**COMPITO di ARCHITETTURA DEI CALCOLATORI 1 del 04-03-2013** **MATRICOLA**_____

**COGNOME**_____

**NOME**_____

MODULO RETI LOGICHE:
I SEGUENTI ESERCIZI (1 E 3) VALGONO 50% DEL VOTO FINALE (40/80) PER GLI INFORMATICI (ARCHITETTURA 1) E (1 E 2) IL 33% DEL VOTO FINALE (20/60) PER GLI ALTRI (ARCHTIETTURA 1A)

**Esercizio 1** (per tutti)
Progettare un circuito digitale che evolve ciclicamente secondo la seguente successione numerica:

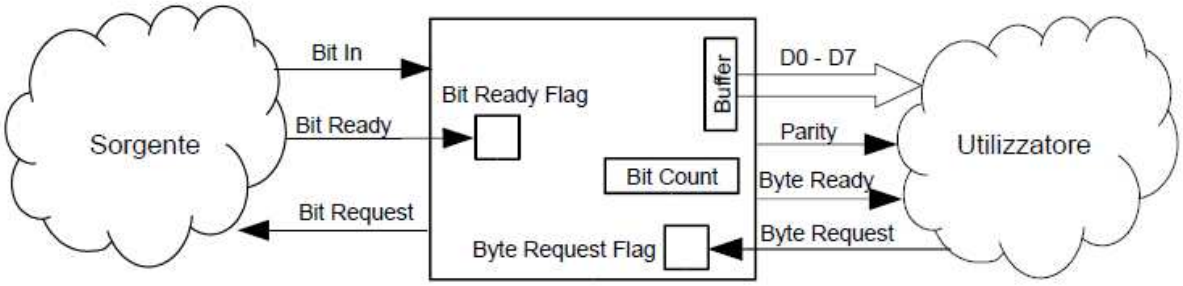
0 2 4 6 8 9 10 11 12 13 14 15 14 13 12 11 10 9 8 6 4 2 0 ...

**Esercizio 2** (per tutti escluso gli Informatici)
Progettare un circuito combinatorio con tre ingressi $A$, $B$, $C$ di $n$ bit, un ingresso $m$ di 1 bit ed una uscita $Z$, che funziona nel seguente modo:

- con $m = 0$ calcola in uscita la differenza $C - \min\{A, B\}$;
- con $m = 1$ calcola in uscita la differenza $\max\{A, B\} - C$.

Calcolare inoltre quanti bit deve avere $Z$ perché tutti i possibili risultati possano essere rappresentati correttamente.

**Esercizio 3** (solo per Informatici)
Progettare un dispositivo di conversione dati collegato da un lato ad una sorgente seriale di dati e dall'altro ad un utilizzatore e funzionante come segue:




a) se il Buffer è pieno, calcola la parità del byte memorizzato nel buffer, fa uscire il contenuto del buffer sulle linee D0-D7, il bit di parità sulla linea Parity ed un bit 1 sulla linea Byte_Ready per segnalare che è pronto un nuovo byte; altrimenti passa al punto d);
b) si pone in attesa ciclica sul flag Byte_Request finché l'utilizzatore segnala di essere pronto a ricevere un nuovo byte;
c) quando il flag Byte_Request diventa 1, lo azzera e azzera anche il contatore Bit_Count;
d) invia un 1 sulla linea Bit_Request per segnalare alla sorgente di trasmettere un bit;
e) si pone in attesa ciclica sul flag Bit_Ready;
f) quando il flag Bit_Ready diventa 1, memorizza nel Buffer il bit sulla linea Bit_In mediante uno shift sinistro di una posizione, incrementa il contatore Bit_Count e azzera il flag Bit_Ready;
g) ritorna al punto a).

MODULO CALCOLATORI ELETTRONICI:
I SEGUENTI ESERCIZI VALGONO 50% DEL VOTO FINALE (40/80) PER ARCHITETTURA 1 E 66% DEL VOTO FINALE (1 E 2) PER ARCHITETTURA 1A. VALGONO 40/40 PER GLI ALTRI.

1. [18] Trovare il codice assembly MIPS corrispondente del seguente programma (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), **rispettando le convenzioni di utilizzazione dei registri dell'assembly MIPS** (riportate in calce, per riferimento). In alternativa, si usi l'assembly x86 anziche' MIPS.Le funzioni non definite sono da considere funzioni esterne al programma.

```
double x[10][10], y[10][10], z[10][10];

void LU(double **a, int n, double **L , double
**U)
{
   int k, i, j;

   for (k=0; k<n; k++) {
      L[k][k] = 1;
      for (i = k+1; i<n; i++) {
         L[i][k] = a[i][k] / a[k][k];
         a[i][k] = L[i][k];
         for (j=k+1; j<n; j++) {
            a [i][j]= a[i][j] - L[i][k] *
a[k][j];
         }
      }
      for (j=k ; j<n; j++) U[k][j]= a[k][j];
}
```

```
main() {
   int m = 10;
   LU(x, m, y, z);
}
```

2. [5] Si modifichi il diagramma a stati di un processore MIPS (o x86) in modo da prevedere la gestione dell'eccezione del caso in cui ci sia una divisione per zero.
3. [5] Si scriva il codice assembly relativo alla gestione dell'eccezione di divisione per zero nel caso di processore MIPS (o x86).
4. [6] Si scriva in assembly il codice di inizializzazione del controllore di DMA che trasferische blocchi di memoria di 512B a partire dall'indirizzo 0x1000 verso l'indirizzo 0xB000 ogni qualvolta si genera l'interrupt 11.
5. [6] Data una TLB di 4 elementi gestita in modalita' LRU, si consideri la seguente sequenza di indirizzi: 0x10001000, 0x12345678, 0x10001008, 0x12348765, 0x10001010, 0x12347658, 0x10001018, 0x1246785, 0x10001020, 0x12347856, 0x10001028, 0x12348567
   Supponendo che la dimensione di pagina sia pari a 4kB, che la memoria fisica totale sia pari ad 1GB e che l'indirizzo base della tabella delle pagine sia 0x20000000, abbinare degli opportuni indirizzi fisici alle pagine virutali e individure il contenuto finale della TLB.

**MIPS instructions**

| Instruction | Example | | Meaning | Comments |
|---|---|---|---|---|
| add | `add` | `$1,$2,$3` | $1 = $2 + $3 | 3 operands; exception possible |
| subtract | `sub` | `$1,$2,$3` | $1 = $2 - $3 | 3 operands; exception possible |
| add immediate | `addi` | `$1,$2,100` | $1 = $2 + 100 | + constant; exception possible |
| subtract immediate | `subi` | `$1,$2,100` | $1 = $2 - 100 | - constant; exception possible |
| multiplication | `mult` | `$1, $2` | Hi,Lo= $1 x $2 | 64-bit Signed Product ; result in Hi,Lo |
| division | `div` | `$1, $2` | Hi= $1 % $2, Lo = $1 / $2 | Signed division |
| move from Hi | `mfhi` | `$1` | $1 = Hi | Create copy of Hi |
| move from Lo | `mflo` | `$1` | $1 = Lo | Create copy of Lo |
| and | `and` | `$1,$2,$3` | $1 = $2 & $3 | 3 register operands; Logical AND |
| or | `or` | `$1,$2,$3` | $1 = $2 \| $3 | 3 register operands; Logical OR |
| nor | `nor` | `$1,$2,$3` | $1 = !($2 \| $3) | 3 register operands; Logical NOR |
| xor | `xor` | `$1,$2,$3` | $1 = $2 ^ $3 | 3 register operands; Logical XOR |
| and immediate | `andi` | `$1,$2,100` | $1 = $2 & 100 | Logical AND register, constant |
| or immediate | `ori` | `$1,$2,100` | $1 = $2 \| 100 | Logical OR register, constant |
| xor immediate | `xori` | `$1,$2,100` | $1 = $2 ^ 100 | Logical XOR register, constant |
| shift left logical | `sll` | `$1,$2,10` | $1 = $2 << 10 | Shift left by constant |
| shift right logical | `srl` | `$1,$2,10` | $1 = $2 >> 10 | Shift right by constant |
| load word | `lw` | `$1,100($2)` | $1 = Memory[$2+100] | Data from memory to register |
| load byte | `lb` | `$1,100($2)` | $1 = Memory[$2+100] | Data from memory to register |
| load byte unsigned | `lbu` | `$1,100($2)` | $1 = Memory[$2+100] | Data from mem. to reg.; no sign extension |
| store word | `sw` | `$1,100($2)` | Memory[$2+100] = $1 | Data from register to memory |
| store byte | `sb` | `$1,100($2)` | Memory[$2+100] = $1 | Data from register to memory |
| load address | `la` | `$1,var` | $1 = &var | Load variable address |
| branch on equal | `beq` | `$1,$2,100` | if ($1 = = $2) go to PC+4+100 | Equal test; PC relative branch |
| branch on not equal | `bne` | `$1,$2,100` | if ($1 != $2) go to PC+4+100 | Not equal test; PC relative |
| set on less than | `slt` | `$1,$2,$3` | if ($2 < $3) $1 = 1; else $1 = 0 | Compare less than; 2`s complement |
| set on less than immediate | `slti` | `$1,$2,100` | if ($2 < 100) $1 = 1; else $1 = 0 | Compare < constant; 2`s complement |
| set on less than unsigned | `sltu` | `$1,$2,$3` | if ($2 < $3) $1 = 1; else $1 = 0 | Compare less than; natural number |
| set on less than imm. unsigned | `sltiu` | `$1,$2,100` | if ($2 < 100) $1 = 1; else $1 = 0 | Compare constant; natural number |
| jump | `j` | `10000` | go to 10000 | Jump to target address |
| jump register | `jr` | `$31` | go to $31 | For switch, procedure return |
| jump and link | `jal` | `10000` | $31 = PC + 4;go to 10000 | For procedure call |
| add.s add.d | `add.x` | `$f0,f2,$f4` | $f0=$f2+$f4 | Single and double precision add |
| sub.s sub.d | `add.x` | `$f0,f2,$f4` | $f0=$f2-$f4 | Single and double precision subtraction |
| mul.s mul.d | `mul.x` | `$f0,f2,$f4` | $f0=$f2*$f4 | Single and double precision multiplication |
| div.s div.d | `div.x` | `$f0,f2,$f4` | $f0=$f2/$f4 | Single and double precision division |
| mov.s mov.d | `mov.x` | `$f0,$f2` | $f0←$f2 | Single and double precision move |
| abs.s abs.d | `abs.x` | `$f0,$f2` | $f0=ABS($f2) | Single and double precision absolute value |
| neg.s neg.d | `neg.x` | `$f0,$f2` | $f0= – ($f2) | Single and double precision absolute value |
| c.lt.s c.lt.d (eq,ne,le,gt,ge) | `c.lt.x` | `$f0,$f2` | Temp=($f0<$f2) | Single and double: compare $f0 and $f2 <,=,!=,<=,>,>= |
| mtc1 (mfc1) | `mtc1` | `$1,$f2` | $f2=$1 | Data from gen.reg. to C1 reg. (no conversion) (and viceversa) |
| branch on false | `bc1f` | `label` | If (Temp = = false) go to label | Temp is 'Condition-Code' |
| branch on true | `bc1t` | `label` | If (Temp = = true) go to label | Temp is 'Condition-Code' |
| load floating point (32bit) | `lwc1` | `$f0,0($1)` | $f0←Memory[$1] | |
| store floating point (32bit) | `swc1` | `$f0,0($1)` | Memory[$1]←$f0 | |
| convert single into double | `cvt.d.s` | `$f0,$f2` | $f0=(double)$f2 | Also cvt.s.d (viceversa) |
| convert single into integer | `cvt.w.s` | `$f1,$f0` | $f1=(int)$f0 | Also cvt.s.w (viceversa) |

**Register Usage**

| Name | Register Num. | Usage |
|---|---|---|
| $zero | 0 | The constant value 0 |
| $s0-$s7 | 16-23 | Saved |
| $t0-$t9 | 8-15,24-25 | Temporaires |
| $a0-$a3 | 4-7 | Arguments |

| Name | Register Num. | Usage |
|---|---|---|
| $v0-$v1 | 2-3 | Results |
| $fp, $sp | 30,29 | frame pointer, stack pointer |
| $ra, $gp | 31,28 | return address, global pointer |
| $k0-$k1 | 26,27 | Kernel usage |

| Name | Usage |
|---|---|
| $f0, $f1, …, $f31 | Single precision floating point registers |
| $f0, $f2, …, $f30 | Double precision floating point registers |

**System calls**

| Service Name | Service Num. ($v0) | INPUT Arguments | OUTPUT Arguments |
|---|---|---|---|
| print_int | 1 | $a0=integer to print | --- |
| print_float | 2 | $f12=float to print | --- |
| print_string | 4 | $a0=address of ASCIIZ string to print | --- |
| Sbrk | 9 | $a0=Number of bytes to be allocated | $v0=pointer to the allocated memory |