COMPITO di ARCHITETTURA DEI CALCOLATORI del 10-02-2016

MATRICOLA______________________

COGNOME_________________________

NOME__________________________

DA RESTITUIRE INSIEME AGLI ELABORATI e A TUTTI I FOGLI
→ NON USARE FOGLI NON TIMBRATI
→ ANDARE IN BAGNO <u>PRIMA</u> DELL'INIZIO DELLA PROVA

SVOLGIMENTO DELLA PROVA:
PER GLI STUDENTI DI "ARCHITETTURA DEI CALCOLATORI – A.A. 2015/16": es. N.1 + es. N.3 + es. N.4
PER GLI STUDENTI DEGLI ANNI PRECEDENTI che devono svolgere sia il modulo CALCOLATORI che il modulo RETI: es. N.1,2,3,5
PER GLI STUDENTI DEGLI ANNI PRECEDENTI che devono svolgere SOLO il modulo CALCOLATORI es. N.1,2,3.
PER GLI STUDENTI DEGLI ANNI PRECEDENTI che devono svolgere SOLO il modulo RETI: es. N.4,5

NOTA: per l'esercizio1 (e analogamente per l'esercizio 4) dovranno essere consegnati due files: il file del programma MIPS (ovvero VERILOG) e il file relativo all'output (screenshot o copy/paste)

---

1. [18] Utilizzando il simulatore SPIM, codificare in assembly MIPS il seguente codice (**utilizzando solo e unicamente istruzioni dalla tabella sottostante**), **rispettando le convenzioni di utilizzazione dei registri dell'assembly MIPS** (riportate in calce). Al termine della codifica consegnare 2 files: il programma in MIPS e l'output relativo.
   Nota: le funzioni main, seidel2, report, test_convergence e compute sono date (stampate in fondo al testo e fornita copia elettronica).

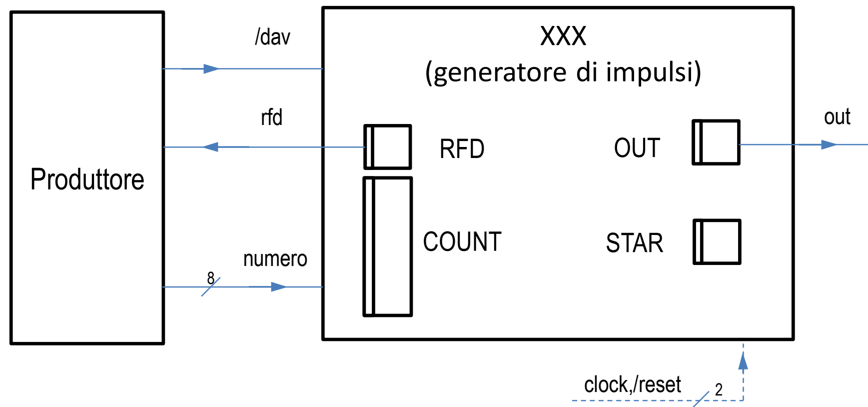
```
#define N 3
#define MAXITER 20
#define EPSILON 0.0001

float A[N*N] = {  4.0, -1.0, -1.0,
                 -2.0,  6.0,  1.0,
                 -1.0,  1.0,  7.0 };
float B[N] = { 3.0, 9.0, -6.0 };
float e2, x[N], y[N], c[N], R[N*N], T[N*N];
int iter = 0;

void setup() {
   int i, j, k;
   float t;
   for (i = 0; i < N; ++i) {
      x[i] = 0.0;
      for (j = 0; j < N; ++j) {
         t = 0.0;
         if (i>j) for (k = j; k < i; ++k)
            t -= A[i*N+k] * T[k*N+j];
         if (i==j) t = 1.0;
         T[i*N+j] = (i<j) ? 0.0 : t / A[i*N+i];
      }
   }
   for (i = 0; i < N; ++i) {
      for (j = 0; j < N; ++j) {
         t = 0.0;
         for (k = 0; k < N; ++k)
            if (k<j) t += T[i*N+k] * A[k*N+j];
         R[i*N+j] = t;
      }
      t = 0.0;
      for (k = 0; k < N; ++k)
         t += B[k] * T[i*N+k];
      c[i] = t;
   }
}
```

```
void seidel2(float *x, float *y, float *a, float c) {
   int j;
   *x = c;
   for (j = 0; j < N; ++j) *x -= a[j] * y[j];
}

void report() {
    int i;
    print_string("X: ");
    for (i = 0; i < N; ++i) {
       print_float(x[i]); print_string(" ");
    }
    print_string(" - iter="); print_int(iter);
    print_string("  e2="); print_float(e2);
    print_string("\n");
}
int test_convergence() {
    int j, r;
    ++iter; e2 = 0;
    for (j = 0; j < N; ++j)
       e2 += (y[j] - x[j]) * (y[j] - x[j]);
    r = (e2 < (EPSILON * EPSILON) || iter == MAXITER);
    report();
    return I;
}

int compute() {
    int i;
    do {
       for (i = 0; i < N; ++i) seidel2(&y[i], x, &R[N*i], c[i]);
       for (i = 0; i < N; ++i) seidel2(&x[i], y, &R[N*i], c[i]);
    } while (!test_convergence());
}

int main()
{
   setup(); compute(); report(); exit(0);
}
```

2. [7] Si consideri una cache di dimensione 128B e a 4 vie di tipo write-back/write-non-allocate. La dimensione del blocco e' 16 byte, il tempo di accesso alla cache e' 4 ns e la penalita' in caso di miss e' pari a 40 ns, la politica di rimpiazzamento e' LRU. Il processore effettua i seguenti accessi in cache, ad indirizzi al byte: 177, 1163, 223, 2181, 200, 3221, 175, 1184, 2182, 3201, 4176, 8173, 2176, 9183, 8251, 4176, 2201, 3180, 5171, 7178, 3191, 181. Tali accessi sono alternativamente letture e scritture. Per la sequenza data, ricavare il tempo medio di accesso alla cache, riportare i tag contenuti in cache al termine, i bit di modifica (se presenti) e la lista dei blocchi (ovvero il loro indirizzo) via via eliminati durante il rimpiazzamento ed inoltre in corrispondenza di quale riferimento il blocco e' eliminato.

3. [5] Per la funzione seidel2 dell'esercizio 1 determinare il tempo di esecuzione del codice assumendo che giri su un processore MIPS senza pipeline con frequenza di clock pari a 5 GHz e con i seguenti valori per il CPI di ciascuna categoria di istruzioni: aritmetico-logiche-salti 1, branch 3, load-store 10, floating-point 5.

4) [10] L'unità XXX preleva dal Produttore un byte 'numero' (8-bit) e genera sull'uscita 'out' un impulso con una durata N cicli di clock essendo N il numero naturale prelevato su 'numero'. Per gestire 'numero' viene effettuato l'handshake tramite la coppia di segnali '/dav' (data valid – attivo basso) e 'rfd' (ready for data).

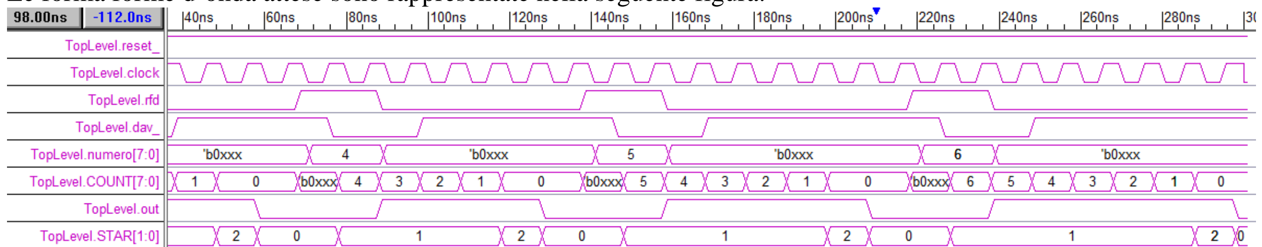


Handshake /dav, rfd: partendo da una condizione inziale in cui /dav è 1 (per indicare che nessun nuovo byte è fornito dal produttore attraverso il segnale 'numero') e rfd è 1 (per indicare che la rete XXX è disponibile a prelevare ed elaborare un nuovo byte), il produttore agisce per primo presentanto un nuovo byte attraverso il segnale 'numero' e notifica questo fatto ponendo /dav a 0. Dopodiché la rete XXX preleva il byte e pone rfd=0 (per indicare che non è ulteriormente disponibile a prelevare altri byte) e inizia ad elaborare il byte prelevato. Il produttore può allora riportare /dav a 1 e attendere che il consumatore, utilizzando tale byte, riporti rfd a 1 (ripristino delle condizioni iniziali).

Uso da parte di XXX del byte prelevato: XXX intepreta il byte prelevato come un numero naturale N e porta a 1 la variabile di uscita out (che normalmente vale 0) per N periodi di clock se N != 0 e per 256 periodi di clock se N=0.

Si descriva e si sintetizzi l'unità XXX e se ne tracci l'evoluzione nell'ipotesi che il Produttore fornisca le rappresentazioni di (numero)= (4), (5), (6) indicando chiaramente nel grafico tali rappresentazioni (il codice Verilog del produttore e del top-module e' fornito nell'ultima pagina del testo).

Le forma forme d'onda attese sono rappresentate nella seguente figura:



5) [12] Sintetizzare una rete sequenziale utilizzando il modello di Moore con un ingresso X su due bit e una uscita Z su singolo bit che si comporta nel seguente modo: se X1=0 e all'ingresso X0 si presenta la successione di quattro bit 0101 allora l'uscita diventa 1; se X1=1 e all'ingresso X0 si presenta la successione di quattro bit 1011, l'uscita diventa 1; in tutti gli altri casi l'uscita è zero. Se X1 cambia da 0 a 1 o viceversa, la rete abbandona l'esame della successione corrente e passa ad esaminare l'altra. Rappresentare la macchina a stati finiti per tale riconoscitore, la tabella delle transizioni, le equazioni booleane delle reti CN1 e CN2 e il circuito sequenziale sincronizzato basato su flip-flop D.

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 unconditional	`b`	`100`	go to PC+4+100	PC relative branch
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
no operation	`nop`		Do nothing	Do nothing
load-linked	`ll`	`$1,100($2)`	$1=Memory[$2+100]	Read and start to monitor the given memory location
store-conditional	`sc`	`$1,100($2)`	Memory[$2+100]=$1 or →	return 0 if a coherence action happens since the previous ll ($1 must be different from 0)
add.s add.d	`add.x`	`$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 opposite 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	`mtc1`	`$1,$f2`	$f2=$1	Data from gen.reg. $1 to C1 reg. $f2 (no conversion)
mfc1	`mfc1`	`$f1,$1`	$1=$f2	Data from gen.reg. to C1 reg. (no conversion)
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[$1]	Data from FP (C1) register to memory
store floating point (32bit)	`swc1`	`$f0,0($1)`	Memory[$1]←$f0	Data from memory to FP (C1) register
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	Reg. 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	Reg.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

Reg. Num.	Usage
$f0, $f2	Return values
$f12,$f14	Function arguments
$f20,$f22,$f24,$f26,$f28,$f30	Saved registers
$f4,$f6,$f8,$f10,$f16,$f18	Temporaries registers

System calls

Service Name	Service Num. ($v0)	INPUT Arguments	OUTPUT Arguments
print_int	1	$a0=integer to print	---
print_float	2	$f12=float to print	---
print_double	3	($f12,$f13)=double to print	---
print_string	4	$a0=address of ASCIIZ string to print	---
read_int	5	---	$v0=integer
read_float	6	---	$f0=float
read_double	7	---	$f0-f1=double
read_string	8	$a0=address of input buffer, $a1=max characters to read	---
sbrk	9	$a0=Number of bytes to be allocated	$v0=pointer to the allocated memory
exit	10	---	---

```
module TopLevel;
  reg reset_; initial begin reset_=0;  #1 reset_=1; #300; $stop; end
  reg clock ; initial clock =0; always #5clock <=(!clock);
  wire rfd, dav_;
  wire[7:0] numero;
  wire[7:0] COUNT=Xxx.COUNT; wire out;
  wire[1:0] STAR=Xxx.STAR;
  XXX Xxx(rfd, dav_,numero, out, clock,reset_);
  Produttore  PRO(rfd,dav_,numero);
endmodule


module Produttore(rfd, dav_,numero);
 input          rfd;
 output         dav_;
 output [7:0]  numero;
 reg DAV_;    assign dav_=DAV_;
 reg [2:0]    APP1_X, APP2_X; assign numero=APP1_X;
 initial begin APP2_X='B00000011; DAV_=1; end
 always
   begin #5; wait(rfd==1); #3 APP1_X=APP2_X; APP2_X=APP2_X+1;
         #5 DAV_=0; wait(rfd==0); #1 APP1_X='HXX; #9 DAV_=1;end
endmodule
```

```
seidel2:
#________NO CALL FRAME___________
# INPUT: a0=*x, a1=*y, a2=*a, f12=c
swc1 $f12, 0($a0)        # store c into *x
add  $t0, $0, $0            # t0=j=0
Seidel2_start_for1:
  slti $t9, $t0, 3        # N <? 3
  beq $t9, $0, Seidel2_end_for1
  #________________________________
  sll   $t3, $t0, 2     # j*sizeof(float)
  add   $t4, $t3, $a2    # &a[j]
  lwc1  $f4, 0($t4)
  add   $t4, $t3, $a1    # &y[j]
  lwc1  $f6, 0($t4)
  mul.s $f8, $f4, $f6   # a[j]*y[j]
  lwc1  $f10, 0($a0)    # load *x
  sub.s $f10, $f10, $f8   # *x-=
  swc1  $f10, 0($a0)    # store *x
  #________________________________
  addi $t0, $t0, 1
  j Seidel2_start_for1
Seidel2_end_for1:
jr $ra

report:
#________CALL FRAME___________
# saved variables: s0       4B
# ra                        4B
#___________________Totale  8B
addi $sp, $sp, -8
sw   $s0, 4($sp)
sw   $ra, 0($sp)
addi $v0, $0, 4      # print "X: "
la   $a0, x_str
syscall
add  $s0, $0, $0    # s0=j=0
Report_start_for1:
  slti $t9, $s0, 3    # j <? N
  beq  $t9, $0, Report_end_for1
  #________________________________
  sll  $t3, $s0, 2  # t3=j*sizof(float)
  la   $t4, x
  add  $t4, $t4, $t3  # t4= x[j]
  lwc1 $f12, 0($t4)
  addi $v0, $0, 2    # print x[j]
  syscall
  addi $v0, $0, 4    # print " "
  la $a0, spazio
  syscall
  #________________________________
  addi $s0, $s0, 1
  j Report_start_for1
Report_end_for1:
addi $v0, $0, 4    # print " - iter="
la $a0, iter_str
syscall
addi $v0, $0, 1    # print iter
la $t0, iter
lw $a0, 0($t0)
syscall
addi $v0, $0, 4    # print "  e2="
la $a0, e_str
syscall
addi $v0, $0, 2
la $t0, e2      # print e2
lwc1 $f12, 0($t0)
syscall
addi $v0, $0, 4    # print "\n"
la $a0, ret
syscall
lw    $s0, 4($sp)
lw    $ra, 0($sp)
addi $sp, $sp, 8
jr $ra

test_convergence:
#________CALL FRAME___________
# ra           4B
#___________________Totale  4B
addi $sp, $sp, -4
sw   $ra, 0($sp)
la   $t0, iter
lw   $t1, 0($t0)
addi $t1, $t1, 1    # ++iter
sw   $t1, 0($t0)
la   $t0, e2      # &e2
sw   $0, 0($t0)      # e2=0
add  $t0, $0, $0    # t0=j=0
Test_Convergence_start_for:
  slti $t9, $t0, 3    # j <? N
  beq  $t9, $0, Test_Convergence_end_for
  #________________________________
  la     $t2,  x
  la     $t3,  y
  sll   $t5,  $t0,  2    # j*sizeof(float)
  add   $t2,  $t2,  $t5  # &x[j]
  lwc1  $f4,  0($t2)
  add   $t3,  $t3,  $t5  # &y[j]
  lwc1  $f6,  0($t3)
  sub.s $f8,  $f6,  $f4  # y-x
  mul.s $f8,  $f8,  $f8   # ()^2
  la     $t6,  e2
  lwc1  $f10, 0($t6)    # load e2
  add.s $f10, $f10, $f8  # e2+=()
  swc1  $f10, 0($t6)    # store e2
  #________________________________
  addi $t0, $t0, 1
  j Test_Convergence_start_for
Test_Convergence_end_for:
jal report
la     $t6,  e2
lwc1   $f10, 0($t6)
la     $t0, EPSILON
lwc1   $f4, 0($t0)
mul.s  $f4, $f4, $f4   # EPSILON^2
c.lt.s $f10, $f4      # e2 < eps
add    $v0, $0, $0    # r=0
bc1f   Test_Convergence_c1_f
#________________________________
  addi   $v0, $0, 1
  j      Test_Convergence_c2_end
Test_Convergence_c1_f:
  #________________________________
  la   $t0, iter
  lw   $t0, 0($t0)
  slti $t2, $t0, 20   # iter ==? MAXITER(20)
  bne  $t2, $0, Test_Convergence_c2_end
  addi $v0, $0, 1
  #________________________________
#________________________________
Test_Convergence_c2_end:
lw   $ra, 0($sp)
addi $sp, $sp, 4
jr   $ra

compute:
#________CALL FRAME___________
# saved variables: s0       4B
# ra                        4B
#____________________Totale 8B
addi $sp, $sp, -8
sw    $s0, 4($sp)
sw    $ra, 0($sp)
Compute_start_do:
  add $s0, $0, $0      # s0=i=0
  Compute_start_for_1:
    slti $t9, $s0, 3    # i <? N
    beq  $t9, $0, Compute_end_for_1
    #________________________________
    sll  $t3, $s0, 2   # t3=i*sizeof(float)
    la   $a0, y
    add  $a0, $a0, $t3  #
a0=&y+i*sizeof(float)
    la   $a1, x      # a1=&x
    addi $t1, $0, 3    # N=3
    mult $t3, $t1    # N*i*sizeof(float)
    mflo $t0      #
    la   $a2, R
    add  $a2, $a2, $t0  #
a2=R+N*i*sizeof(float)
    la   $t4, c      # t4=&c[0]
    add  $t4, $t4, $t3  # t4=&c[i]
    lwc1 $f12, 0($t4)
    jal seidel2
    #________________________________
    addi $s0, $s0, 1
    j Compute_start_for_1
  Compute_end_for_1:
  add $s0, $0, $0      # s0=i=0
  Compute_start_for_2:
    slti $t9, $s0, 3    # i <? N
    beq  $t9, $0, Compute_end_for_2
    #________________________________
    sll  $t3, $s0, 2  # t3=i*sizeof(float)
    la   $a0, x
    add  $a0, $a0, $t3  #
a0=&x+i*sizof(float)
    la   $a1, y      # a1=&y
    addi $t1, $0, 3    # N=3
    mult $t3, $t1    # N*i*sizeof(float)
    mflo $t0      #
    la   $a2, R
    add  $a2, $a2, $t0  #
a2=R+N*i*sizeof(float)
    la   $t4, c      # t4=&c[0]
    add  $t4, $t4, $t3  # t4=&c[i]
    lwc1 $f12, 0($t4)
    jal seidel2
    #________________________________
    addi $s0, $s0, 1
    j Compute_start_for_2
  Compute_end_for_2:
  jal test_convergence # Output:v0
  beq $v0, $0, Compute_start_do
Compute_end_do:
lw  $s0, 4($sp)
lw  $ra, 0($sp)
addi $sp, $sp, 8
jr $ra

main:
#________NO CALL FRAME___________
jal setup
jal compute
jal report
addi $v0, $0, 10
syscall
```