[9] Progettare una macchina a stati finiti in grado di leggere un testo e di riconoscere all'interno del testo la parola 'con'.
Supponiamo che:
la macchina legga in ogni istante un carattere.
per semplicità supponiamo che i caratteri che possono essere letti sono: 'Ɵ','c','o','n'.
Definire lo STG, la STT, codificarla e realizzare il circuito mediante porte logiche. Qual è il cammino critico della macchina?

Soluzione:
L'automa a stati finiti deve ricordare ad ogni istante quale prefisso della parola da cercare ha già visto in precedenza, utile per formare eventualmente la frase completa e generare una volta riconosciuta la frase corretta l'uscita opportuna. Se ne deduce quindi che gli "stati" possibili quindi sono:

{*empty, "c", "co", "con"*}

L'automa, partendo dallo stato *empty*, ricorderà l'ultimo prefisso visto in ingresso fino a riconoscere l'intera frase "con" . In quest'ultimo stato l'automa non può riutilizzare nessun suffisso proprio della frase riconosciuta per costruire una nuova frase quindi dovrà partire a riconoscere la nuova frase dall'inizio (l'ingresso di un nuovo carattere 'c' porta l'automa nello stato 'c').
In definitiva, gli stati possibili sono:

Stato	***Descrizione***
E	Nessuno prefisso in memoria
CON	Parola "con" riconosciuta, nessuno prefisso in memoria
C	Prefisso "c" in memoria
CO	Prefisso "co" in memoria

Gli ingressi possibili, come da specifica, sono:

Ingresso	***Descrizione***
Ɵ	Altro carattere
c	Carattere "C"
o	Carattere "O"
n	Carattere "N"

Le uscite possibili sono:

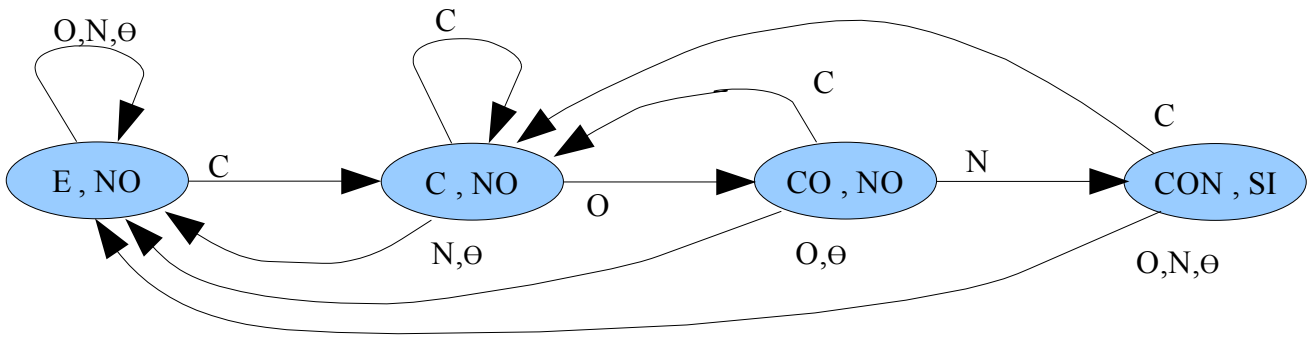
Uscita	***Descrizione***
NO	Parola non riconosciuta
SI	Parola riconosciuta

La STT è così combinata:

	Ingressi				**Uscita**
Stato	**Ɵ**	**c**	**o**	**n**	
E	**E**	**C**	**E**	**E**	**NO**
CON	**E**	**C**	**E**	**E**	**SI**
C	**E**	**C**	**CO**	**E**	**NO**

Stato	*Ingressi*				Uscita
	ϴ	c	o	n	
CO	E	C	E	CON	NO

Lo STG è così combinato:



Definiamo un mapping per ingressi, uscite e stati. Occorrono ceil($\log_2$ 4) = 2 bit per gli ingressi, ceil($\log_2$ 2) = 1 bit per le uscite, ceil($\log_2$ 4) = 2 bit per gli stati.

Stato	*Codifica*
E	**00**
CON	**11**
C	**01**
CO	**10**

Ingresso	*Codifica*
Θ	00
c	01
o	10
n	11

Uscita	*Codifica*
NO	0
SI	1

La STT diventa (riordino le righe secondo la codifica dello stato):

$\delta(S_1S_0)$ = $S_1^*S_0^*$	I_1I_0				$\lambda(S_1S_0)$
S_1S_0	00	01	10	11	O
00	00	01	00	00	0
01	00	01	10	00	0
10	00	01	00	11	0
11	00	01	00	00	1

La funzione di uscita diventa:

$$O = S_1S_0$$

Il suo cammino critico è **1.**
La funzione di transizione stato prossimo diventa:

$$S_1^* = \sim S_1S_0I_1\sim I_0 + S_1\sim S_0I_1I_0 = I_1(\sim S_1S_0\sim I_0 + S_1\sim S_0I_0) = (I_1\{[\sim S_1(S_0\sim I_0)] + [S_1(\sim S_0I_0)]\})$$
$$I_0 = \sim I_1I_0 + S_1\sim S_0I_1I_0 = I_0(\sim I_1 + S_1\sim S_0I_1) = (I_1\{[\sim S_1(S_0\sim I_0)] + [S_1(\sim S_0I_0)]\})$$

Il cammino critico delle due funzioni stato prossimo, come si vede dalla parentesizzazione, è **4**.

Il cammino critico dell'intero automa è pari al cammino critico della funzione stato prossimo più il tempo di setup dei F/F più il tempo di commutazione dei F/F più il cammino critico della funzione di uscita.

[3+3] Tradurre in linguaggio Assembly e poi in linguaggio macchina il seguente spezzone di codice C:

```
i = 0;
for (i=0; i<N; i++) {
        A[i] = s + p + i*2;
        B[i+2] = A[i];
}
```

Alcuni codici operativi:

mnemonico	OP Code	Funct.
lw	35	
sw	43	
add	0	32
jr	0	8
mfhi	0	16
mflo	0	18
mult	0	24
beq	4	
slt	0	42
slti	10	
bne	5	
sub	0	34
addi	8	
j	2	
lui	15	

Alcuni numeri di registro:

$t0	$8
$a0	$4
$v0	$2
$s0	$16
$sp	$29
$r1	$31

Supporre che l'istruzione i = 0, abbia indirizzo in memoria 0x1000.

Supponiamo che i valori di **A, B** ed **N** siano fissi e quindi definiti al momento della compilazione mentre i valori **s** e **p** siano passati al programma tramite **$a0** ed **$a1.** Usiamo **$s0** per sintetizzare la variabile **i**.

```
#realizzo il for
        li $s0 , 0              # uso $s0 per la variabile i        [0x00001000] 0x00008020 add $16, $0, $0
                                                                    [0x00001004]
iniziociclo:                                                                    0x2010000a addi $8, $0, 10
        li $t0 , 10             # definisco N = 10                  [0x00001008] 0x0208082a slt $1, $16, $8
        blt $s0, $t0, eseguiciclo # controllo del for               [0x0000100c] 0x14200002 bne $1, $0, 8 [eseguiciclo-0x0000100c]
        j fineciclo                                                 [0x00001010] 0x08000414 j 0x00001054 [fineciclo]
eseguiciclo:                                                        [0x00001014]
# realizzo A[i] = s + p + i*2;
        li $t0 , 4                                                                      0x20100004 addi $8, $0, 4
        mult $t0 , $s0          # $t0 = i * sizeof(word)            [0x00001018] 0x01100018 mult $8, $16
        mflo $t0                                                    [0x0000101c] 0x00004012 mflo $8
        la $t1 , A              # $t1 = indirizzo di A[0]           [0x00001020] 0x3c01xxxx lui $1, hi(A) [Hi vettA]
                                                                    [0x00001024] 0x2029xxxx addi $9, $1, lo(A) [Lo vettA]
                                                                    [0x00001028] 0x01284820 add $9, $9, $8
        add $t1 , $t1 , $t0     # $t1 = indirizzo di A[i]           [0x0000102c] 0x02105020 add $10, $16, $16
        add $t2, $s0, $s0       # $t2 = i * 2                       [0x00001030] 0x01445020 add $10, $10, $4
        add $t2, $t2 , $a0      # $t2 = s + i*2                     [0x00001034] 0x01455020 add $10, $10, $5
        add $t2, $t2 , $a1      # $t2 = p + s + i*2                 [0x00001038] 0xad2a0000 sw $10, 0($9)
        sw $t2 , 0($t1)         # A[i] = s + p + i*2;
# realizzo B[i+2] = A[i];                                           [0x0000103c] 0x3c01xxxx lui $1, hi(B) [Hi vettB]
        la $t1 , vettB          # $t1 = indirizzo di B[0]           [0x00001040] 0x2029xxxx addi $9, $1, lo(B) [Lo vettB]
                                                                    [0x00001044] 0x01284820 add $9, $9, $8
                                                                    [0x00001048] 0xad2a0008 sw $10, 8($9)
        add $t1 , $t1 , $t0     # $t1 = indirizzo di B[i]           [0x0000104c] 0x22100001 addi $16, $16, 1
        sw $t2 , 8($t1)         # B[i+2] = A[i]                     [0x00001050] 0x08000405 j 0x00001014 [iniziociclo]
        addi $s0 , $s0 , 1      # i++                               [0x00001054]
        j iniziociclo
fineciclo:
```

Calcolo l'opcode di **j 0x00001050**: istruzione di tipo **J, target** uguale ai bit 27-2 di **0x00001050**

OPCode: 6 bit	*Target: 26 bit*
0000 10	**00 0000 0000 0000 0100 0001 0100**

da cui:

j 0x00001050 = 0x08000414
j 0x00001014 = 0x08000405

Calcolo l'opcode di **add $s0, $0 , $0**: istruzione di tipo **R (add rd , rs, rt)**

OPCode: 6 bit	*rs: 5bit*	*rt: 5bit*	*rd: 5bit*	*shmat: 5bit*	*funct: 6bit*
0000 00	**00 000**	**0 0000**	**1000 0**	**000 00**	**10 0000**

da cui:

add $s0, $0 , $0 = 0x00008020
add $t1, $t1 , $t0 = 0x01284820
add $t2, $s0, $s0 = 0x02105020
add $t2, $t2 , $a0 = 0x01445020
add $t2, $t2 , $a1 = 0x01455020

Calcolo l'opcode di **mult $t0, $t0 , $s0**: istruzione di tipo **R (mul rs, rt)**

OPCode: 6 bit	*rs: 5bit*	*rt: 5bit*	*10bit*	*funct: 6bit*
0000 00	**01 000**	**1 0000**	**0000 0000 00**	**01 1000**

da cui:

mult $t0 , $s0 = 0x0110018

Calcolo l'opcode di **mflo $t0**: istruzione di tipo **R (mflo rd)**

OPCode: 6 bit	*10bit*	*rd: 5bit*	*5bit*	*funct: 6bit*
0000 00	**00 0000 0000**	**0100 0**	**000 00**	**01 0010**

da cui:

mult $t0 , $s0 = 0x00004018

Calcolo l'opcode di **addi $t0, $0 , 10**: istruzione di tipo **I (add rt , rs, immediato)**

OPCode: 6 bit	*rs: 5bit*	*rt: 5bit*	*immediato: 16bit*
0010 00	**00 000**	**0 1000**	**0000 0000 0000 1010**

da cui:

addi $t0, $0 , 10 = 0x2008000A
addi $t0, $0 , 4 = 0x20080004
addi $s0, $s0 , 1 = 0x22100001
addi $t1, $1 , xxxx = 0x2029xxxx

Calcolo l'opcode di **lui $1, xxxx**: istruzione di tipo **I (lui rt , immediato)**

OPCode: 6 bit	*5bit*	*rt: 5bit*	*immediato: 16bit*
0011 11	**00 000**	**0 0001**	**xxxx xxxx xxxx xxxx**

da cui:

lui $1, xxxx = 0x3C01xxxx

Calcolo l'opcode di **sw $t2, 0($t1)**: istruzione di tipo **I (sw rt, offset(base))**

OPCode: 6 bit	*base: 5bit*	*rt: 5bit*	*offset: 16bit*
1010 11	**01 001**	**0 1010**	**0000 0000 0000 0000**

da cui:

sw $t2, 0($t1) = 0xAD2A0000
sw $t2, 8($t1) = 0xAD2A0008

Calcolo l'opcode di **slt $1, $s0 , $t0**: istruzione di tipo **R (slt rd , rs, rt)**

OPCode: 6 bit	*rs: 5bit*	*rt: 5bit*	*rd: 5bit*	*shmat: 5bit*	*funct: 6bit*
0000 00	**10 000**	**0 1000**	**0000 1**	**000 00**	**10 1010**

da cui:

slt $1, $s0 , $t0 = 0x0208082A

Calcolo l'opcode di **bne $1, $0, 8**: istruzione di tipo **I (bne rs, rt, offset)**, **offset** = (indirizzo_arrivo-indirizzo_bne)/4

OPCode: 6 bit	*rs: 5bit*	*rt: 5bit*	*offset: 16bit*
0001 01	**00 001**	**0 0000**	**0000 0000 0000 0010**

da cui:

bne $1, $0, 8 = 0x14200002

[9] Scrivere la procedura ricorsiva, ricors, che calcola un numero N a partire dai primi N numeri primi nel seguente modo:
ricors(N) = N*(N-1)*ricors(N-1) + N + 3; ricors(1) = 1

La serie per i primi valori vale:

ricors(1) = 1
ricors(2) = 2*1*ricors(1) + 2 + 3 = 2*1 + 5 = 7
ricors(3) = 3*2*ricors(2) + 3 + 3 = 6*7 + 6 = 48
ricors(4) = 4*3*ricors(3) + 4 + 3 = 12*48 + 7 = 583
ricors(5) = 5*4*ricors(4) + 5 + 3 = 20*583 + 8 = 16688
....

Definiamo una procedura che accetti come argomento **$a0** il numero **N** e restituisca in **$v0** il risultato della ricorsione.
La procedura innanzi tutto deve verificare se si è verificato il caso base della ricorsione, e rispondere immediatamente con il valore del caso base, **ricors(1) = 1** in questo esempio. Altrimenti deve effettuare una chiamata ricorsiva a se stessa e calcolare il valore della formula al ritorno della chiamata.
La procedura deve aver cura di preservare il registro **$ra** nello stack da eventuali chiamate ricorsive a se stessa. Inoltre per effettuare il calcolo della formula, decidiamo di creare due variabili locali alla procedura nello stack in cui memorizzare **N** e **N-1**. Al ritorno dall'eventuale chiamata ricorsiva verranno recuperati i valori e calcolata la formula della ricorsione.
Il codice di questa implementazione è indicato di seguito:

```
ricors:
        # $a0 contiene N

        # crea uno spazio di 3 word nello stack, una per preservare $ra
        # due per creare le variabili locali
        addi $sp, $sp,-12
        sw $ra, 0($sp)          # salva l'indirizzo di ritorno della chiamata

        # check del caso base
        li $t0, 1               # se n> 1 esegui ricorsione
        bgt $a0, $t0, ricorsione

#casobase:
        li $v0, 1               # altrimenti restituisci 1
        j return

ricorsione:
        sw $a0, 4($sp)          # salva N nelle variabili locali (X <- N)
        addi $a0, $a0,-1        # N -> N-1
        sw $a0, 8($sp)          # salva N-1 nelle variabili locali (Y <- N-1)

        # chiamata ricorsiva ricors(N-1)
        jal  ricors

        # calcola ricors(N-1)*N*(N-1) + N + 3
        lw $t0, 4($sp)          # leggo N dalle variabili locali ($t0 <- X)
        lw $t1, 8($sp)          # leggo N-1 dalle variabili locali ($t1 <- Y)
        mul $t2, $t0, $v0       # $t2 = N * ricors(N-1)
        mul $t2, $t1, $t2       # $t2 = (N-1) * N * ricors(N-1)
        add $t2, $t2, $t0       # $t2 = N * (N-1) * ricors(N-1) + N
        addi $t2, $t2, 3        # $v0 = N * (N-1) * ricors(N-1) + N + 3

        move $v0 , $t2          # preparo il valore di ritorno della procedura

return:
        lw $ra, 0($sp)          # ripristina $ra
        addi $sp, $sp, 12       # disalloca lo spazio creato nello stack
        jr $ra                  # ritorna al punto di chiamata
```

[5] Dati i seguenti due programmi

```
Main:
li $v0,2
add $t0, $t1, $t2
j end
jal Proc_A
add $t1, $t1, $t3
lw 0($gp)
...
```

```
Proc_A:
li $v1, 1
sub $t0, $t2, $t3
sw 0($gp)
....
```

Scrivere il programma eseguibile (linked), risultante, sapendo che il segmento del testo del Main è di 1Kbyte e di Proc_A è di 2 Kbyte e che il segmento dati del Main è di 10Kbyte e di Proc_A è di 512byte.

Il segmento di memoria dedicato al codice parte normalmente da 0x00400000. Il linker rilocherà i segmenti Main: e ProcA: a partire da quest'indirizzo, prima Main: e poi ProcA, sistemando le etichette dei salti assoluti ed i riferimenti assoluti all'area dati.
L'area dati parte normalmente da 0x10000000. Come per l'area testo, il linker posizionerà i dati di Main: a partire da quest'indirizzo, seguiti dai dati di ProcA. Lo stack e l'area dati dinamica prenderanno posto di seguito a questa area di dati statici. La memoria dopo la fase di link risulta essere:

indirizzo	***label***	***Descrizione***
....		**S.O.**
0x004000000	Main:	li $v0,2
0x004000004		add $t0, $t1, $t2
0x004000008		j end
		jal Proc_A = jal 0x00400400
		add $t1, $t1, $t3
		lw 0($gp)
		...
0x004000000 + 1KByte -1		*Ultima istruzione Main:*
0x004000000 + 1KByte = 0x00400400	ProcA:	li $v1, 1
		sub $t0, $t2, $t3
		sw 0($gp)
		...
0x004000400 + 2KByte -1		*Ultima istruzione ProcA:*
0x004000400 + 2Kbyte= 0x00400c00		
....		...
0x10000000	Dati_Main:	**....**
0x10000000 + 10Kbyte -1		*Ultimo dato Main:*
0x10000000 + 10Kbyte = 0x10002800	Dati_ProcA:	**...**
0x10002800 + 512byte -1		*Ultimo dato ProcA:*
0x10002800 + 512byte = 0x10002A00		*Area utilizzabile per Heap e Stack*
...		

[3] Come viene indirizzata la memoria del MIPS? Dato un indirizzo della memoria, è univocamente determinata l'istruzione che legge il dato da quell'indirizzo della memoria e lo trasferisce nel register file? Cosa si intende per "register spilling"?

La memoria nel MIPS viene indirizzata per byte, ogni indirizzo si riferisce ad un particolare byte in memoria, eventualmente l'inizio di una word.
Le word sono caricate nella CPU secondo la convenzione *Big-Endian*, *il più grande in fondo, cioè* il byte più significativo in fondo alla sequenza. Contrapposta a questa sta la convenzione *Little-endian, il più piccolo in fondo, cioè* il byte meno significativo in fondo.

Un esempio. Supponiamo che la memoria all'indirizzo **0x10000000** e seguenti ccontenga:

Indirizzo di memoria	Valore in memoria
0x10000000	0x01
0x10000001	0x02
0x10000002	0x03
0x10000003	0x04

e supponiamo di eseguire l'istruzione **lw $t0 , 0($a0)** con **$a0=0x10000000** cioè vigliamo caricare nel registro **$t0** la word che parte da **0x10000000**. Su un microprocessore MIPS, che adotta la convenzione *Big-endian*, alla fine dell'istruzione **$t0** conterrà **0x04030201**. Su un microprocessore Intel-x86, che usa la convenzione *Little-endian* alla fine avremmo **0x01020304**.

Per accedere ai dati in memoria, in lettura o scrittura, nel MIPS sono disponibili le istruzioni di LW ed SW che puntano al byte o alla word di interesse tramite un registro base ed un offset: l'indirizzo finale viene calcolato sommando il contenuto del registro base all valore dell offset. Ne segue che, dato un indirizzo, è possibile costruirlo in vari modi variando in maniera opportuna i valori di base ed offset, ex: **0x1004** -> **base=0x1000** , **offset = 4** o anche **base=0x1004** , **offset = 0**.

Poiché il MIPS ha una quantità di registri limitata (altri tipi di CPU possono ad esempio usare direttamente *pezzi* di memoria al posto dei registri veri e propri) ed è in grado di effettuare operazioni solo tra registri, è necessario nel normale funzionamento muovere periodicamente i dati di interesse dalla CPU verso la memoria e viceversa *(spilling = sgocciolare)* al fine di manternere nella CPU i dati su cui si deve di volta in volta agire.

[2] Cos'è lo stack? Come viene gestito? A cosa serve?

Uno stack è una struttura di memoria utilizzata per memorizzare dinamicamente dati all'occorrenza in cui 'accesso ai dati inseriti avviene in modalita LIFO, Last In First Out: i dati vengono messi nello stack uno impilato sull'altro e è possibile accedere solo alla cima della pila. Se occorre accedere ad un dato interno bisogna prima rimuovere tutti i dati sovrastanti.
Su uno stack sono definite tre operazioni fondamentali:

PUSH(X): Mette un elemento X in cima allo stack
X =POP(): Estraggo l'elementoin cima allo stack (l'elemento viene rimosso dallo stack)
Y=TOP(): Leggo la cima dello stack (l'elemento non viene rimosso dallo stack)

In ambiente assembly, lo stack viene usato per preservare il valore originario dei registri che occorre usare per eseguire i calcoli e per creare variabili locali ad una procedura distinte per ogni chiamata.
Nel MIPS non esistono funzioni dedicate alla creazione e gestione di uno stack; lo stack viene simulato tramite il registro $sp e maneggiato tramite usuali operazioni aritmetiche e di load/store. $sp normalmente viene inizializzato all'ultimo indirizzo disponibile nella memoria fisica, ex. 0x7FFFFFFF. Mano a mano che occorre mettere dati nello stack, il registro $sp viene decrementato di un valore opportuno a creare lo spazio necessario in cima allo stack. Per rimuovere dati dallo stack, il registro $sp viene semplicemente incrementato. Nel MIPS, per il fatto che lo stack viene simulato tramite normali operazioni di load/store in memoria, è possibile accedere non solo alla cima ma anche agli elementi interni dello stack, se necessario.

[2] Indicare le modalità di indirizzamento previste dal MIPS.

Premesso che si parla di indirizzamento anche nel caso di valori usati in calcoli ed indicati per esteso nell'istruzione stessa, le modalità di indirizzamento possibili nel MIPS sono:

- **Indirizzamento a registro:** il valore finale è contenuto in un registro (**jr $ra , add $a0, $a1, $a2**).
- **Indirizzamento immediato:** il valore finale è codificato direttamente nell'istruzione (**addi $a0, $a0 , -1**)
- **Indirizzamento tramite base e offset (diretto con offset):** il valore finale dell'indirizzo è ottenuto sommando il valore di un registro con un valore di offset codificato nell'istruzione (**lw $t0, 8($sp)**)
- **Indirizzamento relativo:** il valore finale dell'indirizzo è ottenuto sommando il valore del Program Counter con un offset codificato nell'istruzione (**bne $t0 $t1, +32**)
- **Indirizzamento pseudo-diretto:** il valore finale dell'indirizzo è codificato per la maggior parte dei bit nell'istruzione. I bit mancanti più significativi, i bit 31-28, vengono copiati dai corrispettivi bit del PC o posti per default a zero, bit 1-0 (**j end**).