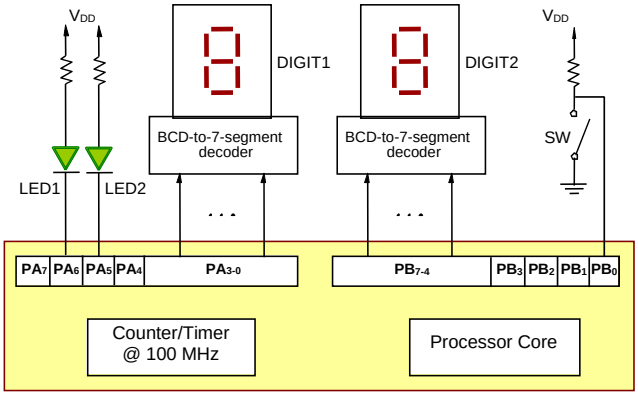
Parallel I/O			
Register	**Name**	**Description**	**Use**
PAIN	Port A input	Input data on PA	x= *PAIN; read PA
POUT	Port A output	Output data to PA	*PAIN = x; output to PA
PADIR	Port A direction	Set PA[x] to 0/1 input/output direction	PADIR = 0x00; set all pins as input
			PADIR = 0xFF; set all pins as output
PBIN	Port B input	Input data on PB	x= *PAIN; read PA
PBOUT	Port B output	Output data to PB	*PAIN = x; output to PA
PBDIR	Port B direction	Set PB[x] to 0/1 input/output direction	*PBDIR = 0x00; set all pins as input
			*PBDIR = 0xFF; set all pins as output
PSTAT	**Status register**		
	0 - PASIN	Set to 1 when new data is on PAIN. Cleared when PAIN is read	
	1 - PASOUT	Set to 1 when data in PAOUT is accepted by connected device. Clear when new data is written into PAOUT	
	2 - PBSIN	~	
	3 - PBSOUT	~	
	4 - IAIN	Set to 1 when an input interrupt due to new data on PAIN	
	5 - IAOUT	Set to 1 when an input interrupt due to PAOUT being ready for new data	
	6 - IBIN	~	
	7 - IBOUT	~	
PCONT	**Control register**		
	0 - PAREG	if 1 a buffer register is used between bins, if 0 then direct path to pin	
	1 - PBREG	~	
	4 - ENAIN	Enables interrupts when PASIN is raised (new data on PAIN)	
	5 - ENAOUT	Enables interrupts when PASOUT is raised (PASOUT ready for new data)	
	6 - ENBIN	Enables interrupts when PBSIN is raised (new data on PBIN)	
	7 - ENBOUT	Enables interrupts when PBSOUT is raised (PBSOUT ready for new data)	

Counter (Counts down)			
Register	**Name**	**Description**	**Use**
CNTM	Initial value	Sets initial value of counter	*CNTM = 0xFF; sets initial value to max value
COUNT	Counter contents	Contains counter value	x = *COUNT; real counter current value
CTSTAT	Status register	Bit 0 is set when counter reaches zero	
CTCON	**Control register**		
	0 - Start	Start the counter when set to 1 clear once counter starts	
	1 - Stop	Stop the counter when set to 1	
	4- Enable Interrupt	When set to 1 trigger interrupt when counter reaches zero in counter mode	
	7 - Counter/Timer	When set to 0 enter counter mode, counts down, when zero is hit, sets CTCON1 to 1 and raises interrupt if enabled. Reloads initial value into COUNT	
		When set to 1 enter timer mode, counts down, when zero is hit, invert Timer_out I/O bit and reload initial value (creates pulse wave)	

1. *[15 points]* The textbook's microcontroller is used in a system shown below and is responsible for two tasks: 1) decrementing either **DIGIT1** (when **LED1** is on) or **DIGIT2** (when **LED2** is on) every second, and 2) alternating between **LED1** and **LED2** being on, whenever the **SW** key is hit (i.e., pressed and then released). Write the corresponding C program, assuming that the **second task** is the main program, and the **first task** is an ISR whose address is stored at location **0x20**. Also, assume that bit **6** of the processor status register (i.e., **PSR[6]**) is the processor's interrupt-enable bit, and **Ports A** and **B** are always ready to be written by the processor. Initially, **LED1** is on, **LED2** is off, and both **DIGIT1** and **DIGIT2** show **0**.

• *Main Program*: Whenever $\mathbf{PB_0}$ first becomes 0 and then 1 again, **LED1** and **LED2** must swap their states: if **LED1** is on and **LED2** is off, then **LED1** becomes off and **LED2** becomes on, and vice versa. (Note: **LED1** and **LED2** should never be both on, or both off.)

• *ISR*: The 100-MHz Counter/Timer must be configured to generate interrupts every second. Its ISR must decrement **DIGIT1** if **LED1** is on, or decrement **DIGIT2** if **LED2** is on. (Note: decrementing **0** gives **9**.)

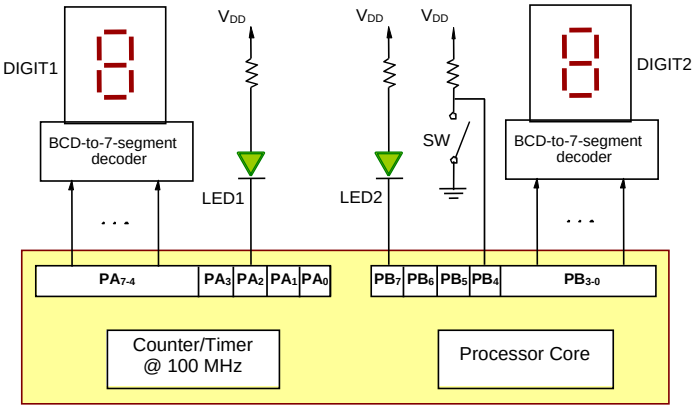



```
interrupt void intserv();

volatile unsigned char digit1 = 0; /* DIGIT1 for display */
volatile unsigned char digit2 = 0; /* DIGIT2 for display */
volatile unsigned char leds = 0x1; /* LED1 on, LED2 off */

int main() {
  *PADIR = 0x6F; /* Set Port A direction *
  *PBDIR = 0xF0; /* Set Port B direction *
  *CTCON = 0x2; /* Stop Timer (if running) *
  *CNTM = 100000000; /* Initialize: 1-s timeout */
  *CTSTAT = 0x0; /* Clear "Reached 0" flag *
  *IVECT = (unsigned int *) &intserv; /* Set interrupt vector *
  asm("MoveControl PSR,#0x40"); /* CPU responds to IRQ */
  *PAOUT = 0x20; /* Initialize port A *
  *PBOUT = 0x00; /* Initialize port B *
  *CTCON = 0x11; /* Start Time
  while (1) {
    while ((*PBIN & 0x01) != 0); /* Wait for SW press */
    while ((*PBIN & 0x01) == 0); /* Wait for SW release */
    leds ^= 0x1; /* Toggle LED flag
    *PAOUT ^= 0x60; /* Flip LED1/LED2 state
  }
  exit(0);
}

interrupt void intserv() {
  *CTSTAT = 0x0; /* Clear "Reached 0" flag *
  if (leds == 0x1) {
    if (digit1 == 0) digit1 = 9;
    else digit1 = digit1 - 1; /* Decrement DIGIT1 */
    *PAOUT = (0x20 | digit1); /* Update port A, LED1 on, LED2 off */
  }
  else {
    if (digit2 == 0) digit2 = 9;
    else digit2 = digit2 - 1; /* Decrement DIGIT1 */
    *PBOUT = digit2 << 4; /* Update port B */
  }
}
```

1. *[15 points]* The textbook's microcontroller is used in a system shown below and is responsible for two tasks: (1) incrementing either **DIGIT1** (when **LED1** is on) or **DIGIT2** (when **LED2** is on) every second, and (2) alternating between **LED1** and **LED2** being on, whenever the **SW** key is hit (i.e., pressed and then released). Write the corresponding C program, assuming that the **second task** is the main program, and the **first task** is an ISR whose address is stored at location **0x20**. Also, assume that bit **6** of the processor status register (i.e., **PSR[6]**) is the processor's interrupt-enable bit, and **Ports A** and **B** are always ready to be written by the processor. Initially, **LED1** is on, **LED2** is off, and both **DIGIT1** and **DIGIT2** show **0**.

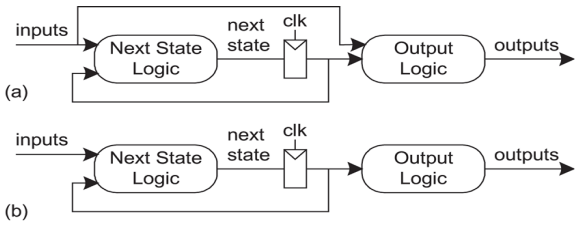
• *Main Program*: Whenever $\mathbf{PB_4}$ first becomes 0 and then 1 again, **LED1** and **LED2** must swap their states: if **LED1** is on and **LED2** is off, then **LED1** becomes off and **LED2** becomes on, and vice versa. (Note: **LED1** and **LED2** should never be both on, or both off.)

• *ISR*: The 100-MHz Counter/Timer must be configured to generate interrupts every second. Its ISR must increment **DIGIT1** if **LED1** is on, or increment **DIGIT2** if **LED2** is on. (Note: incrementing **9** gives **0**.)

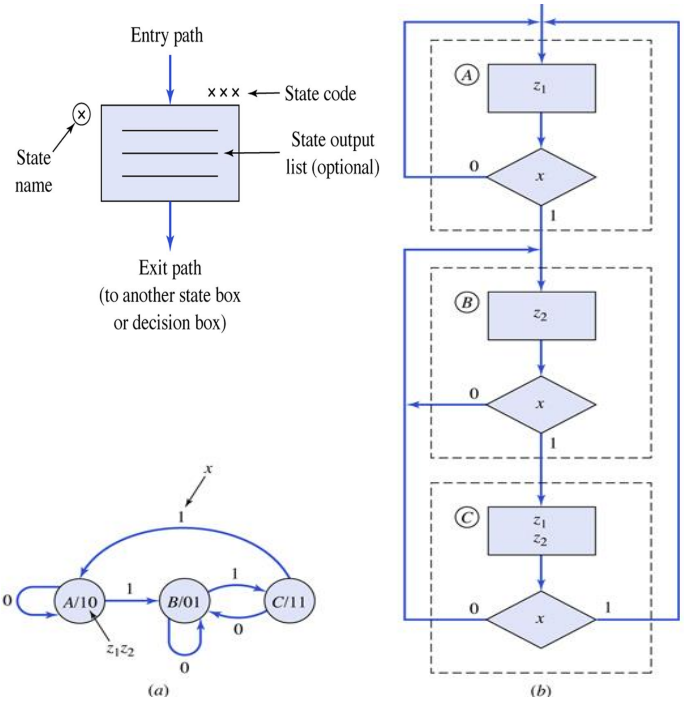



```
#define PAIN (volatile unsigned char *) 0xFFFFFFF0
#define PAOUT (volatile unsigned char *) 0xFFFFFFF1
#define PADIR (volatile unsigned char *) 0xFFFFFFF2
#define PBIN (volatile unsigned char *) 0xFFFFFFF3
#define PBOUT (volatile unsigned char *) 0xFFFFFFF4
#define PBDIR (volatile unsigned char *) 0xFFFFFFF5
#define CNTM (volatile unsigned int *) 0xFFFFFFD0
#define CTCON (volatile unsigned char *) 0xFFFFFFD8
#define CTSTAT (volatile unsigned char *) 0xFFFFFFD9
#define IVECT (volatile unsigned int *) (0x20)

interrupt void intserv();

volatile unsigned char digit1 = 0;        /* DIGIT1 for display */
volatile unsigned char digit2 = 0;        /* DIGIT2 for display */
volatile unsigned char leds = 0x1;        /* LED1 on, LED2 off */

int main() {
  *PADIR = 0xF4;                          /* Set Port A direction */
  *PBDIR = 0x8F;                          /* Set Port B direction */
  *CTCON = 0x2;                           /* Stop Timer (if running) */
  *CNTM = 100000000                       /* Initialize: 1-s timeout */
  *CTSTAT = 0x0;                          /* Clear "Reached 0" flag */
  *IVECT = (unsigned int *) &intserv;     /* Set interrupt vector */
  asm("MoveControl PSR,#0x40");           /* CPU responds to IRQ */
  *PAOUT = 0x0;                           /* Initialize port A */
  *PBOUT = 0x80;                          /* Initialize port B */
  *CTCON = 0x11;                          /* Start Timer
  while (1) {
    while ((*PBIN & 0x10) != 0);          /* Wait for SW press */
    while ((*PBIN & 0x10) == 0);          /* Wait for SW release */
    leds ^= 0x1;                          /* Toggle LED flag
    *PAOUT ^= 0x04;                       /* Flip LED1 state
    *PBOUT ^= 0x80;                       /* Flip LED2 state
  }
  exit(0);
}

interrupt void intserv() {
  *CTSTAT = 0x0;                          /* Clear "Reached 0" flag */
  if (leds == 0x1) {
      digit1 = (digit1+1)%10;             /* Increment DIGIT1 */
      *PAOUT = digit1 << 4;               /* Update port A, LED1 on */
    }
    else {
      digit2 = (digit2+1)%10;             /* Increment DIGIT2 */
      *PBOUT = digit2;                    /* Update port B, LED2 on */
    }
}
```

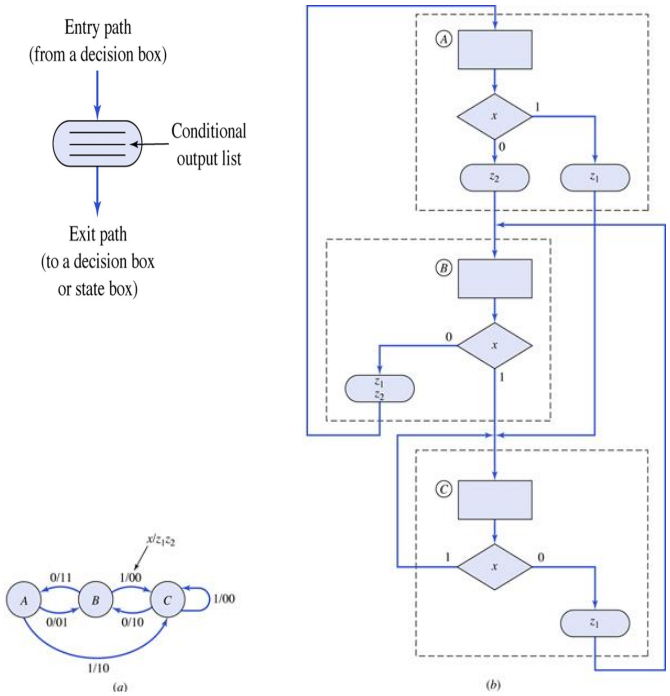
- Mealy (a) and Moore (b) FSMs
  - **Mealy**: output depends on input, one cycle faster
  - **Moore**: output does not depend on input, more reliable



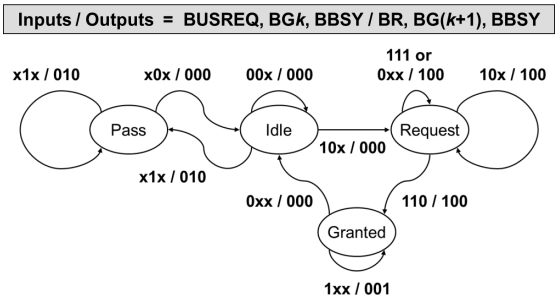
Moore Chart Example



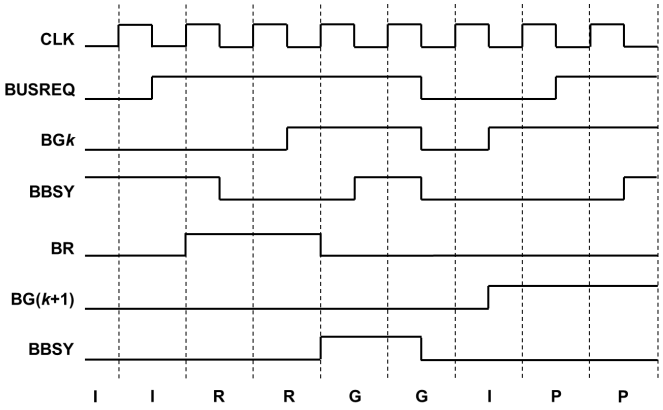
Mealy Chart Example



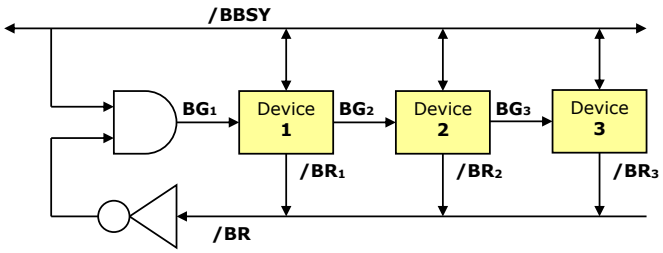
3. *[5 points]* Consider the Mealy FSM state diagram of some daisy-chain device as shown below, where **x** represents **don't-care**. Given the input waveforms shown below, draw the corresponding output waveforms, assuming that the FSM is initially in state **Idle**.



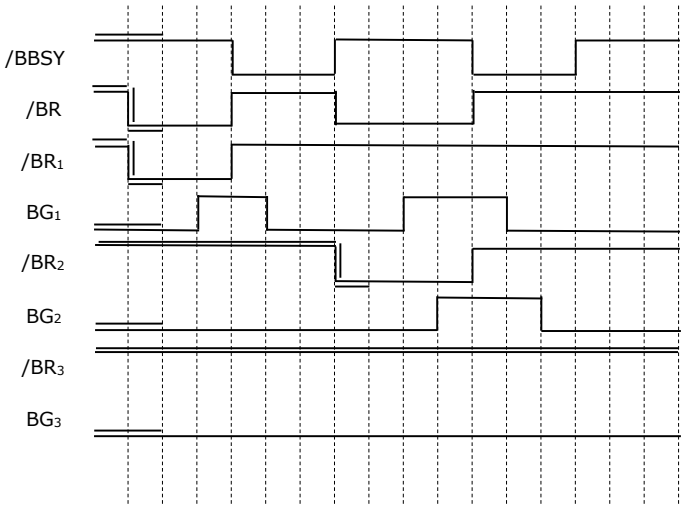
SOLUTION:



3. *[10 points]* Consider the daisy-chain arbitration scheme shown below. Assume that the input-to-output signal propagation delays are the same and equal to **d** for all three devices, the inverter, and the **AND** gate. Also, assume that device **x** is able to start using the bus (making **/BRx = 1** and **/BBSY = 0**) only when it receives a **0-1** transition on its bus-grant input **BGx** and detects that the bus is not currently busy (i.e., **/BBSY = 1**). Also, assume that device **x** lets the bus-grant propagate through only when it is neither requesting nor using the bus. Finally, assume that any of the three devices will need to use the granted bus for only **3d** time units. Complete the timing diagram shown on the next page.



3.

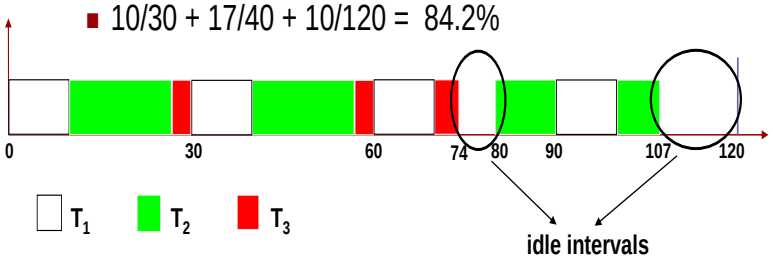


Task $T_i = (C_i, D_i, P_i, \phi_i)$, where:

C_i – worst case execution time (WCET), ϕ_i – initial delay, P_i – period, D_i – deadline (often same as P_i)

- Single-processor scheduling algorithm:
  - When task T_i arrives at time $\phi_i + kP_i$, assign a certain priority value τ_{ik} to it and place T_i into the prioritized queue of tasks ready for execution
    - Tasks in the ready queue are always ordered in the increasing order of their priorities
  - If τ_{ik} is greater than the priority of a task currently being executed, suspend that task and start T_i
  - Otherwise, once a current task is finished, say at time **t**, start the next highest-priority task in the ready queue
  - Examples of static priority assignment
    - Rate Monotonic (**RM**): $\tau_{ik} = 1/P_i$ (independent of **k**)
    - Deadline Monotonic (**DM**): $\tau_{ik} = 1/D_i$ (independent of **k**)
  - Examples of dynamic priority assignment
    - Earliest Deadline First (**EDF**): $\tau_{ik} = 1/(\phi_i + kP_i + D_i)$
    - Least Laxity First (**LLF**): $\tau_{ik} = 1/(\phi_i + kP_i + D_i - t - \Delta C_i)$, where ΔC_i is the remaining execution time of T_i
      - **Note:** $\Delta C_i = C_i$ if T_i has not been suspended previously
  - CPU utilization: $C_1/P_1 + C_2/P_2 + \ldots$

- Set of three pre-emptive tasks $T_i = (C_i, D_i, P_i, \phi_i)$
  - $\{ T_1=(10,30,30,0), T_2=(17,30,40,0), T_3=(10,120,120,0) \}$
- RM scheduling prioritization:
  - $\tau_{1k} = 1/30$ (highest), $\tau_{2k} = 1/40$, $\tau_{3k} = 1/120$ (lowest)
- CPU utilization:
  - 10/30 + 17/40 + 10/120 = 84.2%



2. *[5 points]* The table below specifies a set of independent pre-emptive tasks to be executed by a single processor. Show the task schedule using the *Rate Monotonic* (**RM**) priority assignment.

Task T_i	Period P_i	WCET C_i	Deadline D_i	Initial Delay ϕ_i
T1	20	5	20	0
T2	25	5	25	0
T3	50	10	50	0
T4	100	15	100	0

2.
The LCM (least common multiple) of all four periods is 100, i.e., we only need to determine our schedule in the time interval **[0, 100)**. RM task priorities are 1/20 for T1 arriving at (0, 20, 40, 60, 80); 1/25 for T2 arriving at (0, 25, 50, 75); 1/50 for T3 arriving at (0, 50); 1/100 for T4 arriving at (0).

RM Schedule
t=0: T1
t=5: T2
t=10: T3
t=20: T1
t=25: T2
t=30: T4
t=40: T1 (T4 preempted)
t=45: T4
t=50: T2
t=55: T3
t=60: T1 (T3 preempted)
t=65: T3
t=70: Idle
t=75: T2
t=80: T1
t=85: Idle
t=100: Repeat…

3. [5 points] The table below specifies a set of independent pre-emptive tasks to be executed by a single processor. Show the task schedule using the **Earliest Deadline First** (EDF) priority assignment. Note: If some tasks happen to have the same EDF priority, break such ties using **Rate Monotonic** (RM) prioritization.

Task T_i	Period P_i	WCET C_i	Deadline D_i	Initial Delay ϕ_i
T1	30	10	30	0
T2	40	10	40	0
T3	60	10	50	0
T4	120	15	100	0

3. The LCM (least common multiple) of all four periods is 120, i.e., we only need to determine our EDF schedule in the time interval **[0, 120)**, after which it is repeated.

EDF task priorities are: (1/30, 1/60, 1/90, 1/120) for T1 arriving at (0, 30, 60, 90); (1/40, 1/80, 1/120) for T2 arriving at (0, 40, 80); (1/50, 1/110) for T3 arriving at (0, 60); (1/100) for T4 arriving at (0).

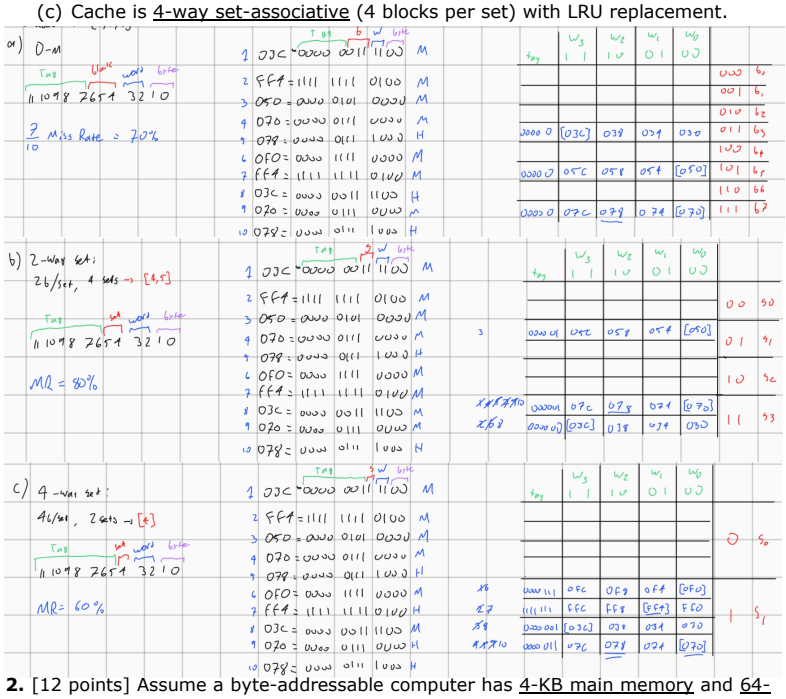
t=0: T1
t=10: T2
t=20: T3
t=30: T1
t=40: T2
t=50: T4
t=60: T1 (T4 preempted)
t=70: T4
t=75: T3
t=85: T2
t=90: T1 (T2 preempted)
t=100: T2
t=105: Idle
t=120: Repeat…

1. [12 points] Consider a byte-addressable computer with 4-KB main memory and 128-byte cache having **eight blocks**, where each block consists of **four 32-bit words**. Assume that the CPU reads 32-bit words from the following sequence of hexadecimal addresses:

03C FF4 050 070 078 0F0 FF4 03C 070 078

Show the cache contents (e.g., **[000]** = contents stored at address **000**) at the end of this sequence (10 addresses) and calculate the corresponding miss rate given that:

(a) Cache is direct-mapped.
(b) Cache is 2-way set-associative (2 blocks per set) with LRU replacement.
(c) Cache is 4-way set-associative (4 blocks per set) with LRU replacement.

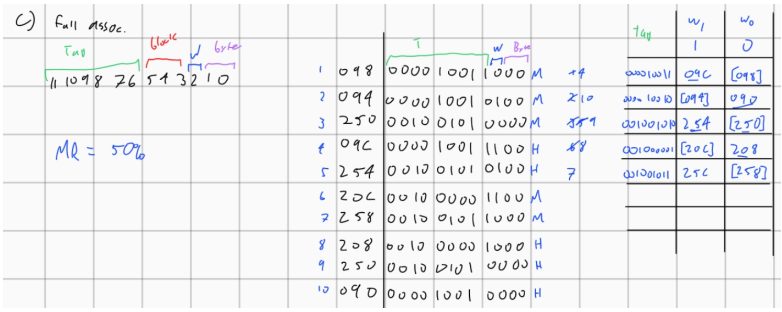


2. [12 points] Assume a byte-addressable computer has 4-KB main memory and 64-byte cache with **eight blocks**, where each block has **two 32-bit words**. While executing some program, the CPU reads 32-bit words from the following sequence of 10 addresses (in hexadecimal format):

098 094 250 09C 254 20C 258 208 250 090

Show the cache contents (e.g., **[000]** = contents stored at address **000**) at the end of this sequence and calculate the corresponding miss rate given that:

(c) Cache is fully-associative with LRU replacement.



3. [1 point] Assume a computer uses the same L1 and L2 caches for both data and instructions. The L1 access time is $C_1 = 2\tau$ (L1 hit), the L2 access time is $C_2 = 8\tau$ (L1 miss, L2 hit), and the main memory access time is $M = 32\tau$ (L2 miss). Assume that for some given application the L1 hit rate is $h_1 = 90\%$ (for both instructions and data). What is the L2 hit rate h_2 (for both instructions and data) such that $T_{ave} = 3\tau$?

Tave = $h_1C_1 + (1-h_1)(h_2C_2 + (1-h_2)M)$ -> solve for h_2

IEEE Single Precision

31	30 … 23	22 … 0
x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx
sign	exponent	mantissa (significand)

$$(-1)^S * 2^{E-127} * 1.M$$

	E = 0	0 < E < 255	E = 255
M=0	0	Powers of Two	∞
M!=0	Denormalized: $(-1)^S 2^{-126}$ 0.M (Underflow)	Ordinary FP Numbers	Not a Number

Converting to IEEE-754

- 1/3 = 0.33333…$_{10}$
  = 0 25 + 0.0625 + 0 015625 0.00390625
  + 0.0009765625 + …
  = 1/4 + 1/16 + 1/64 + 1/256 + 1/1024 + …
  = $2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10}$ + …
  = 0.0101010101…$_2$ * ^{0}
  = 1.0101010101…$_2$ * 2^{-2}
- Sign: 0 Exponent: -2 + 127 = 125$_{10}$ = 01111101$_2$ Significand: 0101010101…

0	01111101	01010101010101010101010

1. [5 points] Consider the code portion of the matrix-vector product computation as shown below: `(double)` **128x128** matrix **A** is multiplied by `(double)` **128x1** vector **X**, producing `(double)` **128x1** result **Y** (initially all 0's).

```
for (i = 0; i < 128; i++) {
      for (j = 0; j < 128; j++) {
            Y[i] = Y[i] + A[i][j]*X[j];
      }
}
```

Storing **X**, **Y**, and **A** (each `double` array element is 8 bytes in size) requires 128*8 + 128*8 + 128*128*8 = **130 KB** of memory. If the cache (assume fully associative) is smaller than 130 KB, the above code will cause many misses, considerably slowing down the program execution. Alternatively, one can perform blocked computation: partition **A** into smaller blocks and perform the product computation block-by-block. If our data blocks can fit into the cache, such blocked computation may significantly outperform the original code.

Rewrite the code fragment above using blocked computation and letting matrix **A**'s blocks be of size **64x64** (i.e., 4 blocks total). Assuming that such blocking yields the best performance, what can you say about the size of the cache?

```
for (p = 0; p < 2; p++) {        //p < (128/64), for p = 0 to 128 by 64
    for (q = 0; q < 2; q++) {    //q < (128/64), for q = 0 to 128 by 64
        for(i = p*64; i < (p+1)*64; i++){   //p=0 -> B1 i = 0-63, p=1 -> B2 i = 64-128
            for(j = q*64; q < 2; q++) {     //q=0 -> B3 j = 0-63, q=1 -> B4 j = 64-128
                Y[i] = Y[i] + A[i][j]*X[j];
            }
        }
    }
}

//Cache size = 128*8 + 128*8 + 64*64*8 = 34kB
```

2. [10 points] Consider a C code fragment below, processing some matrix `float X[N][N]` (stored row by row, i.e., in the row-major order), where **N = 512**:

```
float y, neg = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        if (X[i][j] < 0) neg++;      /* count negative elements */
    }
}
y = neg/(N*N);
for (i = 0; i < N; i++) {
    X[i][i] = X[i][i] + y;           /* add to diagonal */
}
```

Determine the **X**-related page fault rate in the following two cases: 1) the main memory uses **1-KB** paging with 4 pages allocated for **X**, and 2) the main memory uses **2-KB** paging with 2 pages allocated for **X**. Initially, no part of **X** is in the main memory. **Note:** `float` = 4 bytes.

```
A) 4 pages, 1kB ea
//#X access = 512*(512*1) + 512*2 = 263,168
//size of X[512][512] = 1MB = 1024kB -> 4 pages, can store 1/2 row per page: X[n][0:255]
//neg element loops
i = 0   j = 0       X[0][0] fault -> load P1 = X[0][0:255]
        j = 1-255   X[0][j] ok
        j = 256     X[0][256] fault -> load P2 = X[0][256:511]
        j = 257-511 X[0][j] ok

i = 1   j = 0       X[1][0] fault -> load P3 = X[1][0:255]
        j = 1-255   X[1][j] ok
        j = 256     X[1][256] fault -> load P4 = X[1][256:511]
        j = 257-511 X[1][j] ok

        ...

i = 510     j = 0           X[510][0] fault -> load P1 = X[510][0:255]
            j = 1-255       X[510][j] ok
            j = 256         X[510][256] fault -> load P2 = X[510][256:511]
            j = 257-511     X[510][j] ok

i = 511     j = 0           X[511][0] fault -> load P3 = X[511][0:255]
            j = 1-255       X[511][j] ok
            j = 256         X[511][256] fault -> load P4 = X[511][256:511]
            j = 257-511     X[511][j] ok

//2 faults per i++ -> #faults = 512*2 = 1024

//add diag loops
i = 0               X[0][0] fault -> load P1 = X[0][0:255]
i = 1               X[1][1] fault -> load P2 = X[1][0:255]
        ...
//1 fault per i++ -> #faults = 512

//total #faults = 1024+512 = 1536
Page fault rate = 1536/263,168 = 0.5837%
```

IEEE Double Precision

63	62 ... 52	51 ... 31 ... 0
x	xx...xxx	xxxxxxxxxx...xxxxxxxxxx
sign	exponent	mantissa (significand)

$(-1)^S * 2^{E-1023} * 1.M$

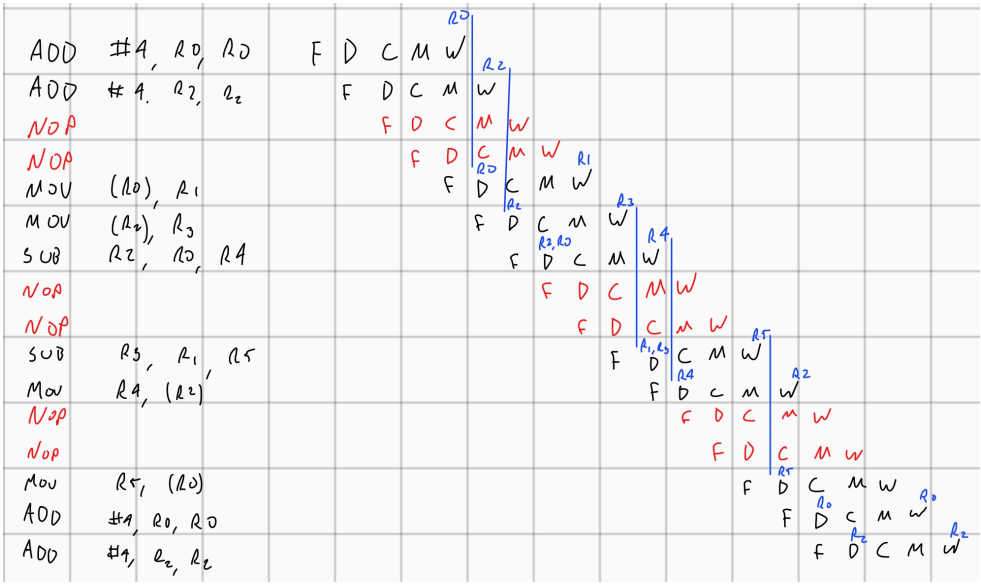
	E = 0	0<E<2047	E = 2047
M=0	0	Powers of Two	∞
M!=0	Denormalized: $(-1)^S\ 2^{-1022}\ 0.M$ (Underflow)	Ordinary FP Numbers	Not a Number

2. [5 points] Consider a pipelined datapath consisting of five stages:

F – fetch the instruction from the memory,
D – decode the instruction and read the source register(s),
C – execute the ALU operation specified by the instruction,
M – execute the memory operation specified by the instruction,
W – write the result in the destination register.

Identify data hazards in the code below and insert **NOP** instructions where necessary.

```
ADD     #4, R0, R0      // R0 = R0 + 4
ADD     #4, R2, R2      // R2 = R2 + 4
MOV     (R0), R1        // R1 = MEMORY[R0]
MOV     (R2), R3        // R3 = MEMORY[R2]
SUB     R2, R0, R4      // R4 = R2 - R0
SUB     R3, R1, R5      // R5 = R3 - R1
MOV     R4, (R2)        // MEMORY[R2] = R4
MOV     R5, (R0)        // MEMORY[R0] = R5
ADD     #4, R0, R0      // R0 = R0 + 4
ADD     #4, R2, R2      // R2 = R2 + 4
```



- 18.75 + 0.1875 = 18.9375
  - $18.75_{10} = 10010.11_2 = 1.001011 * 2^4$
    $= (-1)^0 * 2^{(131-127)} * 1.001011$
    $= 0\ 10000011\ 00101100000000000000000$
  - $0.1875_{10} = 0.0011_2 = 1.1 * 2^{-3}$
    $= (-1)^0 * 2^{(124-127)} * 1.1$
    $= \ \ 01111100\ 10000000000000000000000$
  - Don't forget implicit **1**:
    `0 1 000011 10010110000000000000000`
    `0 0 111100 10000000000000000000000`
  - Next: match the exponents before addition!
    - The difference: `10000011 - 01111100 = 00000111 = 7`
    - Need to right-shift the smaller number by **7** bits
  - Match the exponents:
    `0 10000011 10010110000000000000000`
    `0 10000011 00000001100000000000000`
  - Add:
    `0 10000011 10010111100000000000000`
  - Sum is already normalized (leading **1**)
  - Actual bits stored:
    `0 10000011 00101111000000000000000`
  - Actual meaning:
    `0 10000011 00101111000000000000000`
    `+ 2^(131-127) * 1.18359375 = 18.9375`

Two's Complement

- 2's complement: negation = bitwise inversion + 1
  - Example:
    $-94_{10} = -01011110 = 10100001+1 = 10100010$
- Example: adding two 2's complement numbers

$$\begin{array}{rl} 01011001_2 = & 9_{10} \\ \underline{11001101_2} = & \underline{-51_{10}} \\ 100100110_2 & 38_{10} \end{array}$$

  - Note:
    - Overflow with 2's complements: positive + positive = negative, or negative + negative = positive
    - Overflow occurs when carry-in to sign bit position is NOT equal to carry-out
    - When there is no overflow, carry-out can be ignored