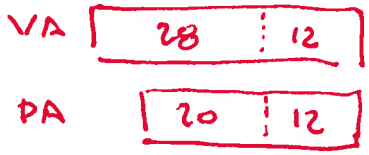
0x604 - 4 = 0x600 ⇒ VPN 6 ⇒ page fault

reuse LRU page (VPN 0xE)

Problem 2.

Consider a Beta processor that includes a 40-bit virtual address, an MMU that supports 4096 (2^{12}) bytes per page, 2^{32} bytes of physical memory, and a large Flash memory that serves as a disk. The MMU and the page fault handler implement an LRU replacement strategy.

(A) What is the size of the page map for this processor? Assuming the page map includes the standard dirty and resident bits, specify the width of each page map entry in bits, and number of entries in the page map.



D+R+ PPN

Size of page map entry in bits: 22

2^V

Number of entries in the page map: 2^{28}

(B) The following test program is running on this Beta processor. The first 8 locations of the page table, just before executing this test program, are shown below; the least-recently-used page ("LRU") and next least-recently-used page ("next LRU") are as indicated. This Beta processor also has a 4 element, fully associative, Translation Lookaside Buffer (TLB) that caches page map translations from VPN to PPN.

```
. = 0x0
0  ADDC(R31,0x2800,R3)
4  LD(R3,0,R5)
8  ST(R5,0x4100,R31)
```

inst @ VPN 0
LD data @ VPN 2
ST data @ VPN 4

TLB

	Tag (VPN)	D	R	PPN
LRU→	~~0x3~~ 0	1	1	~~0x1~~ 7
	0x2	0	1	0x3
	0x6	0	1	0x2
Next LRU→	0x1	0	1	0x5

Page Map

VPN	D	R	PPN
0	1	1	0x7
1	0	1	0x5
2	0	1	0x3
LRU→ ~~3~~	~~1~~	~~1~~ 0	~~0x1~~ –
4	~~–~~ 1	~~0~~ 1	~~–~~ 0x1
5	0	1	0x0
6	0	1	0x2
Next LRU→ 7	0	1	0x6

write to disk!

For each virtual page that is accessed by this program, specify the **VPN**, whether or not it results in a **TLB hit** on the first access to that page, whether or not it results in a **page fault**, and the **PPN** that the page ultimately maps to. *You may not need to use all rows of the table.*

VPN	TLB Hit (Yes/No)	Page Fault (Yes/No)	PPN
0	NO	NO	7
2	YES	NO	3
4	NO	YES	1

(C) Which physical pages, if any, need to be written to disk during the execution of the test program in part B?

Physical page numbers written to disk or NONE: 1

(D) What is the physical address of the LD instruction?

Physical address of LD instruction: 0x 7004

Problem 3.

Consider a Beta processor that includes a 32-bit virtual address, an MMU that supports 4096 (2^{12}) bytes per page, 2^{24} bytes of physical memory, and a large Flash memory that serves as a disk. The MMU and the page fault handler implement an LRU replacement strategy.

(A) The designers are thinking about implementing the page map using a separate SRAM memory with L entries, where each entry has B bits. If the page map includes the standard dirty and resident bits, what are the appropriate values for the parameters L and B?

$2^{32}/2^{12} = 2^{20}$ virtual pages **Appropriate value for the parameter L:** 2^{20}

Appropriate value for the parameter B: 12+2 = 14

$2^{24}/2^{12} = 2^{12}$ physical pages

(B) If the designers decide to decrease the page size to 2048 (2^{11}) bytes but keep the same size virtual and physical addresses, what affect will the change have on the following architectural parameters? Use a letter "a" through "e" to indicate how the *new* value of the parameter compares to the *old* value of the parameter:

(a) doubled (b) increased by 1 (c) stays the same (d) decreased by 1 (e) halved

$2^{24}/2^{11} = 2^{13}$ physical pages **Size of page map entry in bits:** B

$2^{32}/2^{11} = 2^{21}$ virtual pages **Number of entries in the page map:** A

Maximum percentage of virtual memory that can be resident at any given time: C

old: $\frac{2^{12}}{2^{20}} = 2^{-8}$ new: $\frac{2^{13}}{2^{21}} = 2^{-8}$

(D) (4 points) A test program has been running on the Beta with a page size of 2^{12} bytes and has been halted *just before* execution of the following instruction at location 0x1234:

```
ST(R1,0x34C8,R31)    | PC = 0x1234
```

VPN 3 — VPN 1

- not resident
- reuse LRU page
- dirty so must write to disk

The first 8 locations of the page table at the time execution was halted are shown below; the least-recently-used page ("LRU") and next least-recently-used page ("next LRU") are as indicated. Assume that all the pages in physical memory are in use. Execution resumes and the ST instruction is executed.

VPN	*D*	*R*	*PPN*
0	1	1	0x1
1	0	1	0x0
2	1	1	0x6
3	~~1~~	~~0~~ 1	~~7~~
Next LRU→ 4	0	1	0x4
5	0	1	0x2
LRU→ 6	~~1~~	~~1~~ 0	~~0x7~~
7	0	1	0x3

write to disk!

Please **show the contents of the page table** after the ST instruction has completed execution by crossing out any values that changed and writing in their new values. Note that the D and PPN fields for a non-resident page do not need to be specified.

(E) (1 point) Which physical pages, if any, need to be written to disk during the execution of the ST instruction in part (D)?

Physical page numbers written to disk or NONE: 7

Problem 4.

Consider a virtual memory system that uses a single-level page map to translate virtual addresses into physical addresses. Each of the questions below asks you to consider what happens when **just ONE of the design parameters** (page size, virtual memory size, physical memory size) of the original system is changed. **Circle the correct answer.**

(A) If the physical memory size (in bytes) is **doubled**, the number of entries in the page table
(a) stays the same
(b) doubles
(c) is reduced by half
(d) increases by one
(e) decreases by one

# of entries depends only on # of bits in VPN

(B) If the page size (in bytes) is **halved**, the number of entries in the page table
(a) stays the same
(b) doubles
(c) is reduced by half
(d) increases by one
(e) decreases by one

VPN is one bit larger

(C) If the virtual memory size (in bytes) is **doubled**, the number of bits in each entry of the page table
(a) stays the same
(b) doubles
(c) is reduced by half
(d) increases by one
(e) decreases by one

PPN size is unchanged

(D) If the page size (in bytes) is **doubled**, the number of bits in each entry of the page table
(a) stays the same
(b) doubles
(c) is reduced by half
(d) increases by one
(e) decreases by one

PPN is one bit smaller

Consider a virtual memory system for the Gamma processor with 4096 (2^{12}) virtual pages and 16384 (2^{14}) physical pages where each page contains 1024 (2^{10}) bytes. The first 8 entries of the current page map are shown below:

14 bits for PPN

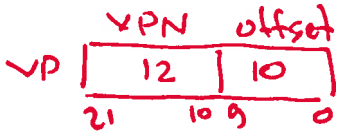
index	D	R	PPN
0	1	1	0x22
1	0	1	0x01
2	--	0	--
3	0	1	0x02
4	1	1	0x03
5	--	0	--
6	1	1	0x15
7	0	1	
...			

(E) What is the total number of bits in the page map?

$(2^{12}$ entries$) \cdot (D + R + PPN)$
$1 + 1 + 14 = 2^4$

Total number of bits in the page map: $2^{12} \cdot 2^4 = 2^{16}$

(F) Which address bits from the CPU are used to choose an entry from the page table?



Address bits used to choose page table entry: A[21 : 10]

(G) What is the physical address for the word at virtual location 0x1234? Write "not resident" if the location is not currently present in physical memory.

Physical address for byte at virtual address 0x1234 or "not resident": 0xE34

0x1234 = 0001 0010 0011 0100
VPN ↔ offset
VPN=4 → PPN=3

phys. addr: E 3 4
1110 0011 0100
↳ offset
↳ vpn

(H) Briefly explain what action caused the D bit for page 6 to be 1.

Briefly explain.

A ST instruction wrote to a location in virtual page 6.

Problem 5.

(A) A particular Beta implementation has 32-bit virtual addresses, 32-bit physical addresses and a page size of 2^{12} bytes. A test program has been running on this Beta and has been halted *just before* execution of the following instruction at location 0x1FFC:

```
LD(R31,0x34C8,R1)    | PC = 0x1FFC
ST(R1,0x6004,R31)    | PC = 0x2000
```

VPN 3 (0x34C8), VPN 1 (0x1FFC), VPN 6 (0x6004), VPN 2 (0x2000)

The first 8 locations of the page table at the time execution was halted are shown below; the least recently used page ("LRU") and next least recently used page ("next LRU") are as indicated. Assume that all the pages in physical memory are in use. Execution resumes and the LD and ST instructions are executed.

Please **show the contents of the page table** after the ST instruction has completed execution by crossing out any values that changed and writing in their new values.

VPN	*D*	*R*	*PPN*
0	1	1	0x1
1	0	1	0x0
LRU→ 2	~~1~~ 0	~~1~~ 1	~~0x6~~ 4
3	~~1~~ 0	~~0~~ 1	~~—~~ 6
Next LRU→ 4	~~0~~	~~1~~ 0	~~0x4~~
5	0	1	0x2
6	~~0~~ 1	1	0x7
7	0	1	0x3

VPN PPN
1 → 0
3 → miss → 6
2 → miss → 4
6 → 7

(B) Which physical pages, if any, needed to be written to disk during the execution of the LD and ST instructions?

Physical page numbers written to disk or NONE: 6

(C) Please give the 32-bit physical memory addresses used for the four memory accesses associated with the execution of the LD and ST instruction.

32-bit physical memory address of LD instruction: 0x 0FFC

32-bit physical memory address of data read by LD: 0x 64C8

32-bit physical memory address of ST instruction: 0x 9000

32-bit physical memory address of data written by ST: 0x 7004

Problem 6.

Consider a system with 40-bit virtual addresses, 36-bit physical addresses, and 64 KB (2^{16} bytes) pages. The system uses a page map to translate virtual addresses to physical addresses; each page map entry include dirty (D) and resident (R) bits.

(A) (2 points) Assuming a flat page map, what is the size of each page map entry, and how many entries does the page map have?

phys addr space : 2^{36} bytes

page size : 2^{16} bytes

2^{20} pages ⇒ 20-bit PPN + R + D = 22 bits

Size of page map entry in bits: 22

Number of entries in the page map: $2^{40-16} = 2^{24}$

(B) (1 point) If changed the system to use 16 KB (2^{14} bytes) pages instead of 64 KB pages, how would the number of entries in the page map change? Please give the ratio of the new size to the old size.

$\frac{2^{40}}{2^{14}} = 2^{26}$ $\quad$ $\frac{2^{40}}{2^{16}} = 2^{24}$

(# entries with 16 KB pages) / (# entries with 64 KB pages): 4

Assume 64 KB pages for the rest of this exercise.

(C) (6 points) The contents of the page map and TLB are shown to the right. The page map uses an LRU replacement policy, and the LRU page (shown below) will be chosen for replacement. For each of these four accesses, compute its corresponding physical address and indicate whether the access causes a TLB miss and/or a page fault. Assume each access starts with the TLB and Page Map state shown to the right.

TLB

VPN (tag)	V	D	PPN
0x0	1	0	0xBE7A
0x3	0	0	0x7
0x5	1	1	0xFF
0x2	1	0	0x900

Fill in table below

	Virt Addr	PPN (in hex)	Phys Addr (in hex)	TLB Miss?	Page Fault?
1.	0x06004	BE7A	BE7A6004	Y / (N)	Y / (N)
2.	0x30286	8	80286	(Y) / N	Y / (N)
3.	0x68030	70	708030	(Y) / N	Y / (N)
4.	0x4BEEF	8	8BEEF	(Y) / N	(Y) / N

miss! so reuse LRU page to hold contents of VPN 4.

Page Map

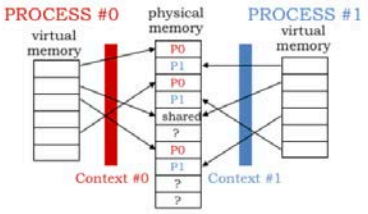
VPN	R	D	PPN	
0	1	0	0xBE7A	
1	0	0	---	
2	1	0	0x900	
3	1	0	0x8	← LRU PAGE
4	0	0	---	
5	1	1	0xFF	
6	1	0	0x70	
...				

6.004 Computation Structures
Spring 2017

Computation Structures

Virtualizing the Processor Worksheet

Building a Virtual Machine (VM)

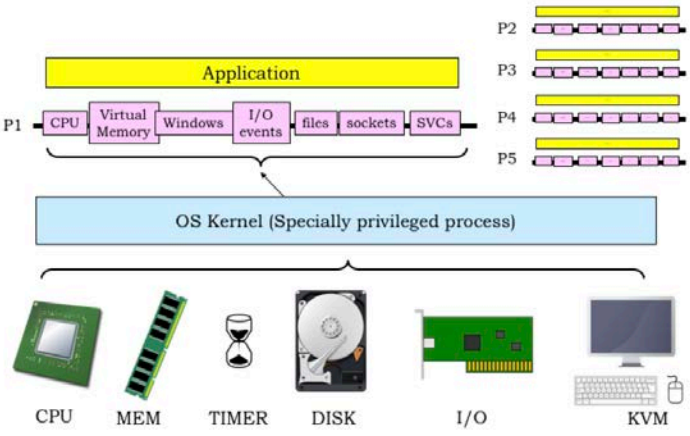


Goal: give each program its own "VIRTUAL MACHINE"; programs don't "know" about each other...

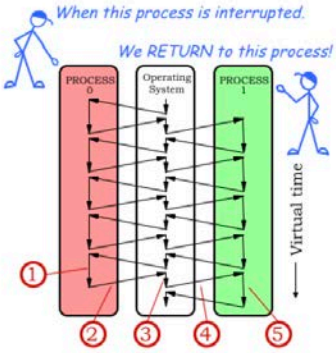
New abstraction: a process which has its own
- machine state: R0, ..., R30
- program (w/ shared code)
- context (virtual address space)
- virtual I/O devices
- PC, stack

"OS Kernel" is a special, privileged process running in its own context. It manages the execution of other processes and handles real I/O devices, emulating virtual I/O devices for each process.

One VM For Each Process



Processes: Multiplexing the CPU



1. Running in process #0
2. Stop execution of process #0 either because of explicit *yield* or some sort of timer *interrupt*; trap to handler code, saving current PC+4 in XP
3. First: save process #0 state (regs, context) Then: load process #1 state (regs, context)
4. "Return" to process #1: just like return from other trap handlers (ie., use address in XP) but we're returning from a *different* trap than happened in step 2!
5. Running in process #1

Interrupt Handler Coding

```
long TimeOfDay;
struct MState { int Regs[31];} UserMState;

/* Executed 60 times/sec */
Clock_Handler(){
   TimeOfDay = TimeOfDay+1;
   if (TimeOfDay % QUANTUM == 0) Scheduler();
}

Clock_h:
    ST(r0, UserMState)        // Save state of
    ST(r1, UserMState+4)      // interrupted
    ...                       //  app pgm...
    ST(r30, UserMState+30*4)
    LD(KStack, SP)            // Use KERNEL SP
    BR(Clock_Handler,lp)      // call handler
    LD(UserMState, r0)        // Restore saved
    LD(UserMState+4, r1)      //   state.
    ...
    LD(UserMState+30*4, r30)
    SUBC(XP, 4, XP)           // execute interrupted inst
    JMP(XP)                   // Return to app.
```

Handler (written in C)

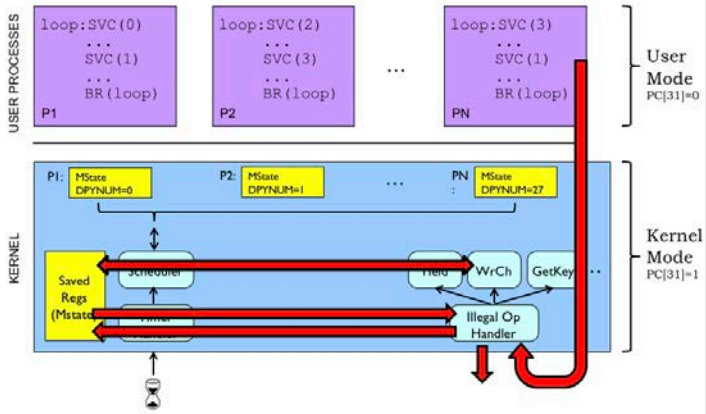
"Interrupt stub" (written in assy.)

Simple Timesharing Scheduler

```
struct MState { int Regs[31]; } UserMState;


struct PCB {       // Process Control Block
   struct MState State;       // Processor state
   struct Context PageMap;    // MMU state for proc
   int DPYNum;     // Console number (and other I/O state)
 } ProcTbl[N];                // one per process

int Cur;                      // index of "Active" process


Scheduler() {
   ProcTbl[Cur].State = UserMState; // Save Cur state
   Cur = (Cur+1)%N;                 // Incr mod N
   UserMState = ProcTbl[Cur].State; // Install state for next User
   LoadUserContext(ProcTbl[Cur].PageMap);  // Install context
}
```

Simple Timesharing Scheduler

OS Organization: Supervisor Calls



Problem 1.

In lecture we arrived at the following implementation for the ReadKey supervisor call, which waits until there is a character available in the keyboard buffer, then returns it to the user in R0. Three lines in the handler have been labeled [A], [B], and [C].

```
    ReadKey_h() {
      int kbdnum = ProcTbl[Cur].DPYNum;
      if (BufferEmpty(kbdnum)) {
[A]     User.Regs[XP] = User.Regs[XP]-4;
[B]     Scheduler();
      } else {
[C]     User.Regs[0] = ReadInputBuffer(kbdnum);
      }
    }
```

Below we'll consider the effect of removing each labeled line in turn. **Please choose one** of the following as the best characterization of the effect of removing the line:

1. Execution appears to halt as there is now a loop in Kernel mode.
2. Both the requesting process and other processes run as before.
3. The requesting process runs as before; other processes receive a smaller percentage of the CPU time.
4. The requesting process runs as before; other processes receive a larger percentage of the CPU time.
5. The requesting process receives an incorrect character.

(A) Line [A] is removed from the handler.

User resumes execution although no character was available.

Effect on execution? (circle one) 1 … 2 … 3 … 4 … (5)

(B) Line [B] is removed from the handler.

Keep re-trying Readkey although no character is available.

Effect on execution? (circle one) 1 … 2 … (3) … 4 … 5

(C) Line [C] is removed from the handler.

Forgot to put character into user's R0!

Effect on execution? (circle one) 1 … 2 … 3 … 4 … (5)

(D) A summer intern decides that if a process is waiting for a character it should be scheduled for execution half as often, so he adds a second call to `Scheduler()`, i.e., he duplicates line [B]. Briefly describe the actual effect of this change. To be concrete assume there are N processes and that it's process 0 that executes the ReadKey SVC.

Brief description

Process #1 is never scheduled while Process #0 is waiting for a character.

Problem 2.

A Beta running the OS from lab 8 is running two processes:

```
// Process 0:              // Process 1:
P0:   GetKey()             P1:   ADDC(R0, 1, R0)
      WrCh()
      BR(P0)                     BR(P1)
```

XP always points here when execution resumes.

You type a few characters and see them echoed.

(A) What can you say about the value in the XP register on return from the **`GetKey()`** SVC?

It may have different values on consecutive returns: TRUE **FALSE**

Since the last statement is true, all the other statements are also true.

It always has a high-order 0 bit: **TRUE** FALSE

It is always a multiple of 4: **TRUE** FALSE

It always has the value P0+4: **TRUE** FALSE

You notice that Process 1 increments R0, whose value increases at some fairly constant rate **R** increments/second. You experiment with several changes below, not typing but monitoring the rate at which the count in Process 1's R0 increases. For each of the experiments below, you are asked to estimate its effect on the R0 counting rate, relative to **R**; choose among

+LOTS: the rate increases considerably, e.g., twice **R**.
SAME: the rate is about the same as **R**
-LOTS: the rate is much lower, e.g., **R/2** or lower.

(B) As an experiment, you eliminate Process 0 (so that only Process 1 is running). How does the rate of increase of Process 1's R0 change? Assume the scheduler time slice for each process is long compared to one iteration of either loop.

P0 usually calls Scheduler (mostly ~~[illegible]~~ there's no character to return). So running R0 takes very little time.

Circle one: +LOTS **SAME** -LOTS

You restore both processes and *eliminate the Scheduler() call* invoked by the GetKey() SVC when no character is ready to be returned.

(C) How does this effect the rate at which R0 increases, relative to the original counting rate **R**?

P0 keeps looping, calling GetKey. Only yields to P1 at end of timesharing slice.

Circle one: +LOTS SAME **-LOTS**

(D) Again running both Process 0 and Process 1, you now replace the single Scheduler() call invoked by GetKey() by **two consecutive** Scheduler() calls. How does the rate at which Process 1 counts change, relative to the original rate **R**?

See 1(C)!

Circle one: +LOTS SAME **-LOTS**

Problem 3.

Real Virtuality, Inc. markets three different computers, each with its own operating system. The systems are:

Model A: A timeshared Beta system whose OS kernel is uninterruptable.
Model B: A timeshared Beta system which enables device interrupts during handling of SVC traps.
Model C: A single-process (not timeshared) system which runs dedicated application code.

Each system runs an operating system that supports concurrent I/O on several devices, including an operator's console with a keyboard. Les N. Dowd, RVI's newly-hired OS expert, is in a jam: he has dropped the shoebox containing the master copies of OS source for all three systems. Unfortunately, three disks containing handlers for the ReadKey SVC trap, which reads and returns the ASCII code for the next key struck on the keyboard, have gotten confused. Of course, they are unlabeled, and Les isn't sure which handler goes into the OS for which machine. The handler sources are

```
ReadCh_h() {                                /* VERSION R1 */
  if (BufferEmpty(0))                       /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;      /* Nope, wait.       */
  else
    User->Regs[0] = ReadInputBuffer(0); /* Yup, return it. */
}

ReadCh_h() {                                /* VERSION R2 */
  int kbdnum=ProcTbl[Cur].KbdNum;
  while (BufferEmpty(kbdnum)) ;             /* Wait for a key to be hit*/
  User->Regs[0] = ReadInputBuffer(kbdnum);  /*...then return it. */
}

ReadCh_h() {                                /* VERSION R3 */
  int kbdnum=ProcTbl[Cur].KbdNum;
  if (BufferEmpty(kbdnum)) {                /* Has a key been typed? */
    User->Regs[XP] = User->Regs[XP]-4;      /* Nope, wait. */
    Scheduler();
  } else
    User->Regs[0] = ReadInputBuffer(kbdnum);   /* Yup, return it. */
}
```

(A) Show that you're cleverer than Les by figuring out which handler goes with each OS, i.e., for each operating system (A, B and C) indicate the proper handler (R1, R2 or R3).

Model A goes with handler (circle one): R1 ... R2 ... R3

Model B goes with handler (circle one): R1 ... R2 ... R3

Model C goes with handler (circle one): R1 ... R2 ... R3

But Les isn't that smart. In order to figure out which handler code goes with each OS version, Les makes copies of each disk and distributes them as "updates" to beta-test teams for each OS. Les figures that if each handler version is tried by some beta tester in each OS, the comments of the testers will allow him to determine the proper OS for each handler.

Les sends out the alleged source code updates, routing each handler source to testers for each OS. In response, he gets a barrage of complaints from many of the testers. Of course, he's forgotten which disk he sent to each tester. He asks your help to figure out which combination of system and hander causes each of the complaints.

For each complaint below, explain which handler and which OS the complainer is trying to use.

(B) Complaint: "I get compile-time errors; Scheduler and ProcTbl are undefined!"

User has handler R3 **on system** C

(C) Complaint: "Hey, now the system always reads everybody's input from keyboard 0. Besides that, it seems to waste a lot more CPU cycles than it used to."

User has handler R1 **on system** A

(D) (Complaint: "Neat, the new system seems to work fine. It even seems to waste less CPU time than it used to!"

User has handler R3 **on system** B

Problem 4.

The Yield() SVC can be used in user-mode programs on a time-sharing system to give up the remainder of their current time slice. The kernel implementation of the Yield simply calls the kernel Scheduler() routine to choose another process to execute. When the yielding process is next scheduled, user-mode execution resumes with the instruction following the Yield() SVC.

Complete the code for the handler for a new SVC, YieldN(), which expects a numeric value, N, in the user's R0 and behaves as if the user program had contained N consecutive Yield() SVCs. When execution resumes following the completion YieldN(), R0 should contain 0.

```
YieldN_h() {
    if (User.Regs[0] > 0) {
        User.Regs[0] -= 1;    // decrement count
        User.Regs[XP] -= 4;   // arrange to re-execute SVC
        Scheduler();          // yield to next process
    } else {

        User.Regs[0] = 0;     // [optional] final value for R0

    }
}
```

Problem 5.

BetaSoft, Inc, the leading provider of Beta OS software, sells an operating system for the Beta similar to that described in lecture. It uses a simple round-robin scheduler, and has no virtual memory -- all processes share a single address space with the kernel, much like the OS of lab 8. The OS timeshares the Beta CPU among N processes using a simple, familiar scheduler shown below.

```
struct MState { int Regs[31]; } User;

struct PCB {              // Process Descriptor Block
  struct MState State;    // Saved process state
  ...;                    // Possible other stuff
} ProcTbl[N];             // One PCB per process

int Cur = 0;

Scheduler() {
  ProcTbl[Cur].State = User;
  Cur = (Cur+1) % N;
  User = ProcTbl[Cur].State;
}
```

Several of Betasoft's customers use the Beta for long, compute-bound applications, and have asked for a tool to help them find where their programs are spending most of their time. To accommodate these requests, BetaSoft has implemented a supervisor call, **SamplePC**, which allows a diagnostic program running in one process to sample the values in the PC of another. Betasoft proposes to write such a program, called a profiler, that takes many samples of PC values from a running program and produces a revealing histogram.

The SamplePC SVC takes a process number p in R0, and returns in R1 the value currently in the program counter of process p. The C portion of the SVC handler is given below:

```
SamplePC_h() {
  int p = User.Regs[0];
  int pc = ProcTbl[p].State.Regs[XP];
  ??? = pc;                          // incomplete code!
}
```

(A) Give the missing code fragment shown above as **???**.

User.Regs[1]

(write missing fragment of C code)

BetaSoft writes a simple profiler using the above SVC and uses it to measure a compute-bound process consisting of a single 10000-instruction loop. Noticing a surprisingly large number of repeated values in the sampled PC data, they cleverly deduce that their profiler is making many **SamplePC** calls during each time quanta for which the profiling process is scheduled.

returning redundant samples from the process being measured.

(B) Suggest a simple change to the **SamplePC_h** code that eliminates the observed problem. Be specific.

add Scheduler() call

(Describe simple modification to SamplePC_h code)

BetaSoft ignores your solution (keeping the original **SamplePC_h** code), arguing that they'll just collect enough samples that the redundant values won't affect the histogram significantly. They produce a working profiler program that takes many samples of another process's PC and produces a histogram showing code "hot spots". Although the profiler proves useful on compute-intensive application code, BetaSoft tries running it on a simple echo loop running in a process:

```
| echo loop, as a test for profiler tool:
.=0x100              | Test program starts at hex 100
loop:  GetKey()      | SVC: read char into R0
       WrCh()        | SVC: type char from R0
       BR(loop)      | ... and keep doing it!
```

(C) When run on the above process, what does the profiler report as the most common value of the PC? Answer "None" if you can't tell.

Often-reported PC value, or "None": 0x 100

The final BetaSoft profiler program itself is mostly a big loop, consisting of a single SamplePC SVC instruction located at **0x1000**, plus lots of compute-intensive additional code to appropriately format the collected data and write it into a file. Out of curiosity, BetaSoft engineers run the profiler in process 0, and ask it to generate a histogram of sampled PC values for process 0 itself.

(D) Which of the following best summaries their findings?

(1) All of the sampled PC values point to kernel OS code.
(2) The sampled PC is always **0x1004**.
(3) The SamplePC call never returns.
(4) None of the above.

Give number of best answer: 4

SamplePC returns XP saved in the process table, which is only updated at the call to Scheduler when switching to next process.

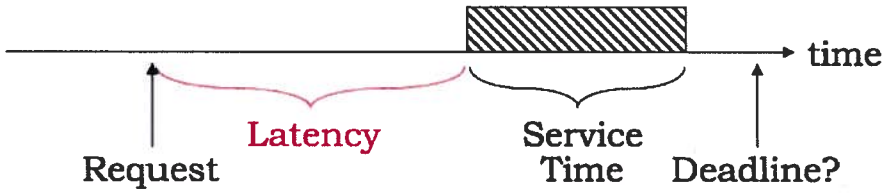
→ It does not read User.MState.

6.004 Computation Structures
Spring 2017

Computation Structures

Interrupts and Real Time Worksheet

Latency (L) = elapsed time before handler code starts to execute
Service time (S) = time required to execute handler code
Deadline (D) = maximum time after request by when handler execution must be *complete*
Maximum allowable latency (Lmax) = largest L such that Lmax + S = D.



Handler priority: when choosing which handler to run, choose the handler with the highest priority. Priorities are chosen to ensure that the deadlines for all handlers will be met.

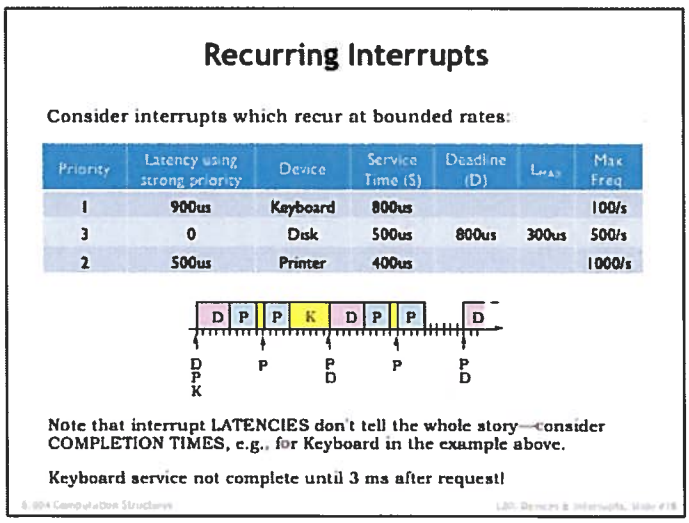
Recurring requests: often requests will occur at some fixed interval ≥ deadline.

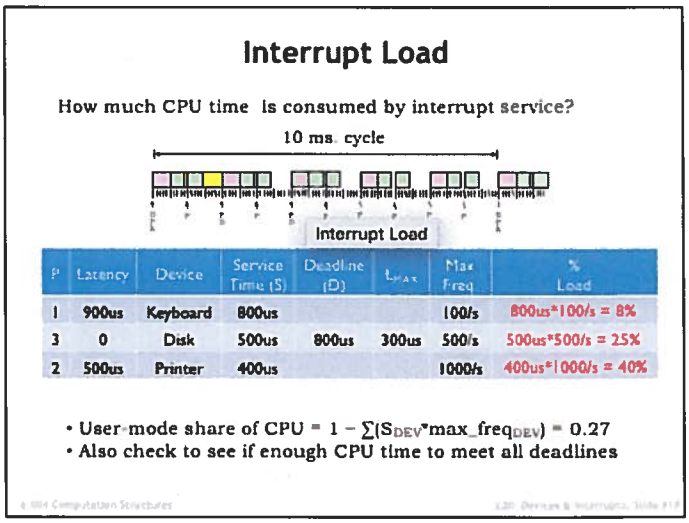
Earliest deadline is a strategy for assigning priorities that is guaranteed to meet the deadlines if any priority assignment can meet the deadlines:

1. Sort the requests by their deadlines
2. Assign the highest priority to the earliest deadline, second priority to the next deadline, and so on.

Weak (non-preemptive) priorities: after current handler completes, look through the pending requests and execute the handler with the highest priority. Latencies with weak priorities: service of each device might be delayed by (1) service of one other (arbitrary) device whose request was just honored, and (2) service of all higher-priority tasks.

Strong (preemptive) priorities: always run the pending handler with the highest priority, possibly interrupting the execution of a lower-priority handler to do so. Latencies with weak priorities: service of each device might be delayed by service of all (possibly recurring) higher-priority tasks.





Problem 1.

Ben Bitdiddle has designed a wrist device called the BenBit to measure how long you've been tooling away without getting up and moving around. The BenBit runs a real-time operating system supporting three tasks whose handlers are executed in response to periodic requests. Each task has a period (time between requests), a service time (time it takes to run its handler), and a deadline (maximum time allowed to elapse between request and completion of the handler).

Task	*Period (ms)*	*Service time (ms)*	*Deadline (ms)*
Check Accelerometer (CA)	80	20	40
Update Display (UD)	200	?	200
Determine heart rate (DHR)	60	10	50

Ben is trying to figure out whether to use a weak or strong priority system to manage task execution. For each of the questions below, please fill the answers for both types of priority system. For both the weak and strong priority system **assume the task priority is CA > DHR > UD**, i.e., CA has the highest priority and UD the lowest. If a calculation requires the service time for UD, use your answer for part (A).

	Using Weak Priorities	**Using Strong Priorities**
(A) What is the maximum service time for UD handler that still allows all deadlines to be met (in ms)?	20 ms otherwise CA will miss deadline	100ms $= 200 - 3*20 - 4*10$
(B). What fraction of the time will the processor spend idle? (%)	48.33% $= 1 - (\frac{20}{80}) - (\frac{20}{200}) - (\frac{10}{60})$	8.33% $= 1 - (\frac{20}{80}) - (\frac{100}{200}) - (\frac{10}{60})$
(C) What is the worst-case completion time for the CA task (in ms)?	40 ms $= UD_{ST} + CA_{ST}$	20 ms $= CA_{ST}$
(D) What is the worst-case completion time for the UD task (in ms)?	50 ms $= DHR_{ST} + CA_{ST} + UD_{ST}$	200 ms $= 3 \cdot CA_{ST} + 4 \cdot DHR_{ST} + UD_{ST}$
(E) What is the worst-case completion time for the DHR task (in ms)?	50 ms $= UD_{ST} + CA_{ST} + DHR_{ST}$	30 ms $= CA_{ST} + DHR_{ST}$

Problem 2.

A particular real-time system has three interrupt handlers. The following table shows the maximum rate at which each interrupt occurs (rate), the time taken to execute each handler (service time), and the maximum allowable interval between the interrupt and completion of the handler (deadline). In your analysis, assume that A, B, and C interrupts can arrive at any time.

% CPU
10/20 = 50
10/80 = 12.5
5/25 = 20

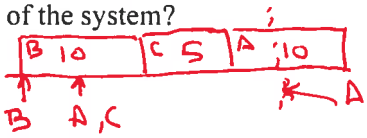
Task	*Rate*	*Service time*	*Deadline*
A	1/20ms	10ms	20ms
B	1/80ms	10ms	80ms
C	1/25ms	5ms	25ms

(A) What is the percentage idle time for this system?

100% - (50 + 12.5 + 20)%

Percentage Idle Time (%): 17.5 %

(B) Assuming a **weak** priority system. Can the priority ordering C > A > B satisfy **all** the constraints of the system?



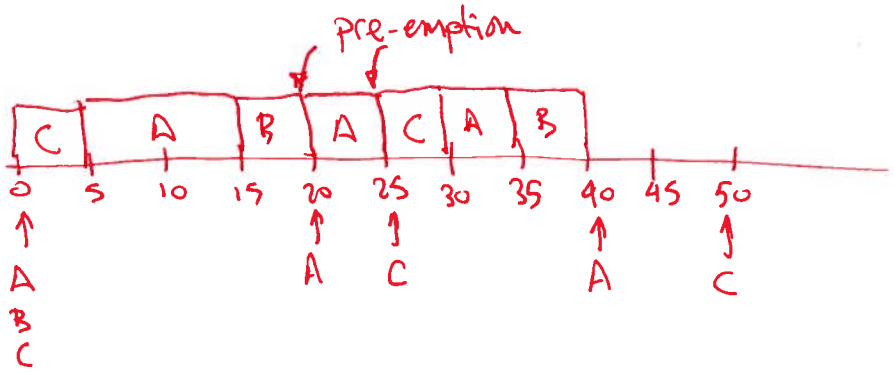
C > A > B (Yes/No): NO

(C) Assuming a **strong** priority system with priority **C > A > B**, what is the worst-case delay between the task's interrupt and **completion** of the task's interrupt handler?

Worst-case delay for task A (ms): 15

Worst-case delay for task B (ms): 40

Worst-case delay for task C (ms): 5



Problem 3.

A particular real-time system has three interrupt handlers. The following table shows the maximum rate at which each interrupt occurs (rate), the time taken to execute each handler (service time), and the maximum allowable interval between the interrupt and completion of the handler (deadline). In your analysis, assume that A, B, and C interrupts can arrive at any time.

Task	*Rate*	*Service time*	*Deadline*
A	1/40ms	5ms	30ms
B	1/100ms	?ms	100ms
C	1/50ms	10ms	25ms

Please answer the following two questions assuming the system has a **weak** priority system.

(A) (1 point) What is the maximum service time for task B that still allows all constraints to be met?

Task C has max latency of 15 ms → limits service time of B.

Maximum service time for task B (ms): 15 if C>A>B in part (B); 10 if A>C>B in part (B)

(B) (1 point) Assuming a suitable service time for B, give a weak priority order for the tasks that meets the constraints. List the task with the highest priority first, the task with the lowest priority last.

earliest deadline first! **Weak priority order for the tasks:** C > A > B

Please answer the following two questions assuming the system has a **strong** priority system where task C has the highest priority and task B has the lowest priority.

(C) (2 points) What is the maximum service time for task B that still allows all constraints to be met?

in 100 ms worst case:
3 task A requests (15 ms total)
2 task C requests (20 ms total)
35 ms service time ⇒ 100 − 35 = 65 ms available for B

Maximum service time for task B (ms): 65

(D) (3 points) Assume B's sevice time is 10ms. For each task, what is the worst-case delay between the task's interrupt and **completion** of the task's interrupt handler?

priority C > A > B

wait for C + A service time = 10 + 5

wait for C, A + B service time = (10 + 5) + 10 = 25

Worst-case delay for task A (ms): 15

Worst-case delay for task B (ms): 25

Worst-case delay for task C (ms): 10

↳ starts immediately = C service time

Problem 4.

A computer system has three devices whose characteristics are summarized in the following table:

Device	Service time	Interrupt Frequency	Deadline
D1	400us	1/(800us)	800us
D2	250us	1/(1000us)	300us
D3	100us	1/(800us)	400us

Service time indicates how long it takes to run the interrupt handler for each device. The maximum time allowed to elapse between an interrupt request and the end of the execution of the interrupt handler is indicated by the deadline.

A. If a user-mode program P takes 100 seconds to execute when interrupts are disabled, approximately how long will P take to run when interrupts are enabled?

Approximate time for P to run with interrupts enabled (seconds): 8·100 = 800

D1: 400/800 = 50% of CPU time
D2: 250/1000 = 25%
D3: 100/800 = 12.5%

} only 12.5% = 1/8 of cpu time available for running P.

B. Can the requirements given in the table above be met using a **weak** priority ordering among the interrupt requests? If so give priority ordering for D1, D2, D3 or list device(s) whose deadlines cannot be met.

Weak priority ordering or list device(s) with missed deadlines: D2, D3

if D1 is running, D2 and D3 may miss deadlines since D2 max latency is 50us and D3 max latency is 300us.

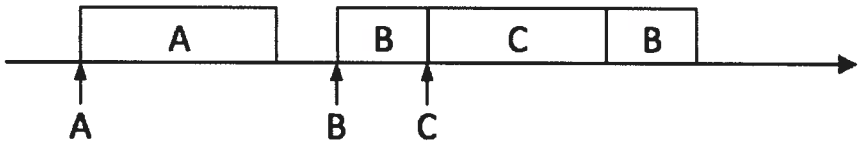
(C) Can the requirements given in the table above be met using a **strong** priority ordering among the interrupt requests? If so give priority ordering for D1, D2, D3 or list device(s) whose deadlines cannot be met.

Strong priority ordering or list device(s) with missed deadlines: D2 > D3 > D1

use earliest deadline strategy!

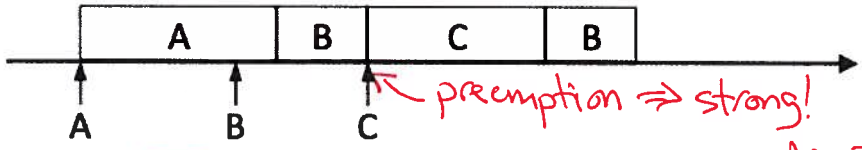
Problem 5.

A real-time operating system with priority interrupts has three interrupt handlers – A, B, C – each running at a different priority level. The handlers are invoked by the A, B, and C interrupts, marked as ↑ in the execution timelines. For example, the following execution timeline shows the A handler running to completion after an A interrupt request, followed by execution of the B handler, which is interrupted by execution of the C handler.



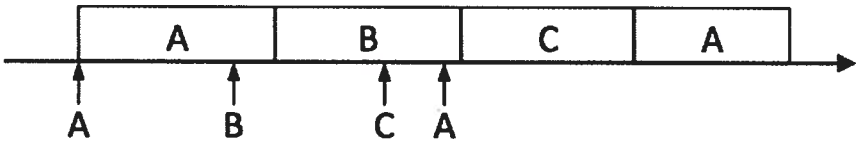
For each of the following execution timelines, please indicate whether the system is using a WEAK or STRONG priority scheme, or CAN'T TELL if the timeline is consistent with either WEAK or STRONG. Also, if WEAK or STRONG, indicate any relative priorities that can be deduced from the timeline (there may be more than one), expressed as inequalities. For example, A > B indicates A has a higher priority than B.

(A)



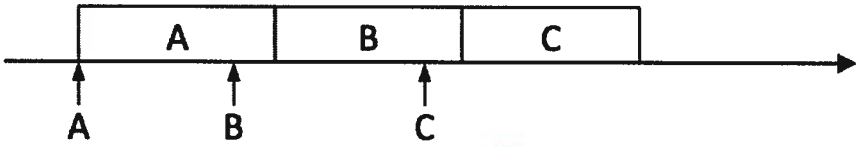
Circle one: (STRONG) ... WEAK ... CAN'T TELL, **Priorities:** A > B, C > B

(B)



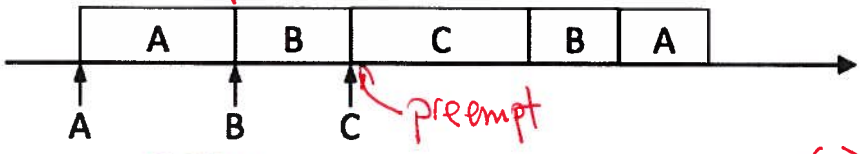
Circle one: STRONG ... (WEAK) ... CAN'T TELL, **Priorities:** C > A

(C)



Circle one: STRONG ... WEAK ... (CAN'T TELL), **Priorities:** —

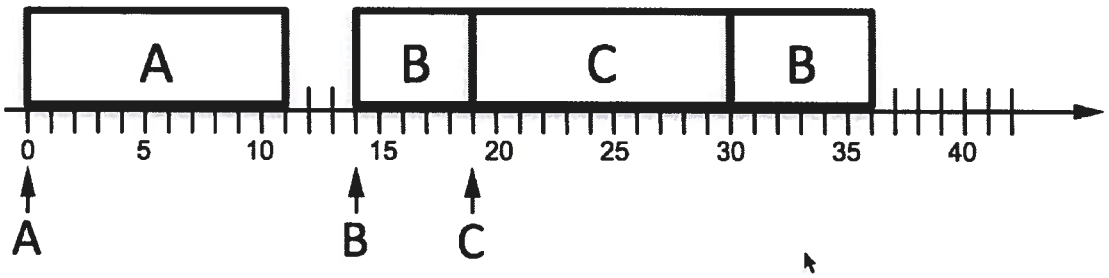
(D)



Circle one: (STRONG) ... WEAK ... CAN'T TELL, **Priorities:** C > B > A

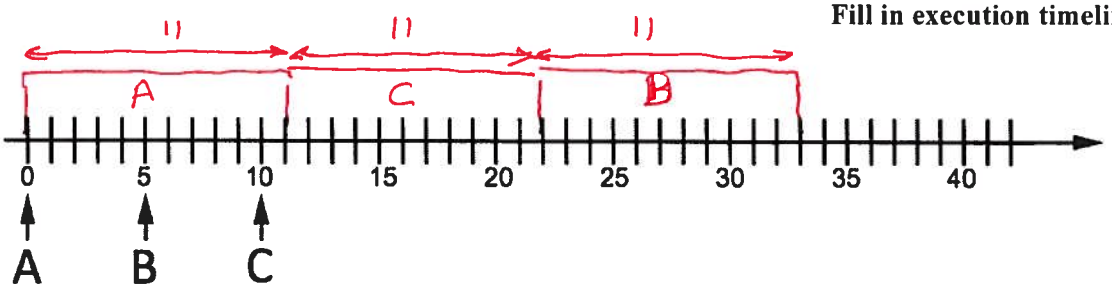
Problem 6.

A real-time operating system with priority interrupt has three interrupt handlers (A, B, C), each of which, when invoked by the appropriate interrupt request (marked as ↑ in the execution timelines), takes 11 time units to execute. For example, the following execution timeline shows the A handler running to completion after an A interrupt request, followed by execution of the B handler, which is itself interrupted by execution of the C handler.



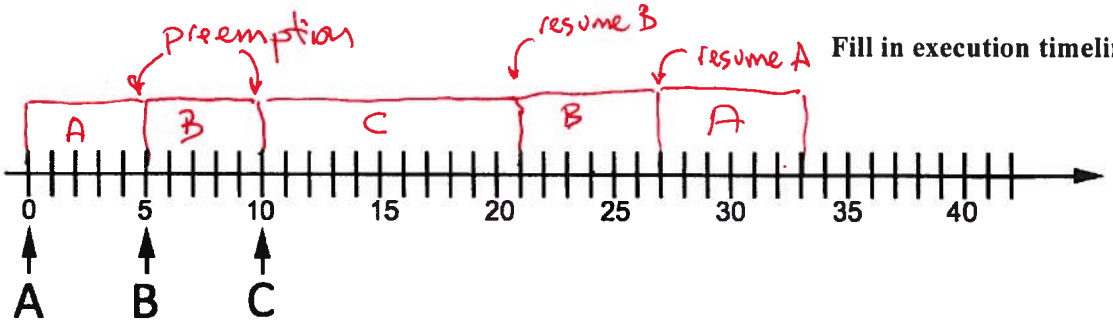
(A) The execution timeline below shows the arrival times of interrupt requests for A, B, and C. Diagram the execution of the A, B, and C handlers assuming a **weak** priority system with the priorities **C > B > A**. Remember to show the complete execution (all 11 time units) for each handler, labeling each block with the handler that is running.

Fill in execution timeline



(B) The execution timeline below shows the arrival times of interrupt requests for A, B, and C. Diagram the execution of the A, B, and C handlers assuming a **strong** priority system with the priorities **C > B > A**. Remember to show the complete execution (all 11 time units) for each handler, labeling each block with the handler that is running.

Fill in execution timeline



6.004 Computation Structures
Spring 2017

Computation Structures

Synchronization Worksheet

Semaphores (Dijkstra)

Programming construct for synchronization:

- NEW DATA TYPE: *semaphore*, an integer ≥ 0

```
semaphore s = K;  // initialize s to K
```

- NEW OPERATIONS (defined on semaphores):
  - `wait(semaphore s)`
    wait until s > 0, then s = s – 1
  - `signal(semaphore s)`
    s = s + 1 (one WAITing process may now be able to proceed)
- SEMANTIC GUARANTEE: A semaphore s initialized to K enforces the constraint:

Often you will see P(s) used for wait(s) and V(s) used for signal(s)! P = "proberen"(test) or "pakken"(grab) V= "verhogen"(increase)

$$signal(s)_i < wait(s)_{i+K}$$

This is a precedence relationship: the i^{th} call to signal must complete before the the $(i+K)^{th}$ call to wait will succeed.

Semaphores for Precedence

```
semaphore s = 0;

Process A        Process B
   A1;              B1;
   A2;              B2;
signal(s);
   A3;              B3;
                    wait(s);
   A4;              B4;
   A5;              B5;
```

Goal: want statement A2 in process A to complete before statement B4 in Process B begins.

A2 < B4

Recipe:

- Declare semaphore = 0
- `signal(s)` at start of arrow
- `wait(s)` at end of arrow

Semaphores for Resource Allocation

Abstract problem:

- POOL of K resources
- Many processes, each needs resource for occasional uninterrupted period
- MUST guarantee that at most K resources are in use at any time.

Semaphore Solution:

In shared memory:

```
semaphore s = K;  // K resources
```

Using resources:

```
wait(s);     // Allocate a resource
 …           // use it for a while
signal(s);   // return it to pool
```

Invariant: Semaphore value = number of resources left in pool

Semaphores for Mutual Exclusion

```
semaphore lock = 1;

Debit(int account, int amount) {
  wait(lock);   // Wait for exclusive access
  t = balance[account];
  balance[account] = t - amount;
  signal(lock); // Finished with lock
}
```

a ◇ b

"a *precedes* b or b *precedes* a" (i.e., they don't overlap)

RESOURCE managed by "lock" semaphore:
Access to critical section

ISSUES:

Granularity of lock
1 lock for whole balance database?
1 lock per account?
1 lock for all accounts ending in 004

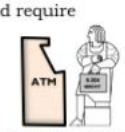
Look up "database" on Wikipedia to learn about systems that support efficient transactions on shared data.

Dealing With Deadlocks

Cooperating processes:

- Establish a fixed ordering to shared resources and require all requests to be made in the prescribed order

```
Transfer(int account1, int account2, int amount) {
  int a = min(account1, account2);
  int b = max(account1, account2);
  wait(lock[a]);
  wait(lock[b]);
  balance[account1] = balance[account1] - amount;
  balance[account2] = balance[account2] + amount;
  signal(lock[b]);
  signal(lock[a]);
}
```



Transfer(6005, 6004, 50)

Transfer(6004, 6005, 50)

Unconstrained processes:
- O/S discovers circular wait & kills waiting process
- Transaction model
- Hard problem

Summary

Communication among parallel threads or asynchronous processes requires synchronization....

- Precedence constraints: a partial ordering among operations
- Semaphores as a mechanism for enforcing precedence constraints
- Mutual exclusion (critical sections, atomic transactions) as a common compound precedence constraint
- Solving Mutual Exclusion via binary semaphores
- Synchronization *serializes* operations, limits parallel execution

Many alternative synchronization mechanisms exist!

Deadlock:

- Consequence of undisciplined use of synchronization mechanism
- Can be avoided in special cases, detected and corrected in others

Problem 1.

Schro Dinger has a company that produces pairs of entangled particles, which are then packaged and sent to manufacturers of quantum computers. Since it's a complicated process, there are multiple machines that produce particle pairs; each machine runs the Producer code shown below.

The completed particle pairs are placed in the particle buffer, where they take up 2 of the buffer locations. There's a single packaging machine that takes a particle pair from the particle buffer and prepares it for shipment; the packing machine runs the Consumer code shown below.

To prevent any violations of the boundary conditions the following rules must be followed:

1. A production machine can only place a particle pair in the buffer if there are two spaces available.
2. The particle pair must be stored in consecutive buffer locations, i.e., a particle from some other production machine can't appear between the particles that make up the pair.
3. The capacity of the buffer (100 particles, or 50 particle pairs) can't be exceeded.
4. The packaging machine breaks if it accesses the buffer and finds it empty – it should only proceed when there are at least two particles in the buffer.

Schro has heard of semaphores but is unsure how to use them to ensure the rules are followed.

- Please insert the appropriate semaphores, WAITs, and SIGNALs into the Producer and Consumer code to ensure correct operation and to prevent deadlock.
- Be sure to indicate initial values for any semaphores you use.
- Remember: **there are multiple producers and a single consumer**!
- For full credit, use a minimum number of semaphores and don't introduce unnecessary precedence constraints.

```
Shared Memory
particle buffer[100];  // holds 100 particles

Semaphores and initial values: P=0, S=50, M=1
```

```
Producer
PLoop:

   Produce pair P1, P2
   WAIT(S); WAIT(M)
   Place P1 in buffer

   Place P2 in buffer
   SIGNAL(M); SIGNAL(P)
   Go to PLoop
```

```
Consumer
CLoop:
   WAIT(P)
   Fetch P1 from buffer

   Fetch P2 from buffer
   SIGNAL(S)
   Package and ship

   Go to CLoop
```

P = # of particle pairs in buffer, enforces rule 4
S = # of pair spaces in buffer, enforces rules 1 and 3
M = mutual exclusion lock, enforces rule 2 (when there are multiple producers)

Problem 2.

The following three processes are run on a shared processor. They can coordinate their execution via shared semaphores that respond to the standard signal(S) and wait(S) procedures. Their intent is to print the word HELLO. Assume that execution may switch between any of the three processes at any point in time.

```
Process 1                    Process 2                    Process 3

Loop1: print("H")            Loop2: print("L")            Loop3: print("O")
       print("E")                   goto Loop2                   goto Loop3
       goto Loop1
```

(A) Assuming that no semaphores are being used, for each of the following sequences of characters, specify whether or not this system could produce that output.

LEHO (YES/NO): No **HLOE (YES/NO):** Yes **LOL (YES/NO):** Yes

Need H before E

(B) You would like to ensure that only the sequence HELLO can be printed and that it will be printed exactly once. Add any missing wait(S) and signal(S) calls to the code below (where S is one of a, b or c) to ensure that the three processes can only print HELLO exactly once. Remember to specify the **initial value** for each of your semaphores. *Recall that semaphores cannot be initialized to negative numbers.*

Semaphores: a = 1; b = 0; c = 0;

```
Process 1                    Process 2                    Process 3

Loop1:                       Loop2:                       Loop3:

    wait(a)                      wait(b)                      wait(c)
                                                              wait(c)
    print("H")                   print("L")                  print("O")

    print("E")                   signal(c)

    signal(b)
    signal(b)
    goto Loop1                   goto Loop2                   goto Loop3
```

Problem 3.

The following pair of processes share the variable `counter`, which has been given an initial value of 10 before execution of either process begins:

```
Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)
    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
A2: ST(R0,counter)                 B2: ST(R0,counter)
…                                  …
```

(A) If Processes A and B are run on a timesharing system, there are six possible orders in which the LD and ST instructions might be executed. For each of the orderings, please give the final value of the counter variable.

A1 A2 B1 B2: counter = 13 **B1 A1 B2 A2: counter =** 11

A1 B1 A2 B2: counter = 12 **B1 A1 A2 B2: counter =** 12

A1 B1 B2 A2: counter = 11 **B1 B2 A1 A2: counter =** 13

In the following two questions you are asked to modify the original programs for processes A and B by adding the minimum number of semaphores and signal and wait operations to guarantee that the final result of executing the two processes will be a specific value for counter. Give the initial values for every semaphore you introduce. For full credit, your solution should allow *all* execution orders that result in the required value.

(B) Add semaphores (with initial values) so that the final value of counter is 12.

Semaphores: X=0, Y=0

```
Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)

    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait(x)                            signal(x); wait(y)
A2: ST(R0,counter)                 B2: ST(R0,counter)
…   signal(y)                      …
```

(C) Add semaphores (with initial values), so that the final value of counter is **not** 13.

Semaphores: X=0, Y=0

```
Process A                          Process B
…                                  …
A1: LD(counter,R0)                 B1: LD(counter,R0)
    signal(x)                          signal(y)
    ADDC(R0,1,R0)                      ADDC(R0,2,R0)
    wait(y)                            wait(x)
A2: ST(R0,counter)                 B2: ST(R0,counter)
…                                  …
```

Problem 4.

P1 and P2 are processes that run concurrently. P1 has two sections of code where section A is followed by section B. Similarly, P2 has two sections: C followed by D. Within each process execution proceeds sequentially, so we are guaranteed that A $\preceq$ B, i.e., A precedes B. Similarly, we know that C $\preceq$ D. There is no looping; each process runs exactly once. You will be asked to add semaphores to the programs – you may need to use more than one semaphore. Please give the initial values of any semaphores you use. For full credit use a minimum number of semaphores and don't introduce any unnecessary precedence constraints.

(A) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that the precedence constraint B $\preceq$ C is satisfied, i.e., execution of P1 finishes before execution of P2 begins.

Add WAIT and SIGNAL statements so that B $\preceq$ C

Semaphore initial values: S=0

Process P1	Process P2
	wait(S)
…Section A code…	… Section C code…
…Section B code…	…Section D code…
signal(S)	

(B) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that D $\preceq$ A *or* B $\preceq$ C, i.e., executions of P1 and P2 cannot overlap, but are allowed to occur in either order.

Add WAIT and SIGNAL statements so that D $\preceq$ A *or* B $\preceq$ C

Semaphore initial values: M=1

Process P1	Process P2
wait(M)	wait(M)
…Section A code…	… Section C code…
…Section B code…	…Section D code…
signal(M)	signal(M)

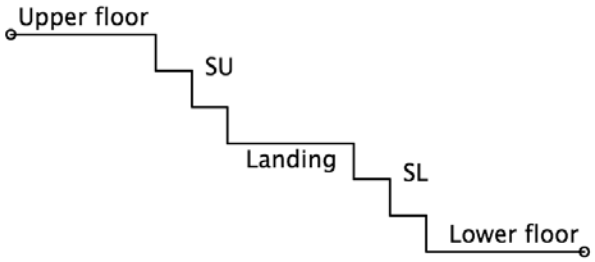
(C) Please add WAIT(…) and SIGNAL(…) statements as needed in the spaces below so that A ≺ D *and* C ≺ B, i.e., the first section (A and C) of **both** processes completes execution before the second section (B or D) of **either** process begins execution.

Add WAIT and SIGNAL statements so that A ≺ D *and* C ≺ B

Semaphore initial values: S=0, T=0

Process P1	Process P2
…Section A code…	… Section C code…
signal(S)	signal(T)
wait(T)	wait(S)
…Section B code…	…Section D code…

Problem 5.



The MIT Safety Office is worried about congestion on stairs and has decided to implement a semaphore-based traffic-control system. Most connections between floors have two flights of stairs with an intermediate landing (see figure). The constraints the Safety Office wishes to enforce are

- Only 1 person at a time on each flight of stairs
- A maximum of 3 persons on a landing
- As a few traffic constraints as possible
- No deadlock (a particular concern if there's bidirectional travel)

Assume stair traffic is unidirectional: once on a flight of stairs, people continue up or down until they've reached their destination floor (no backing up!), although they may pause at the landing.

There are three semaphores: they control the upper flight of stairs (SU), the landing (L), and the lower flight of stairs (SL). Please provide appropriate initial values for these semaphores and add the necessary wait() and signal() calls to the Down() and Up() procedures below. Note that the Down() and Up() routines will be executed by many students simultaneously and the semaphores are the only way their code has of interacting with other instances of the Down() and Up() routines. To get full credit your code must avoid deadlock and enforce the stair and landing occupancy constraints. **Hint**: for half credit, implement a solution where only 1 person at time is in-between floors (but be careful of deadlock here too!).

```
// Semaphores shared by all students, provide initial values
semaphore SU = 1, SL = 1, L = 3;
```

```
// code for going downstairs
Down() {
    wait(L)
    wait(SU)
    Enter SU;

    Exit SU/enter landing;
    signal(SU)
    wait(SL)
    Exit landing/enter SL;
    signal(L)
    Exit SL;
    signal(SL)
}
```

```
// code for going upstairs
Up() {
    wait(L)
    wait(SL)
    Enter SL;

    Exit SL/enter landing;
    signal(SL)
    wait(SU)
    Exit landing/enter SU;
    signal(L)
    Exit SU;
    signal(SU)
}
```

Problem 6.

(A) Semaphore S is used to implement mutual exclusion on accesses to a shared buffer. No other semaphores are used. What should its initial value be?

A value of 1 for S allows at most one WAIT to succeed – others will stall until first one SIGNALs. This implements mutual exclusion.

Initial value for S: 1

(B) Indicate whether each of the following sets of semaphore-synchronized processes can deadlock. The last two cases are variants of the first one; differences are *underlined*.

Circle answers below

```
Initial semaphore values: s1 = 1, s2 = 1, s3 = 1
P1:                 P2:                 P3:
wait(s1);           wait(s2);           wait(s1);
wait(s2);           wait(s3);           wait(s2);
print("1");         print("2");         wait(s3);
signal(s2);         signal(s3);         print("3");
signal(s1);         signal(s2);         signal(s3);
                                        signal(s2);
                                        signal(s1);
```

Uses a global ordering for semaphores (S1 > S2 > S3). No deadlock possible.

Can it deadlock?

YES **NO** **Can't tell**

```
Initial semaphore values: s1 = 1, s2 = 1, s3 = 1
P1:                 P2:                 P3:
wait(s1);           wait(s2);           wait(s2);
wait(s2);           wait(s3);           wait(s3);
print("1");         print("2");         wait(s1);
signal(s2);         signal(s3);         print("3");
signal(s1);         signal(s2);         signal(s1);
                                        signal(s3);
                                        signal(s2);
```

Deadlock scenario:
P1 holds S1, waiting on S2
P3 holds S2, waiting on S1

Can it deadlock?

YES **NO** **Can't tell**

```
Initial semaphore values: s1 = 2, s2 = 1, s3 = 1
P1:                 P2:                 P3:
wait(s1);           wait(s2);           wait(s2);
wait(s2);           wait(s3);           wait(s3);
print("1");         print("2");         wait(s1);
signal(s2);         signal(s3);         print("3");
signal(s1);         signal(s2);         signal(s1);
                                        signal(s3);
                                        signal(s2);
```

Now both WAIT(S1) statements will succeed.

Can it deadlock?

YES **NO** **Can't tell**

6.004 Computation Structures
Spring 2017