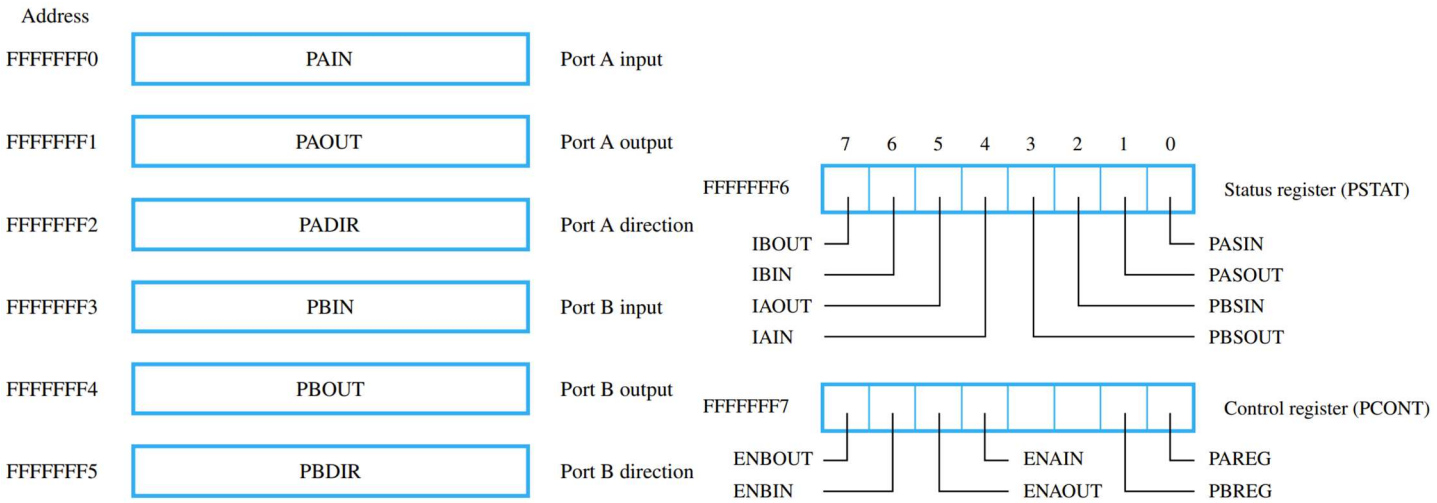
Textbook's Microprocessor:

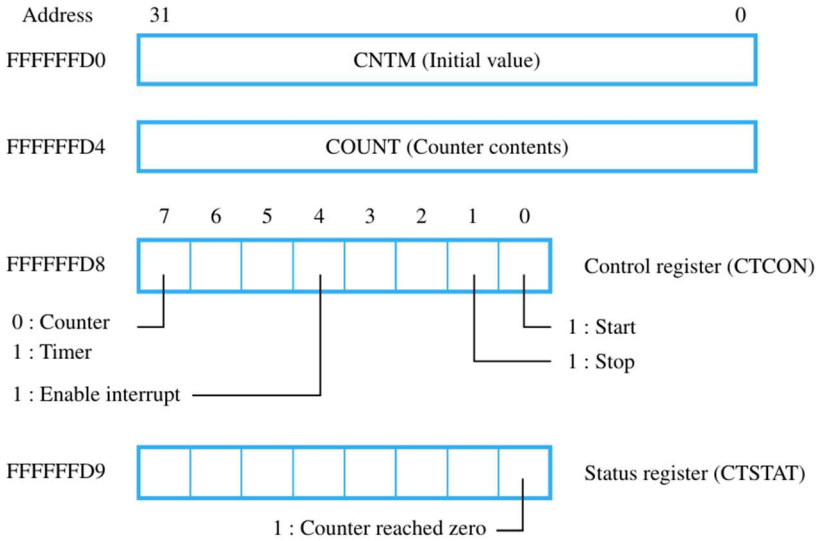


For PADIR, PBDIR: 0 -> input, 1 -> output

For PASIN/PBSIN: 1 -> new data are in PAIN or PBIN, 0 -> new data has been read

For PASOUT/PBSOUT: 1 -> data from PAOUT or PBOUT have been accepted by external device, 0 -> new data has been written

PCONT -> used to enable interrupts on change of P(A/B)IN or P(A/B)OUT, PCONT[0]-[3] should be set to 0.



Textbook counter always counts down.

Use CNTM to change length of timer. Textbook timer runs at 100MHz, so set CNTM = 100,000,000 (w/o commas) to set length to 1 second.

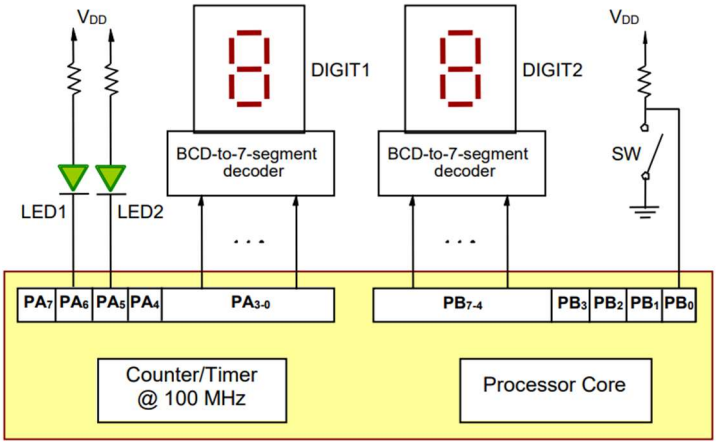
CTCON [7] = 0 (use counter mode, not timer mode)

Setting CTCON[0] = 0 does not stop the timer, must set CTCON[1] = 1

If polling, check for counter reached 0 flag = 1 to tell when time has elapsed

Must reset counter flag manually

Coding example: Timer interrupt, I/O main loop



Task 1 (main loop):

- Whenever button is pressed and released, LED 1 and 2 should swap states

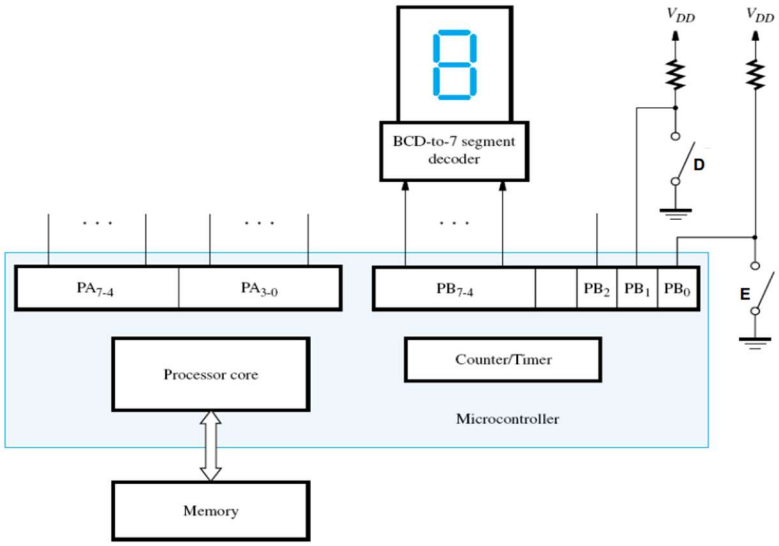
Task 2 (interrupt):

- Every second, decrement DIGIT1 if LED 1 is on, or decrement DIGIT2 if LED2 is on. (decrementing 9 gives 0)

```
interrupt void intserv();

volatile unsigned char display_1 = 0;
volatile unsigned char display_2 = 0;

volatile unsigned char led_1_on = 1;

int main() {
  *CTCON = 0x2; /* stop timer */
  *CTSTAT = 0x0; /* clear reached 0 flag */

  *PADIR = 0xFF; /* all out*/
  *PBDIR = 0xF0; /* ports [7]-[4] out, [3]-[0] in (only care about [0])*/

  *PAOUT = 0xB0; /* set display = 0, LED1 = on and LED2 = off (1011 0000)*/
  *PBOUT = 0x0; /* set display = 0*/

  *CNTM = 100000000; // 100MHz counter -> 100,000,000 cycles per second

  *IVECT = (unsigned int *) &intserv; /* set contents of address IVECT to memory address of intserv function */
  asm("MoveControl PSR, #0x40"); /* sets PSR[6] = 1, enabling interrupts */

  *CTCON = 0x11; /* start timer, enable interrupts */

  while(1) {
    if ((*PBIN | 0x1) == 0) { // if button is pressed
      while((*PBIN | 0x1) == 0); // loop while button held

      // when button released
      if (led_1_on) {
        *PAOUT = (unsigned char)(0xD0 | (*PAOUT & 0x0F)); // keep PAOUT[0]-[3], set bits [4]-[7] = 1101
        led_1_on = 0;
      } else {
        *PAOUT = (unsigned char)(0xB0 | (*PAOUT & 0x0F)); // keep PAOUT[0]-[3], set bits [4]-[7] = 1011
        led_1_on = 1;
      }
    }
```

```
  }
  exit(0);
}

interrupt void intserv() {
  *CTSTAT = 0x0; /* reset reached 0 flag */
  if (led_1_on) {
    display_1 = display_1 == 0 ? 9 : display_1 - 1; // if subtracting 1 would make display roll over, set to 9 instead
    *PAOUT = (unsigned char)((*PAOUT & 0xF0) | display_1); // keep PAOUT[4]-[7], set bits [0]-[3] = display[0]-[3]
  } else {
    display_2 = display_2 == 0 ? 9 : display_2 - 1; // if subtracting 1 would make display roll over, set to 9 instead
    *PBOUT = (display_2 << 4); // 0000 XXXX -> XXXX 0000 (shift bits left by 4)
  }
}
```

Coding example: Timer loop, I/O interrupt



Task 1 (main loop):

- Increment digit every second if E was pressed last. If D was pressed last, don't increment (incrementing 9 gives 0)

Task 2 (interrupt):

- Interrupt sent when PBIN updated. If the update was D being pressed, disable incrementing digit. If the update was E being pressed, enable incrementing digit

```
interrupt void intserv();

unsigned char display = 0;

int main() {
  *CTCON = 0x2; /* stop timer */
  *PBDIR = 0xF3; /* ports [7]-[4] out, [0] and [1] in*/
  *PBOUT = 0x0; /* set display = 0*/
  *PCONT = 0x40; /* enable interrupts for port B in (ENBIN = 1)*/
  *CNTM = 100000000; // 100MHz counter -> 100,000,000 cycles per second

  *IVECT = (unsigned int *) &intserv; /* set contents of address IVECT to memory address of intserv function */
  asm("MoveControl PSR, #0x40"); /* sets PSR[6] = 1, enabling interrupts */

  while(1) {
    while((*CTSTAT & 0x1) == 0x0); /* do nothing until 0 is reached*/

    *CTSTAT = 0x0; /* reset reached 0 flag */
    display = (display + 1) % 10;
    *PBOUT = (display << 4); // 0000 XXXX -> XXXX 0000 (shift bits left by 4)
```

```
  }

  exit(0);
}

interrupt void intserv() {
  if ((*PBIN & 0x02) == 0) *CTCON = 0x2; /* if disable button pressed (PBIN[1] == 0) stop timer */
  else if ((*PBIN & 0x01) == 0) *CTCON = 0x1; /* if enable button pressed (PBIN[0] == 0), start timer */
}
```

Scheduling:

Priority algorithms:

Rate monotonic	Deadline monotonic	Earliest deadline first	Least laxity first
$\tau_{ik} = \frac{1}{P_i}$	$\tau_{ik} = \frac{1}{D_i}$	$\tau_{ik} = \frac{1}{\phi_i + kP_i + D_i}$	$\tau_{ik} = \frac{1}{\phi_i + kP_i + D_i - t - \Delta C_i}$

Where ΔC_i is the remaining execution time of task i

CPU utilisation = $C_1/P_1 + C_2/P_2 + \ldots$

Steps for solving problems:

Step 1: Determine max time period to consider (LCM of task periods)

Step 2: Mark when tasks are added to the queue on the diagram

Step 3: Calculate priorities using priority algorithm

Step 4: Consider task priorities every time a new task is added (write out the current queue to determine next task to be scheduled if needed)

Step 5: Check to make sure all tasks finish before their deadlines

E.g. 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.

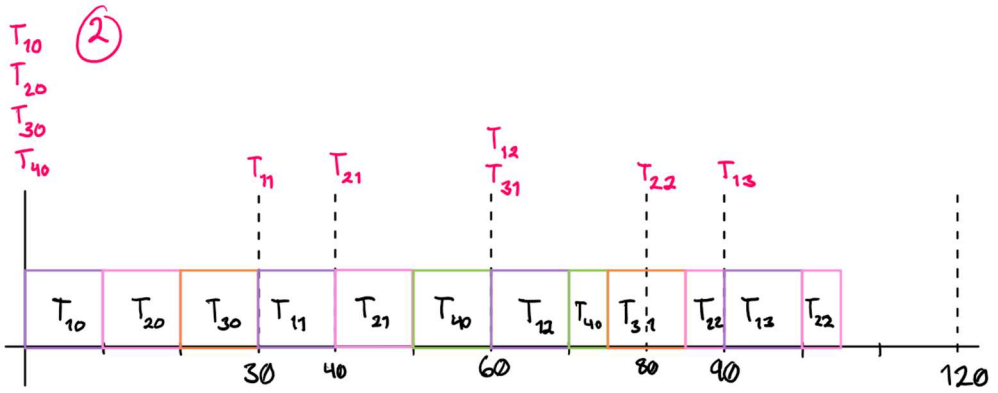
Task $\mathbf{T_i}$	Period $\mathbf{P_i}$	WCET $\mathbf{C_i}$	Deadline $\mathbf{D_i}$	Initial delay ϕ_i
T1	30	10	30	0
T2	40	10	40	0
T3	60	10	50	0
T4	120	15	100	0

Step 1: LCM = 120

Step 2: See pink text in diagram

Step 3: See orange text in diagram

Step 4: See coloured squares in diagram



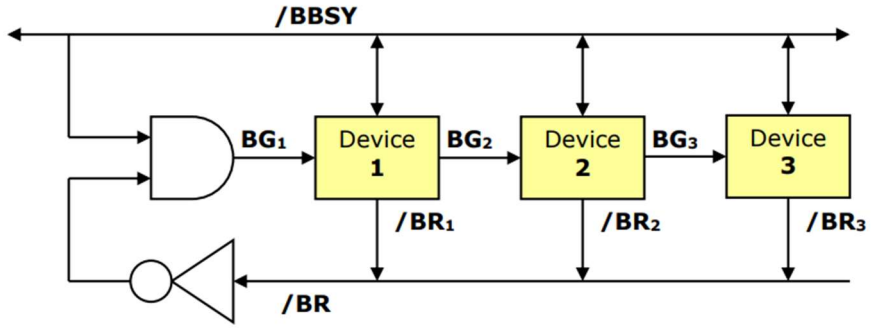
③ $\tau_{10} = \frac{1}{0 \cdot 30 + 30} = \frac{1}{30}$ $\tau_{11} = \frac{1}{1 \cdot 30 + 30} = \frac{1}{60}$ $\tau_{12} = \frac{1}{2 \cdot 30 + 30} = \frac{1}{90}$ $\tau_{13} = \frac{1}{3 \cdot 30 + 30} = \frac{1}{120}$

$\tau_{20} = \frac{1}{0 \cdot 40 + 40} = \frac{1}{40}$ $\tau_{21} = \frac{1}{1 \cdot 40 + 40} = \frac{1}{80}$ $\tau_{22} = \frac{1}{2 \cdot 40 + 40} = \frac{1}{120}$

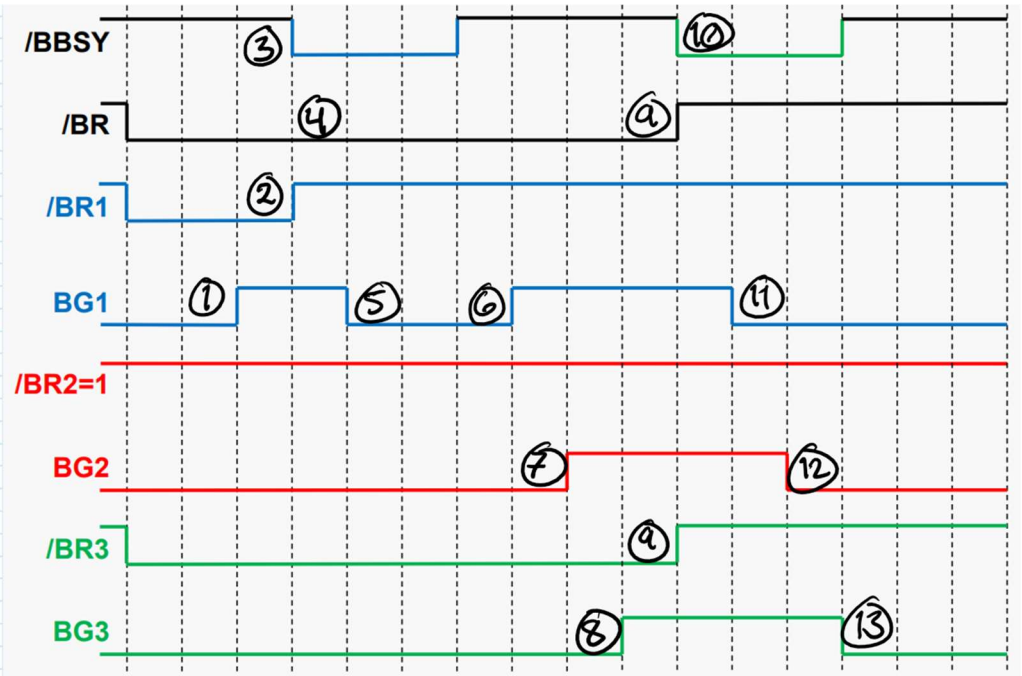
$\tau_{30} = \frac{1}{0 \cdot 60 + 50} = \frac{1}{50}$ $\tau_{31} = \frac{1}{1 \cdot 60 + 50} = \frac{1}{110}$

$\tau_{40} = \frac{1}{0 \cdot 120 + 100} = \frac{1}{100}$

Daisy chain arbitration timing diagrams:



Note: "/" means active/asserted low



1. AND gate sees /BR and /BBSY during first time interval and takes one time delay to assert BG1

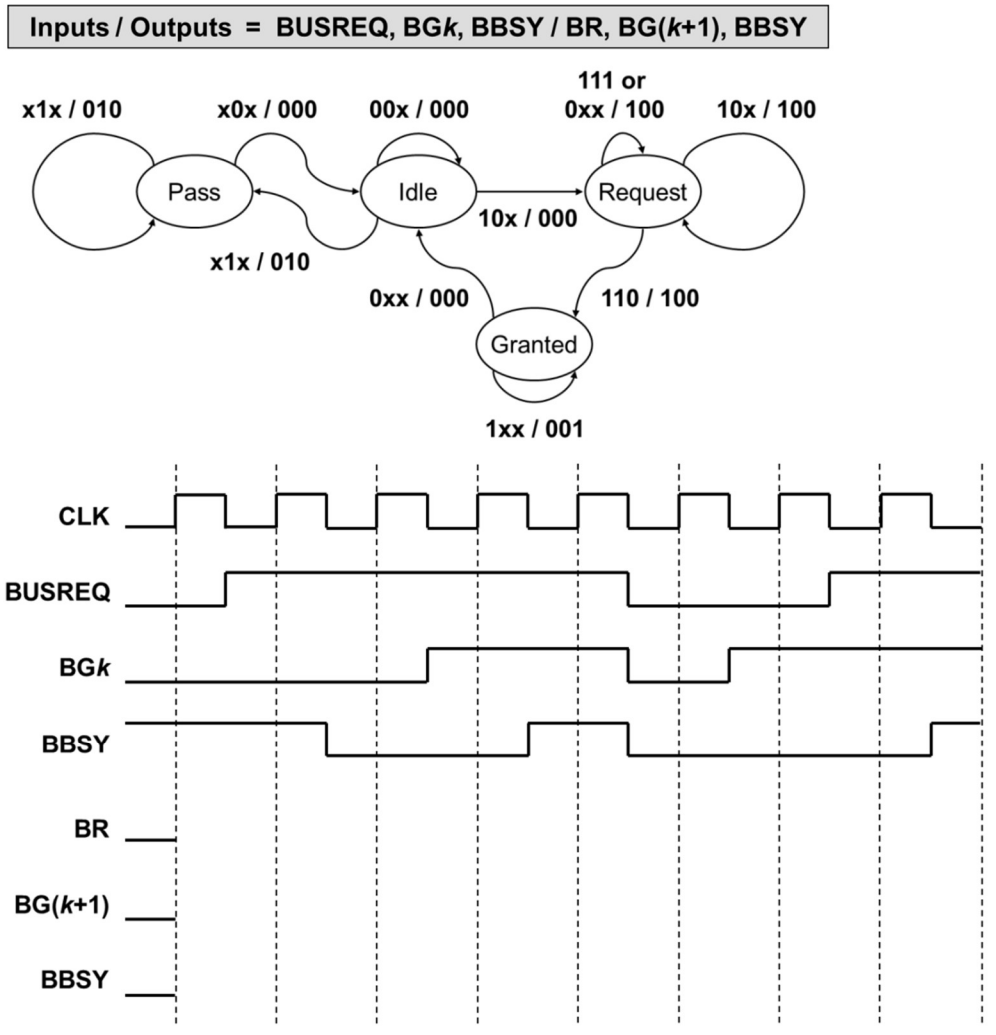
2. /BR1 takes one time delay after BG1 to stop asserting
3. /BBSY also takes one time delay after BG1 to assert
4. /BR remains asserted because /BR3 is still active
5. BG1 goes low one time delay after /BBSY stops asserting (one delay through AND gate)
6. Since there's still a request, BG1 goes high one time delay after /BBSY has stopped asserting
7. Device 1 doesn't want the signal, so BG2 goes high after one delay
8. Device 2 doesn't want the signal, so BG3 goes high after one delay
9. /BR3 takes one time delay after BG3 to stop asserting. Since /BR3 is directly connected to /BR and no other devices are requesting, /BR stops being asserted immediately.
10. /BBSY also takes one time delay after BG3 to assert
11. /BBSY is active, so after one delay BG1 stops being asserted
12. BG1 isn't active, so after one delay BG2 stops being asserted
13. BG2 isn't active, so after one delay BG3 stops being asserted

Mealy FSM waveform:

Step 1: Sample input at each rising clock edge and use values to determine state progression

Step 2: Use state information to determine outputs. The outputs will react if there are changes mid clock cycle, so your diagram must reflect this. Make sure to use the output values from the current state!! (The state never changes mid clock cycle)

E.g.



State progression:

Given that first state is idle

1st clock edge: sample was 001, prev state was idle, so period 1 is idle

2nd clock edge: sample was 101, prev state was idle, so period 2 is request

3rd clock edge: sample was 100, prev state was request, so period 3 is request

4th clock edge: sample was 110, prev state was request, so period 4 is granted

5th clock edge: sample was 111, prev state was granted, so period 5 is granted

6th clock edge: sample was 000, prev state was granted, so period 6 is idle

7th clock edge: sample was 010, prev state was idle, so period 7 is pass

8th clock edge: sample was 110, prev state was pass, so period 8 is pass

Outputs:

Period 1 (I): first half of clock cycle input 001 gives output of 000, second half of clock cycle input 101 gives output of 000

Period 2 (R): first half of clock cycle input 101 gives output of 100, second half of clock cycle input 100 gives output of 100

Period 3 (R): first half of clock cycle input 100 gives output of 100, second half of clock cycle input 110 gives output of 100

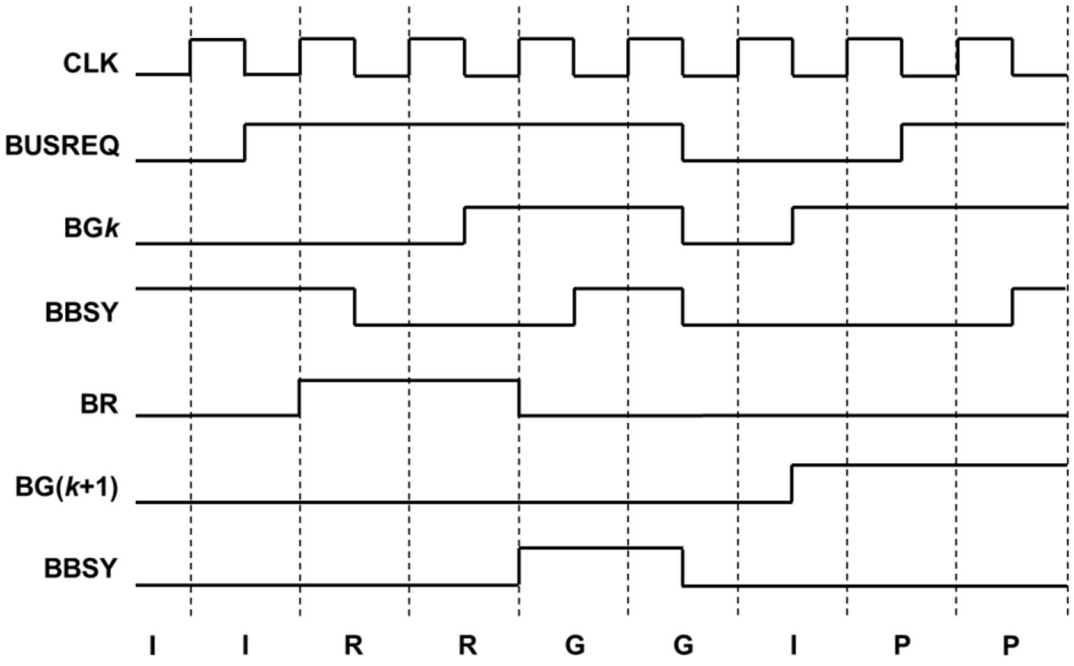
Period 4 (G): first half of clock cycle input 110 gives output of 001, second half of clock cycle input 111 gives output of 001

Period 5 (G): first half of clock cycle input 111 gives output of 001, second half of clock cycle input 000 gives output of 000

Period 6 (I): first half of clock cycle input 000 gives output of 000, second half of clock cycle input 010 gives output of 010

Period 7 (P): first half of clock cycle input 010 gives output of 010, second half of clock cycle input 110 gives output of 010

Period 8 (P): first half of clock cycle input 110 gives output of 010, second half of clock cycle input 111 gives output of 010



Cache Mapping:

- Word size determines number of bits needed for the byte field (e.g. for a 32-bit machine, words are 4 bytes long and need 2 bits for the byte field)
- Block size determines number of bits needed for the word field

For direct mapping:

- The number of blocks determines number of bits needed for the block field

For set-associative:

- "n-way" associative means n blocks per set
- Number of bits needed for set field is determined by the number of blocks / n

For fully-associative:

- Just uses byte, word, and tag fields

- Remaining bits used for tag field
- Memory size determines length of address (e.g. 4KB → $2^2 \times 2^{10} = 2^{12}$ so 12 bits needed)

E.g. A block-set-associative cache consists of a total of 64 blocks, divided into 4-block sets. The main memory contains 4096 blocks, each consisting of 32 words. Assuming a 32-bit byte-addressable address space, how many bits are there in each of the Tag, Set, and Word fields?

64/4 = 16 sets so 4 bits needed for set field, one block has 32 words so 5 bits needed for word field, assume 32 bit machine (i.e. word size is 4 bytes) so 2 bits needed for byte field. Remaining 21 bits (32 - 4 - 5 - 2 = 21) used for tag field.

Steps for doing the questions without writing out the cache table each time:

1. Write out the converted binary numbers in the same sequence as the hex addresses
2. Group the columns according to the cache method by drawing lines across the rows of the converted addresses
3. Starting from the top of the list…
    a. For direct mapping: draw symbols next to the entries that share the same block bits.
    b. For set-associative: draw symbols next to the entries that share the same set bits. Also write position within each set next to the entry (1st position, 2nd position, etc.) next to the symbol, representing replacing the least recently used entry by reusing the number.
    c. For fully-associative: draw symbols next to the entries which share the same tags.
4. Starting from the top, count the misses and hits. Hits occur when row/set and tag bits match a previous entry. Calculate the miss rate by dividing the misses by the total number of accesses and multiplying by 100.
5.
    a. For direct mapping: starting from the bottom of the list, underline any address that has a new shape next to it. Using the underlined entries, fill in the final memory table.
    b. For set-associative: starting from the bottom, underline the entries with new shape/number combinations. Using the underlined entries, fill in the final memory table.
    c. For fully-associative: starting from the top, fill the final memory table with each new shape encountered.

E.g. 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

From the question statement, the byte field is address bits [0:1] (word size is 32 bit or 4 bytes) and the word field is address bits [2:3] (each block is 4 words wide).

Step 1 (convert to binary):

Each hex digit represents 4 binary digits.

0000 0011 1100	03C
1111 1111 0100	FF4
0000 0101 0000	050
0000 0111 0000	070
0000 0111 1000	078
0000 1111 0000	0F0
1111 1111 0100	FF4
0000 0011 1100	03C
0000 0111 0000	070
0000 0111 1000	078

0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Direct mapped:

Step 2 (group columns for direct mapping):

8 rows/blocks, so need 3 bits for block field.

Tag	Block	Word	Byte	
0000 0	011	11	00	03C
1111 1	111	01	00	FF4
0000 0	101	00	00	050
0000 0	111	00	00	070
0000 0	111	10	00	078
0000 1	111	00	00	0F0
1111 1	111	01	00	FF4
0000 0	011	11	00	03C
0000 0	111	00	00	070
0000 0	111	10	00	078

Step 3 (mark entries with same row):

Tag	Block	Word	Byte		
0000 0	011	11	00	03C	*
1111 1	111	01	00	FF4	X
0000 0	101	00	00	050	•
0000 0	111	00	00	070	X
0000 0	111	10	00	078	X
0000 1	111	00	00	0F0	X
1111 1	111	01	00	FF4	X
0000 0	011	11	00	03C	*
0000 0	111	00	00	070	X
0000 0	111	10	00	078	X

Step 4 (count misses and total accesses):

Tag	Block	Word	Byte			
0000 0	011	11	00	03C	*	miss
1111 1	111	01	00	FF4	X	miss
0000 0	101	00	00	050	•	miss
0000 0	111	00	00	070	X	miss
0000 0	111	10	00	078	X	hit
0000 1	111	00	00	0F0	X	miss
1111 1	111	01	00	FF4	X	miss
0000 0	011	11	00	03C	*	hit
0000 0	111	00	00	070	X	miss
0000 0	111	10	00	078	X	hit

Step 5 (determine final entries):

Tag	Block	Word	Byte			
0000 0	011	11	00	03C	*	miss
1111 1	111	01	00	FF4	X	miss
0000 0	101	00	00	**050**	•	miss
0000 0	111	00	00	070	X	miss
0000 0	111	10	00	078	X	hit
0000 1	111	00	00	0F0	X	miss
1111 1	111	01	00	FF4	X	miss
0000 0	011	11	00	**03C**	*	hit
0000 0	111	00	00	070	X	miss
0000 0	111	10	00	**078**	X	hit

7/10 misses, so 70% miss rate

Final cache:

Tag	Word 11	Word 10	Word 01	Word 00	Row
					000
					001
					010
00000	**[03C]**	[038]	[034]	[030]	011
					100
00000	[05C]	[058]	[054]	**[050]**	101
					110
00000	[07C]	**[078]**	[074]	[070]	111

2-way associative:

Step 2 (group columns for sets):

(8 blocks)/(2 blocks per set) = 4, so need 2 bits for set field.

Tag	Set	Word	Byte	
0000 00	11	11	00	03C
1111 11	11	01	00	FF4
0000 01	01	00	00	050
0000 01	11	00	00	070
0000 01	11	10	00	078
0000 11	11	00	00	0F0
1111 11	11	01	00	FF4
0000 00	11	11	00	03C
0000 01	11	00	00	070
0000 01	11	10	00	078

Step 3 (mark entries with same row):

Tag	Set	Word	Byte			Replaces
0000 00	11	11	00	03C	X, 1	
1111 11	11	01	00	FF4	X, 2	
0000 01	01	00	00	050	•, 1	
0000 01	11	00	00	070	X,1	03C
0000 01	11	10	00	078	X,1	
0000 11	11	00	00	0F0	X, 2	FF4
1111 11	11	01	00	FF4	X, 1	070
0000 00	11	11	00	03C	X, 2	0F0
0000 01	11	00	00	070	X,1	FF4
0000 01	11	10	00	078	X,1	

Note: “replaces” column helps explain what’s happening and isn’t required.

Step 4 (count misses and total accesses):

Tag	Set	Word	Byte			
0000 00	11	11	00	03C	X, 1	miss
1111 11	11	01	00	FF4	X, 2	miss
0000 01	01	00	00	050	•, 1	miss
0000 01	11	00	00	070	X,1	miss
0000 01	11	10	00	078	X,1	hit
0000 11	11	00	00	0F0	X, 2	miss
1111 11	11	01	00	FF4	X, 1	miss
0000 00	11	11	00	03C	X, 2	miss
0000 01	11	00	00	070	X,1	miss
0000 01	11	10	00	078	X,1	hit

8/10 misses, so 80% miss rate

Step 5 (determine final entries):

Tag	Set	Word	Byte			
0000 00	11	11	00	03C	X, 1	miss
1111 11	11	01	00	FF4	X, 2	miss
0000 01	01	00	00	**050**	•, 1	miss
0000 01	11	00	00	070	X,1	miss
0000 01	11	10	00	078	X,1	hit
0000 11	11	00	00	0F0	X, 2	miss
1111 11	11	01	00	FF4	X, 1	miss
0000 00	11	11	00	**03C**	X, 2	miss
0000 01	11	00	00	070	X,1	miss
0000 01	11	10	00	**078**	X,1	hit

Final Cache:

Tag	Word 11	Word 10	Word 01	Word 00	Set
					00
					00
000001	[05C]	[058]	[054]	**[050]**	01
					01
					10

					10
000001	[07C]	**[078]**	[074]	[070]	11
000000	**[03C]**	[038]	[034]	[030]	11

Fully associative:

Step 2 (group columns):

Tag	Word	Byte	
0000 0011	11	00	03C
1111 1111	01	00	FF4
0000 0101	00	00	050
0000 0111	00	00	070
0000 0111	10	00	078
0000 1111	00	00	0F0
1111 1111	01	00	FF4
0000 0011	11	00	03C
0000 0111	00	00	070
0000 0111	10	00	078

Step 3 (mark entries with same tag):

Tag	Word	Byte		
0000 0011	11	00	03C	X
1111 1111	01	00	FF4	•
0000 0101	00	00	050	▪
0000 0111	00	00	070	*
0000 0111	10	00	078	*
0000 1111	00	00	0F0	°
1111 1111	01	00	FF4	•
0000 0011	11	00	03C	X
0000 0111	00	00	070	*
0000 0111	10	00	078	*

Step 4 (count misses and total accesses):

Tag	Word	Byte			
0000 0011	11	00	03C	X	miss
1111 1111	01	00	FF4	•	miss
0000 0101	00	00	050	▪	miss
0000 0111	00	00	070	*	miss
0000 0111	10	00	078	*	hit
0000 1111	00	00	0F0	°	miss
1111 1111	01	00	FF4	•	hit
0000 0011	11	00	03C	X	hit
0000 0111	00	00	070	*	hit
0000 0111	10	00	078	*	hit

5/10 misses, so 50% miss rate

Step 5 (determine final entries):

Tag	Word	Byte			
0000 0011	11	00	**03C**	X	miss
1111 1111	01	00	**FF4**	•	miss
0000 0101	00	00	**050**	▪	miss
0000 0111	00	00	**070**	*	miss
0000 0111	10	00	078	*	hit
0000 1111	00	00	**0F0**	°	miss
1111 1111	01	00	FF4	•	hit
0000 0011	11	00	03C	X	hit
0000 0111	00	00	070	*	hit
0000 0111	10	00	078	*	hit

Final cache:

Tag	Word 11	Word 10	Word 01	Word 00
0000 0011	**[03C]**	[038]	[034]	[030]
1111 1111	[FFC]	[FF8]	**[FF4]**	[FF0]
0000 0101	[05C]	[058]	[054]	**[050]**
0000 0111	[07C]	[078]	[074]	**[070]**
0000 1111	[0FC]	[0F8]	[0F4]	**[0F0]**

Page fault rates for virtual memory:

Step 1: calculate the row size by multiplying the number of entries by the size of each element

Step 2: count the total number of memory accesses (both reads AND writes)

Step 3: write out where faults occur for various values of the row and column variables until a pattern is visible

Step 4: count the number of faults by adding the faults from each looping section and calculate the page fault rate by dividing the total faults by the total memory accesses and multiplying by 100.

E.g. 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 three cases: a) the main memory uses 1-KB paging with 4 pages allocated for X, b) the main memory uses 2-KB paging with 2 pages allocated for X, and c) the main memory uses 4-KB paging with 1 page allocated for X. Initially, no part of X is in the main memory. Note: float = 4 bytes.

Step 1:

Row size is 512 elements × 4 bytes per elements, so 2KB.

Step 2:

For the loop that counts negative elements, the inner loop accesses the memory 1 time per value of j. The inner loop runs 512 times for each value of i. The total number of memory accesses for that section is 1 × 512 (max j) × 512 (max i) = 262,144.

For the loop that adds to the diagonal, it accesses memory 2 times per value of i. So the total number of memory accesses for that section is 2 × 512 (max i) = 1024.

The total number of memory accesses is 262,144 + 1,024 = 263,168.

Step 3 (for case a):

1KB page means X elements brought into memory in chunks like X[0][0:255], X[0][256:511], X[1][0:255], X[1][256:511], etc.

For each value of i in loop that counts negative elements…

j = 0	X[i][0]	→ fault
j = [1:255]	X[i][1:255]	→ no fault
j = 256	X[i][256]	→ fault
j = [257:511]	X[i][257:511]	→ no fault

So 2 faults per value of i. Total faults for this section is 2 × 512 (# of i values) = 1,024.

For diagonal loop adds to diagonal…

i = 0	X[0][0]	→ fault
i = 1	X[1][1]	→ fault
…		

So 1 fault per value of i. Total faults for this section is 1 × 512 (# of i values) = 512.

Overall faults = 1,024 + 512 = 1,536.

Step 4 (for case a):

Fault rate = (1,536 / 263,168) × 100 = 0.5836%

Step 3 (for case b):

2KB page means X elements brought into memory in chunks like X[0][0:511], X[1][0:511], X[2][0:511], etc.

For each value of i in loop that counts negative elements…

j = 0	X[i][0]	→ fault
j = [1: 511]	X[i][1:511]	→ no fault

So 2 faults per value of i. Total faults for this section is 1 × 512 (# of i values) = 512.

For diagonal loop adds to diagonal…

i = 0	X[0][0]	→ fault
i = 1	X[1][1]	→ fault
…		

So 1 fault per value of i. Total faults for this section is 1 × 512 (# of i values) = 512.
Overall faults = 512 + 512 = 1,024.

Step 4 (for case b):

Fault rate = (1,024 / 263,168) × 100 = 0.3891%

Step 3 (for case c):

4KB page means X elements brought into memory in chunks like X[0][0:511] and X[1][0:511], X[2][0:511] and X[3][0:511], etc.

For loop that counts negative elements…

i, j = **0**, 0	X[0][0]	→ fault
i, j = **0**, [1: 511]	X[0][1:511]	→ no fault
i, j = **1**, [0: 511]	X[1][0:511]	→ no fault
…		

So 1 fault per every other value of i. Total faults for this section is 1 × (512 × 0.5) = 256.

For diagonal loop adds to diagonal…

i = 0	X[0][0]	→ fault
i = 1	X[1][1]	→ no fault
…		

So 1 fault per value of i. Total faults for this section is 1 × (512 × 0.5) = 256.
Overall faults = 256 + 256 = 512.

Step 4 (for case c):

Fault rate = (512 / 263,168) × 100 = 0.1946%

<u>IEEE-754 Decimal Representation:</u>

31	30 23	22 0
x	xxxxxxxx	xxxxxxxxxxxxxxxxxxxxxxx
s i g n	exponent	mantissa (significand)

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

	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

E.g. convert 0 00000000 11000000000000000000000 from IEEE-754 to decimal

Special case (denormalised), $(-1)^0 \times 2^{-126} \times 0.11_2 = (0.5 + 0.25) \times 2^{-126} \approx 8.81620763 \times 10^{-39}$

E.g. convert 1 11111111 00000000000000000000000 from IEEE-754 to decimal

Special case (inf), $(-1)^1 \times \infty = -\infty$

E.g. convert -0.625 from decimal to IEEE-754

$-0.625 = -(0.5 + 0.125) = -0.101_2 = (-1)^1 \times 2^{-1} \times 1.01_2 = (-1)^1 \times 2^{126-127} \times 1.01_2$

1 01111110 01000000000000000000000

Adding and subtracting floating point numbers:

E.g. Subtract the following numbers and convert the result to decimal

X = 0 01111011 10000000000000000000000

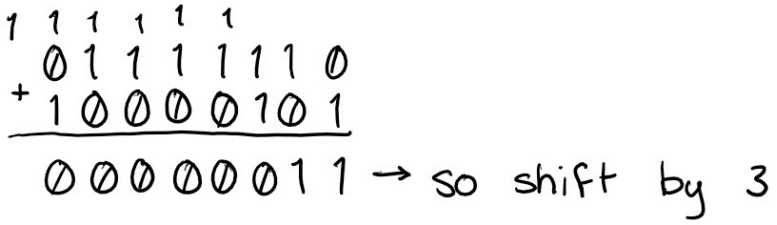
Y = 1 01111110 11010000000000000000000

Step 1: convert to decimal representation and add the implicit 1

X = 0 01111011 **1**10000000000000000000000

Y = 1 01111110 **1**11010000000000000000000

Step 2: match the exponents by subtracting the smaller exponent from the larger one (example subtracts using 2's complement)



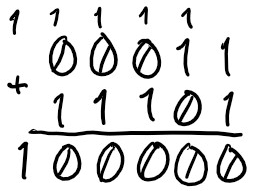
X = 0 01111110 000**1**10000000000000000000

Y = 1 01111110 **1**11010000000000000000000

Step 3: flip the sign of the subtrahend (only do for subtraction)

-Y = 0 01111110 **1**11010000000000000000000

Step 4: add the significands



Result: 0 01111110 **10**00000000000000000000000

Step 5: normalise the result and remove the implicit 1

Result: 0 01111111 00000000000000000000000

Step 6: convert to decimal

$(-1)^0 \times 2^{127-127} \times 1.0 = 1.0$

<u>Pipelining:</u>

5 stages of pipelining in our examples:

1. F – fetch instruction from memory
2. D – decode the instruction and read the source registers
3. C – execute ALU operation specified by the instruction
4. M – execute the memory operation specified by the instruction
5. W – write the result in the destination register

Stage 5 must be complete before stage 2 of the next instruction if using the same register

E.g. insert NOPs where necessary for the following instructions

1. ADD R2, R4, R1 // R1 = R2 + R4
2. ADD R4, R6, R5 // R5 = R4 + R6
3. ADD R0, R2, R3 // R3 = R0 + R2
4. MOV R6, (R1) // MEMORY[R1] = R6
5. MOV (R3), R6 // R6 = MEMORY[R3]
6. MOV R4, R2 // R2 = R4
7. ADD #4, R4, R4 // R4 = R4 + 4
8. ADD R0, R2, R1 // R1 = R0 + R2
9. MOV R2, R0 // R0 = R2

1	F	D	C	M	W	*R1 OK*							
2	R4, R6	F	D	C	M	W	R5 OK						
3		R0, R2	F	D	C	M	W	*R3 OK*					
4			*R1, R6*	F	D	C	M	W					
5				*R3*	F	D	C	M	W	R6 OK			
6					R4	*F*	D	C	M	W	*R2 OK*		
7						R4	F	D	C	M	W	R4 OK	
8							*R0, R2*	F	D	C	M	W	R1 OK
9								*R2*	F	D	C	M	W

1	F	D	C	M	W												
2		F	D	C	M	W											
3			F	D	C	M	W										
NOP				F	D	C	M	W									
4					F	D	C	M	W								
NOP						F	D	C	M	W							
5							F	D	C	M	W						
6								F	D	C	M	W					
7									F	D	C	M	W				
NOP										F	D	C	M	W			
NOP											F	D	C	M	W		
8												F	D	C	M	W	
9													F	D	C	M	W