1) Scrivere correttamente il codice Assembly che implementi il seguente programma:

b = c – d se c > d;
b = c + d se c <= d;
z = a + b;

Il valore di z dovrà essere calcolato da una procedura chiamata "somma", il valore di b dovrà essere calcolato da un seconda procedura chiamata "opera". Questa seconda procedura dovrà essere chiamata da "somma". Non è necessario scrivere le funzioni di input/output.
Creare il programma eseguibile dai moduli assembly che avete scritto.
Tradurre in linguaggio macchina la procedura che calcola il valore di b.

Soluzione:

La procedura somma richiede in ingresso le variabili **a**, **c** e **d**. Codifichiamo il programma dato in un segmento di pseudo-codice C:

```
void main( ) {
      int z , a , c , d;
      ......                  # assegna ad a , c e d in qualche modo, ex. I/O
      z = somma (a, c, d ) ;
      ......                  # utilizza z in qualche modo, ex. I/O
      return ;
}

int somma( int a , int c , int d ) {
      int b ;                 # per come è strutturato il programma è possibile
      int z ;                 # implementare queste due variabili con il solo
                              # registro di uscita $v0
      b = opera( c, d ) ;
      z = a + b ;
      return z ;
}

int opera( int c, int d ) {
      int b ;                 # stesso discorso della nota precedente vale per
                              # questa variabile
      if ( c > d ) b = c –d ;
      else b = c +d ;
      return b;
}
```

Nelle convenzioni MIPS gli argomenti delle funzioni vengono passate attraverso i registri **$a0**-**$a3** mentre i risultati vengono restituiti nei registri **$v0** e **$v1** (se gli argomenti sono maggiori di 4 e/o i risultati sono maggiori di due occorre usare lo stack come area di passaggio). L'indirizzo di ritorno di una chiamata a procedura sarà presente in **$ra**. Ne segue che per codificare **somma** useremo **$a0** per passare il valore di **a, $a1** per il valore di **c** ed **$a2** per il valore di **d.** Useremo infine **$v0** per passare il valore di z alla procedura chiamante. In maniera analoga per la procedura **opera** useremo **$a0** per passare il valore di **c** ed **$a1** per il valore di **d** mentre useremo **$v0** per restituire il valore di b alla procedura **somma.**
E' possibile codificare **somma** utilizziamo direttamente **$v0** come area di memoria per la variabile **z** e la variabile **b**. Poiché **somma** chiama **opera** per calcolare **b** e poiché il

valore di **a** presente all'interno di **$a0** è necessario dopo la chiamata ad **opera**, occorre salvare nello stack il valore di **$ra** ed il valore di **$a0** prima di invocare **opera.** Al ritorno da **opera** sarà possibile ripristinare i valori originali.

```
# Funzione Somma
# input:  $a0 => a , $a1 => c , $a2 => d,
# output: $vo => z
somma:      addi $sp , $sp , -8          # Salva nello stack $ra ed $a0
            sw $ra , 0($sp)
            sw $a0 , 4($sp)

            # b = opera (c ,d)
            move $a0 , $a1               # prepara gli argomenti per opera
            move $a1 , $a2               # move rd , rs è una pseudo-istruzione eq. a addu rd , rs , $zero
            jal opera                    # call opera() . Il risultato è in $v0 al ritorno dalla chiamata.

            lw $ra , 0($sp)              # Rimette a posto $ra ed $a0
            lw $a0 , 4($sp)
            addi $sp , $sp , 8           # Sistema lo stack

            # z = a + b
            add $v0 , $v0 , $a0

            jr $ra                       # return alla funzione chiamante.
```

La procedura **opera** non ha necessità di salvare nessun registro poiché non usa nessuna procedura esterna. Come per **somma** è possibile realizzare tutti i calcoli direttamente nel registro di uscita **$v0**.

```
# Funzione Opera
# input:  $a0 => c, $a1 => d
# output: $vo => b
            # if (c > d) then b = c - d ; else b = c+d ;
opera:      bgt $a0 , $a1 , then         # c > d ? se si salta a then altrimenti continua (else)
                                         # pseudo-istruzione equivalente a
                                         # slt $at , $a1 , $a0
                                         # bne $at , $zero , then
else:       sub $v0 , $a0 , $a1          # caso else
            j endif
then:       add $v0 , $a0 , $a1          # caso then
endif:      jr $ra                       # return alla funzione chiamante.
end:
```

Supponiamo ora di compilare e linkare i due moduli così costituiti rilocando gli indirizzi nell'area testo partendo dall'indirizzo 0x400000: la tabella dei simboli è

```
somma:      0x00400000          # inizio area testo
opera:      0x0040002c          # inizio di somma + 11 istruzioni
else:       0x00400034          # inizio di opera + 2 istruzioni
then:       0x0040003c          # inizio di opera + 4 istruzioni
endif:      0x00400040          # inizio di opera + 5 istruzioni
end:        x004000044          # fine area testo
```

Il salto relativo presente nella prima pseudo-istruzione di opera (bne $at , $zero , then) salta avanti di due istruzioni. L'offset viene calcolato sottraendo dall'indirizzo di arrivo del salto l'indirizzo dell'istruzione successiva al branch. Ne segue che l'offset del salto sarà uguale a:

offset = then – else = 0x40003c - 0x400034 = 8

Nota: L'offset è espresso in byte. Dato che nel MIPS gli indirizzi di salto sono sempre allineati alle word, ne segue che l'offset sarà sempre un multiplo di 4 CIOè i due bit più significativi dell'offset saranno sempre uguali a 0. Al momento di codificare il branch nella relativa istruzione in linguaggio macchina è quindi conveniente memorizzare i 16 bit 18-3 dell'offset cioè memorizzare l'offset diviso per 4 (shift a destra di 2). Questo permette di avere una più ampia zona di arrivo per i salti relativi. Un discorso analogo può essere fatto per le istruzioni di salto incondizionato.

Traduciamo **opera** in linguaggio macchina MIPS. Il primo passo consiste nel sostituire le pseudo-istruzioni con il relativo codice, i nomi simbolici dei registri con il loro ordinale, le etichette di salto con il relativo indirizzo e le etichette di salto relativo con il rispettivo offset.

```
0x0040002c    slt $1 , $5 , $4          # bgt $a0 , $a1 , then
0x00400030    bne $1, $0 , 8
0x00400034    sub $2 , $4 , $5          # sub $v0 , $a0 , $a1
0x00400038    j 0x00400040              # j endif
0x0040003c    add $2 , $4 , $5          # add $v0 , $a0 , $a1
0x00400040    jr $31                    # jr $ra
```

Ora è possibile codificare ogni singola istruzione costruendo la relativa parola:

slt $1 , $5 , $4 è un istrizione di tipo R quindi la trama dell'istruzione sarà:

op.code	reg.sorg.1	reg.sorg.2	reg.dest	shift	func.
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

L'**opcode** di **slt** è **0**, la **function** è **42=101010$_2$** e lo **shift** è pari a **0**. Quindi:

slt $1 , $5 , $4 = 000000 00101 00100 00001 00000 101010
= 00000000 10100100 00001000 00101010
= 0x00a4082a

bne $1, $0 , 8 è un istruzione di tipo **I**:

op.code	reg.sorg.1	reg.sorg.2	doffset
6 bit	5 bit	5 bit	16 bit

L'**opcode** di **bne** è **5**. **doffset** è uguale all'offset diviso per 4, cioè **2** *(vedi nota precedente circa l'offset del un salto condizionato)*: Quindi:

bne $1 , $0 , 8 **= 000101 00001 00000 0000000000000010**
= 00010100 00100000 00000000 00000010
= 0x14200002

sub $2 , $4 , $5 è un istrizione di tipo R, con **opcode** pari a **0**, **function** pari a **34=100010$_2$** e **shift** pari a **0**.

sub $2 , $4 , $5 **= 000000 00100 00101 00010 00000 100010**
= 00000000 10000101 00010000 00100010
= 0x00851022

j 0x00400040 è un istruzione di tipo **J**:

op.code	daddress
6 bit	26 bit

L'**opcode** di **j** è **2**. **daddress** è pari ai bit da 2 a 28 dell'indirizzo di salto *(vedi nota precedente circa l'offset del un salto condizionato)* , cioè:

0x00400040 = ~~0000~~ 0000 0100 0000 0000 0000 0100 00~~00~~

Quindi:

j 0x00400040 **= 000010 00000100000000000000010000**
= 00001000 00010000 00000000 00010000
= 0x08100010

add $2 , $4 , $5 è un istruzione di tipo R, con **opcode** pari a **0**, **function** pari a **32=100000$_2$** e **shift** pari a **0**.

add $2 , $4 , $5 **= 000000 00100 00101 00010 00000 100000**
= 00000000 10000101 00010000 00100000
= 0x00851020

jr $31 è un istrizione di tipo R, con **opcode** pari a **0**, **function** pari a **8=001000$_2$** e **shift** pari a **0**. Il registro utilizzato per il salto è memorizzato in **reg.sorg.1** mentre reg.sorg.2 e reg.dest sono posti uguali a 0.

jr $31 **= 000000 11111 00000 00000 00000 001000**
= 00000011 11100000 00000000 00001000
= 0x03E00008

Riepologando il codice in linguaggio macchina di **opera** è:

0x0040002c	slt $1 , $5 , $4	**0x00a4082a**
0x00400030	bne $1, $0 , 8	**0x14200002**
0x00400034	sub $2 , $4 , $5	**0x00851022**
0x00400038	j 0x00400040	**0x08100010**
0x0040003c	add $2 , $4 , $5	**0x00851020**
0x00400040	jr $31	**0x03E00008**

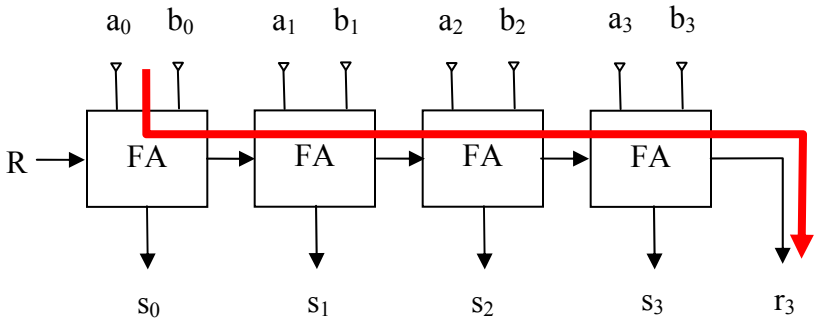
2) Disegnare uno dei possibili circuiti firmware della moltiplicazione intera.

Soluzione: *Vedi slide del corso.*

3) Scrivere la funzione logica implementata da un addizionatore ad anticipazione di riporto a 4 bit e calcolare il risparmio in tempi di cammino critico rispetto ad un addizionatore che non anticipi il riporto.

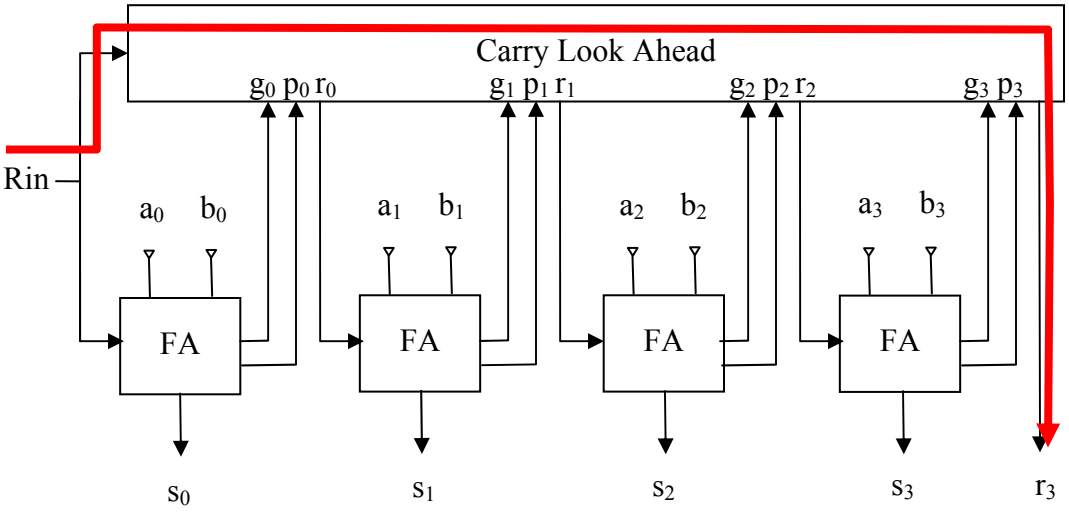
Soluzione:

Sia un sommatore a quattro bit senza anticipazione del riporto composto da quatro Full Adder in cascata.



Ogni Full Adder ha cammino critico pari a 3 e corrispondente al tempo di propagazione degli ingressi verso il riporto in uscita. Ne segue che il cammino critico in questo caso è pari a 3 * 4 = **12**.

Sia ora un sommatore a quattro bit con anticipazione di riporto:



Posto $\mathbf{g_i = a_i b_i}$ e $\mathbf{p_i = a_i \oplus b_i}$ per i = 0..3 , Valgono le seguenti formule:

$\mathbf{r_0 = g_0 + Rin\ p_0}$
$\mathbf{r_1 = g_1 + r_0\ p_1 = g_1 + (g_0 + Rin\ p_0)\ p_1 = g_1 + g_0\ p_1 + Rin\ p_0\ p_1}$
$\mathbf{r_2 = g_2 + r_1\ p_2 = g_2 + (g_1 + g_0\ p_1 + Rin\ p_0\ p_1)\ p_2 = g_2 + g_1\ p_2 + g_0\ p_1\ p_2 + Rin\ p_0\ p_1\ p_2}$
$\mathbf{r_3 = g_3 + r_2\ p_3 = g_3 + (g_2 + g_1\ p_2 + g_0\ p_1\ p_2 + Rin\ p_0\ p_1\ p_2)\ p_3}$
$\mathbf{= g_3 + g_2\ p_3 + g_1\ p_2\ p_3 + g_0\ p_1\ p_2\ p_3 + Rin\ p_0\ p_1\ p_2\ p_3}$

Il calcolo di $\mathbf{g_i}$ e $\mathbf{p_i}$ ha cammino critico pari a **1.** Se consideriamo solo porte a due ingressi otteniamo che:

- il calcolo di $\mathbf{r_0}$ ha cammino critico pari a **3** (il calcolo di $\mathbf{g_i}$ e $\mathbf{p_i}$ più l'attraversamento della porta and e della porta or)**.**
- il calcolo di $\mathbf{r_1}$ ha cammino critico pari a **4** (il calcolo di $\mathbf{g_i}$ e $\mathbf{p_i}$ più due passi per calcolare $\mathbf{g_1 + g_0\ p_1}$ e $\mathbf{Rin\ p_0\ p_1}$ in parallelo più un passo per la or)**.**
- il calcolo di $\mathbf{r_2}$ ha cammino critico pari a **5** (il calcolo di $\mathbf{g_i}$ e $\mathbf{p_i}$ più due passi per calcolare in parallelo i termini della or più due passi la or)**.**
- il calcolo di $\mathbf{r_3}$ ha cammino critico pari a **6** (il calcolo di $\mathbf{g_i}$ e $\mathbf{p_i}$ più tre passi per calcolare in parallelo $\mathbf{g_3 + g_2\ p_3 + g_1\ p_2\ p_3}$, $\mathbf{g_0\ p_1\ p_2\ p_3}$ **e** $\mathbf{Rin\ p_0\ p_1\ p_2\ p_3}$, più due passi per la or)**.**

Considerando che in un sommatore Full Adder vale $\mathbf{s = (a \oplus b) \oplus r}$

- il calcolo di **s0** ha cammino critico pari a 2
- il calcolo di **s1** ha cammino critico pari al cammino critico di **r0** più una ulteriore porta xor, cioè 4.
- il calcolo di **s2** ha cammino critico pari a il tempo di calcolo di **r1 + 1** , cioè 5
- il calcolo di **s3** ha cammino critico pari a il tempo di calcolo di **r2 + 1** , cioè 6

Riepilogando:
il cammino critico di un sommatore con CLA è pari a 6.
il cammino critico di un sommatore senza CLA è pari a 12.

4) Domande

a) Cosa si intende per "big endian" e "little endian" e mostrarne un esempio.

Soluzione:

Un parola binaria composta da più di 8 bit per essere memorizzata in memoria deve essere suddivisa in più byte contigui, es. word di 32 bit devono essere memorizzate usando 4 byte. A seconda di come questi byte vengono ordinati all'interno della memoria si parla di **"big endian"** oppure **"little endian".** Nello schema **"big endian"** il primo byte contiene gli 8 bit meno significativi, il secondo i successivi 8 bit meno significativi e così via fino all'ultimo byte che contiene i bit più significativi della parola ("grande in fondo"). Nello schema **"little endian"** il primo byte contiene gli 8 bit più significativi, il secondo i successivi 8 bit più significativi e così via fino all'ultimo byte che contiene i bit meno significativi della parola ("piccolo in fondo").

Ex: Supponiamo di voler memorizzare la word **0x01020304** nel segmento dati alle locazioni **0x10000000 - 0x10000003:**

Indirizzo Memoria	0x10000000	0x10000001	0x10000002	0x10000003
Big Endian	0x04	0x03	0x02	0x01
Little Endian	0x01	0x02	0x03	0x04

b) Indicare cosa si inserisce nei seguenti registri: $zero, $a2, $v1, $t0, $s0, $gp, $sp, $fp, $ra e spiegarne il motivo.

Soluzione:

L'uso ed il contenuto dei registri del MIPS è determinato, oltre che da particolari istruzioni (si pensi all'effetto dell'istruzione jal su $ra), anche da convenzioni adottate per rendere più semplice ed efficiente la compilazione dei programmi assembly.
Ne segue che:

- **$zero** viene costantemente posto uguale a 0 ed utilizzato principalmente per implementare pseudo-istruzioni non presenti nel set originale del MIPS, come *move rd, rs* che viene tradotto in *addu rd , rs, $zero*
- **$a2** viene convenzionalmente usato per passare il terzo parametro (se necessario) ad una procedura. Una procedura chiamata non è tenuta a preservarne il valore (ne segue che non è garantito che al ritorno da una sottoprocedura il valore di **$a2** sia lo stesso di quello presente prima della chiamata). Analogo discorso vale per tutti i registri **$a0-$a3.** Nel caso siano necessari più di quattro parametri si utilizza lo stack.

- **$v1** viene convenzionalmente usato per restituire (se necessario) il secondo risultato di una procedura. Analogo discorso vale per il registro **$v0.** Nel caso siano necessari più di due risultati di ritorno si utilizza lo stack.
- **$t0** viene convenzionalmente usato per memorizzare risultati temporanei. Una procedura chiamata non è tenuta a preservarne il valore (ne segue che non è garantito che al ritorno da una sottoprocedura il valore di $t2 sia lo stesso di quello presente prima della chiamata). Analogo discorso vale per tutti i registri **$t0-$t9.**
- **$s0** viene convenzionalmente usato per mappare le variabili del programma (ev. scritto in linguaggio ad alto livello) nei registri del MIPS. Una procedura chiamata è tenuta a preservarne il valore presente al momento della chiamata, eventualmente salvandone il valore nello stack e recuperandolo poco prima del salto di ritorno alla procedura chiamante. Analogo discorso vale per tutti i registri **$s0-$s7.**
- **$gp** viene usato per puntare alla metà di un segmento di 64k nell'area dati statica. Viene inizializzato convenzionalmente a 0x10008000 che corrisponde alla metà del primo segmento di 64K presente nell'area dati. Questo permette di caricare ogni dato di questo segmento con una sola istruzione di load con offset (**lw rs , offset($gp)**).
- **$sp** viene usato indicare l'area dello stack utilizzata. Questo registro indica l'indirizzo di memoria dell'ultimo dato inserito nello stack (la *cima*, o *top*, dello stack). Convenzionalmente viene inizializzato all'ultimo indirizzo disponibile di memoria, decrementato verso zero mano a mano che i dati vengono accumulati ed incrementato mano a mano che i dati vengono tolti dallo stack. La gestione di **$sp** non è automatica ma deve essere gestita direttamente dal programma (non esistono nel MIPS istruzioni dedicate ad inserire e togliere dati dallo stack e contemporaneamente aggiustare **$sp**).
- **$fp** viene usato convenzionalmente per puntare all'inizio del frame di attivazione corrente nello stack.
- **$ra** viene posto dall'istruzione **jal** all'indirizzo immediatamente seguente. La procedura chiamata al termine della sua esecuzione può riprendere il flusso di esecuzione dopo la **jal** con l'istruzione **jr $ra.** Si presuppone ovviamente che il registro $ra non sia stato alterato dalla procedura stessa. Nel caso sia necessario chiamare un'ulteriore sottoprocedura occorrerà salvare il valore di **$ra**, ad esempio, nello stack.

c) Indicare alcuni dei più significativi passi storici di sviluppo delle architetture.

Soluzione: *Vedi slide del corso.*

d) Definire quali sono i tempi da considerare per definire la frequenza di clock e perché.

Soluzione: *Vedi slide del corso.*

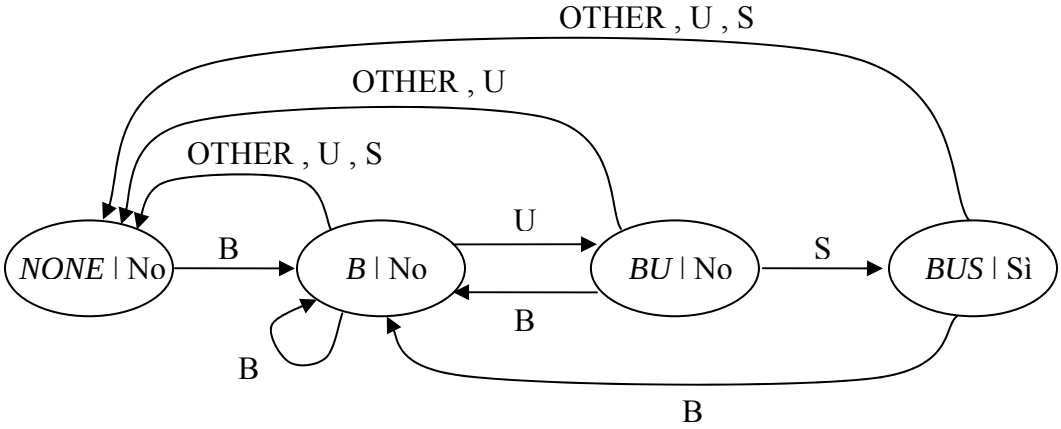
5) Progettare e sintetizzare un controllore (macchina a stati finiti) che riconosce la sequenza "BUS" all'interno di un testo. La macchina è in grado di leggere un carattere alla volta e non distingue tra maiuscolo e minuscolo. Suggerimento: considerare come possibili input 4 insiemi diversi: "B" , "U" , "S" e "Tutti gli altri caratteri".

Soluzione:

La macchina a stati finiti dovrà ricordare ad ogni passo quale prefisso della sequenza da cercare ha già riconosciuto fino a quel momento *(notate che non interessa ricordare tutto quello che è già passato nella macchina ma solo quello che può essere utile per un riconoscimento: sapere che ad un dato tempo la sequenza in ingresso era "ZZZZZZZZZZZZZ" non dà nessuna informazione utile mentre è utile ricordare, ad esempio, che gli ultimi due caratteri letti erano "BU")*. Questo significa che dovrà esistere uno stato per ogni prefisso possibile della stringa da riconoscere. Nel nostro caso, quindi *(uso un nome mnemonico per comodità)*:

Stato	Descrizione	Nome Mnemonico
S_0	Nessun carattere riconosciuto	*NONE*
S_1	Riconosciuto "B"	*B*
S_2	Riconosciuto "BU"	*BU*
S_3	Riconosciuto "BUS"	*BUS*

Suppongo che nel testo la sequenza "**BUS**" possa comparire più di una volta e che la macchina riconosca la sequenza ogni volta. Indico con **OTHER** l'insieme di tutti i caratteri diversi da **B**, **U** ed **S** ed indichiamo per comodità con **No** l'uscita "sequenza non riconosciuta" e con **Sì** l'uscita "sequenza riconosciuta". Lo **State Transition Graph** diventa quindi:



La **State Transtion Table** della macchina è quindi:

Stato	Ingresso B	U	S	OTHER	Uscita
NONE	B	*NONE*	*NONE*	*NONE*	No
B	B	BU	*NONE*	*NONE*	No
BU	B	*NONE*	BUS	*NONE*	No
BUS	B	*NONE*	*NONE*	*NONE*	Sì

Diamo una possibile associazione tra i valori in ingresso codice binario *(dato che non ne viene indicata nessuna nelle specifiche)*. Poiché i possibili valori in ingresso sono quattro occorreranno:

$$\textbf{N.bit = int\_no\_less(} \log_2 4 \textbf{) = 2}$$

Quindi:

Insieme di ingresso	**Codifica**
B	00
U	01
S	10
OTHER	11

Usiamo la seguente codifica per le uscite possibili *(dato che non ne viene indicata nessuna nelle specifiche)*:

Uscita	**Codifica**
Sequenza non riconosciuta	0
Sequenza riconosciuta	1

ed usiamo la seguente arbitraria codifica per gli stati possibili (4 stati = 2 bit ne segue che il circuito fisico di memoria dell'automa sarà composto da 2 latch):

Stato	**Codifica**
NONE	00
B	01
BU	10
BUS	11

Possiamo ora riscrivere la STT usando le codifiche definite sopra:

$Q_0^* Q_1^*$	Ingresso= I_0I_1				
Stato = Q_0Q_1	00	01	10	11	Uscita = O_0
00	01	00	00	00	0
01	01	10	00	00	0
10	01	00	11	00	0
11	01	00	00	00	1

Diamo la funzione tabellare della STT per Q_0^* e Q_1^*:

$Q_0Q_1I_0I_1$	Q_0^*	Q_1^*
0000	0	1
0001	0	0
0010	0	0
0011	0	0
0100	0	1
0101	1	0
0110	0	0
0111	0	0
1000	0	1
1001	0	0
1010	1	1
1011	0	0
1100	0	1
1101	0	0
1110	0	0
1111	0	0

Usiamo la forma SOP per sintetizzare le funzioni stato prossimo Q_0^* e Q_1^*:

$\mathbf{Q_0^*}$ $\mathbf{= \sim Q_0Q_1\sim I_0I_1 + Q_0\sim Q_1I_0\sim I_1}$

$\mathbf{Q_1^*}$ $= \sim Q_0\sim Q_1\sim I_0\sim I_1 + \sim Q_0Q_1\sim I_0\sim I_1 + Q_0\sim Q_1\sim I_0\sim I_1 + Q_0\sim Q_1I_0\sim I_1 + Q_0Q_1\sim I_0\sim I_1$

$= \sim I_0\sim I_1 + Q_0\sim Q_1I_0\sim I_1 = \sim I_1(\sim I_0 + Q_0\sim Q_1I_0)$

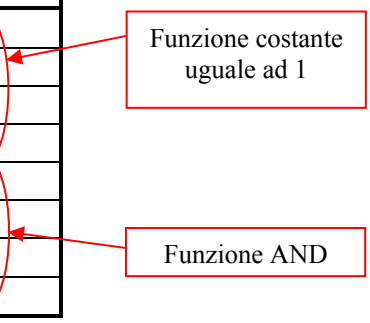
$= \sim I_1(\sim I_0 + Q_0\sim Q_1\sim I_0 + Q_0\sim Q_1I_0) = \mathbf{\sim I_1 (\sim I_0 + Q_0 \sim Q_1)}$

Esaminando la STT si verifica che la funzione di uscita è banalmente:

$$\mathbf{O_0 = Q_0Q_1}$$

6) Data la funzione logica espressa dalla seguente tabella della verità:

xyz	w
000	1
001	1
010	1
011	1
100	0
101	0
110	0
111	1



Sintetizzare la funzione nella prima forma canonica, semplificarla, scrivere l'implementazione mediante porte logiche e calcolare il cammino critico.

Soluzione:

*Anzitutto occore notare che la funzione data assume due particolari andamenti a seconda del valore di **x:** nelle prime quattro righe della tabella (corrispondenti a **x** uguale a 0) la funzione è simile alla funzione costante uguale ad 1 mentre nelle restanti righe (corrispondenti a **x** uguale a 1) la funzione assomiglia alla tabella di verità della funzione AND. Ne segue che durante la semplificazione dovrebbe essere utile tentare di raccogliere sulla **x**.*

Calcoliamo la prima forma canonica SOP.Ad ogni 1 nella funzione corrisponde un min-termine.

xyz	w	
000	1	→ **~X~Y~Z**
001	1	→ **~X~YZ**
010	1	→ **~XY~Z**
011	1	→ **~XYZ**
100	0	
101	0	
110	0	
111	1	→ **XYZ**

Da cui:

$$\mathbf{W = \sim X \sim Y \sim Z + \sim X \sim Y Z + \sim X Y \sim Z + \sim X Y Z + X Y Z}$$

Semplifichiamo:

W = **(~X~Y ~Z + ~X~Y Z)** + ~XY~Z + ~XYZ + XYZ **7a.**

= ~X~Y **(~Z + Z)** + ~XY~Z + ~XYZ + XYZ **4b.**

= ~X~Y **1** + ~XY~Z + ~XYZ + XYZ **1a.**

= ~X~Y + **(~XY~Z + ~XYZ**) + XYZ **7a + 4b + 1a.**

= **(~X~Y + ~XY)** + XYZ **7a + 4b + 1a.**

= ~X + XYZ **8b.**

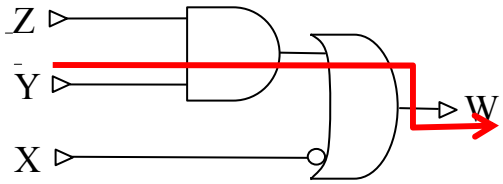
= ~X + (~XYZ + XYZ) **7a + 4b + 1a.**

= ~X + YZ

cioè:

$$W = \sim X + YZ$$

Il circuito logico della funzione è:



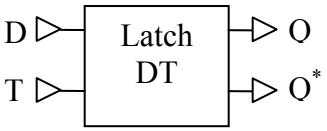
Supponendo il tempo di commutazione di tutte le porte uguale ad un'unità di tempo, ne segue che il tempo di commutazione globale è:

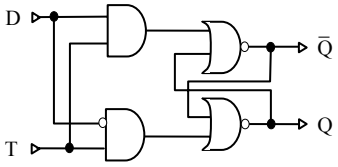
Cammino Critico = 2

7) Scrivere il circuito logico di un bistabile DT, e scrivere la tabella di transizione.

Soluzione:

Un latch DT è un latch SC in cui i segnali S e C vengono generati da un unica linea dati D e controllati attraverso un segnale di controllo T. Quando è presente un segnale attivo sull'ingresso T *(fronte di salita/discesa se il latch è di tipo edge-sensitive o livello alto/basso se il latch è level-sensitive)* il latch memorizza il dato presente sull'ingresso D. Quando non è presente un segnale attivo su T il latch conserva lo stato corrente.





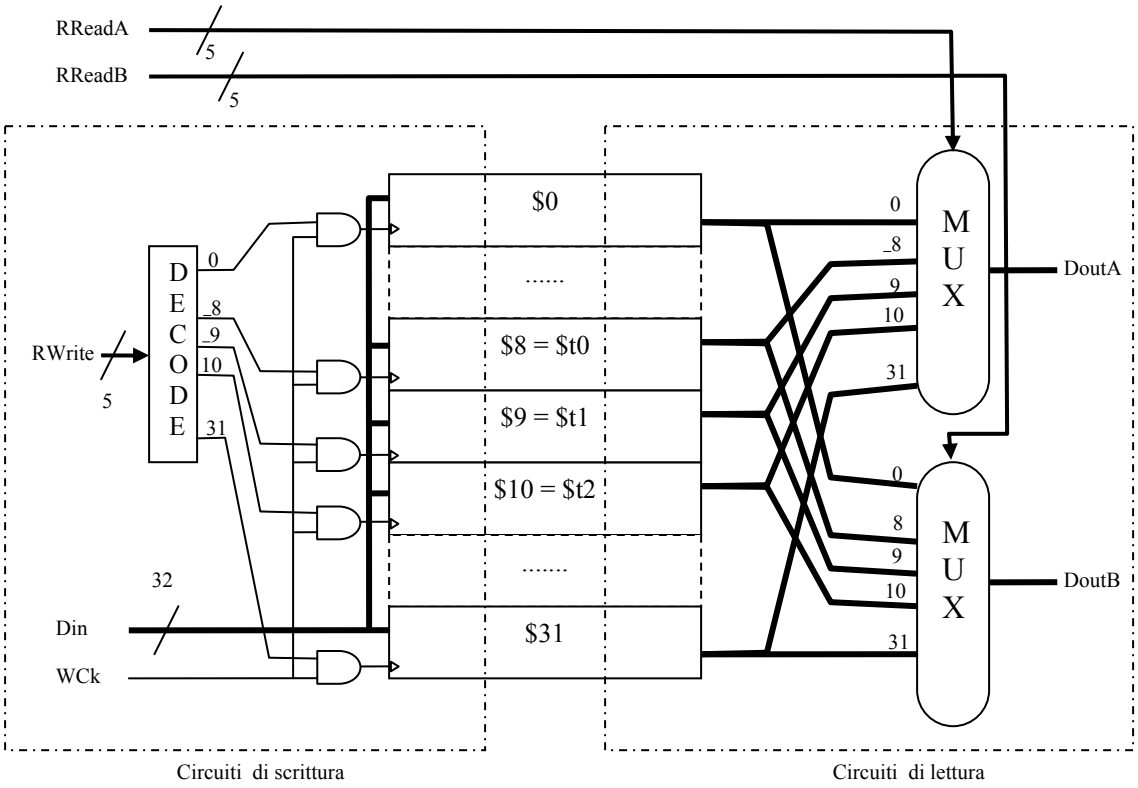
La sua tabella di transizione che indica lo stato prossimo Q^* in funzione di D, T e dello stato corrente è la seguente, $Q^* = f(D, T, Q)$:

TQ	**D = 0**	**D = 1**
00	0	0
01	1	1
10	0	1
11	0	1

8) Disegnare la porta di scrittura del register file e riportare tutti i valori presenti sulle linee del circuito durante l'operazione add $t0 , $t1 , $t2

Soluzione:

Il circuito logico del register file è il seguente:



RReadA e **RReadB** contengono l'indice dei due registri da leggere. Nel nostro caso ad un certo istante conterranno i valori **9** e **10** corrispondenti all'indice dei registri **$t1** e **$t2.** I due segnali pilotano due multiplexer che hanno il compito di selezionare quali registri del register file vanno comunicati in uscita attraverso **DoutA** e **DoutB.**

RWrite contiene l'indice del registro che andrà sovrascritto. Nel nostro caso ad un certo istante conterràil valore **8** corrispondente all'indice del registro **$t0.**

Ad un dato istante sulle uscite della ALU sarà presente il risultato della somma tra **$t1** e **$t2.** Attraverso le linee **Din** questo dato è reso disponibile a tutti i registri. Il segnale di scrittura presente su **WCk** attraverso la selezione operata dal decoder e dalla batteria di AND viene mandato al solo registro selezionato da **RWrite** che potrà quindi memorizzare il dato **Din.**