Esercizio 0 (2021-03-31)

Si realizzi in assembly MIPS il seguente programma. Sia data in memoria un array `A` di half-word di lunghezza `L`. Si assume che `L` sia un numero tra 1 e 32.

Di ogni half-word in `A` (esempio, `h = A[0] = 0x5BE1 = 0101 1011 1110 0001`) si vuole calcolare il bit di (even) parity – ossia, lo XOR in sequenza dei bit: `XOR( … (XOR( XOR( XOR( h[32], h[31] ), h[30] ), h[29] ), … h[0]))…)` . Si vuole restituire una parity word che abbia all'i-esimo bit il parity bit dell'i-esima half-word. Inoltre, a valle della computazione, si vuole calcolare il parity bit della parity word stessa.

Se, ad esempio, L è 3 ed A è un array come segue:
`A[2]: 0x002A = 0000 0000 0010 1010`
`A[1]: 0xA5FF = 1010 0101 1111 1111`
`A[0]: 0x5BE1 = 0101 1011 1110 0001`
i parity bit di `A[2]`, `A[1]` e `A[0]` saranno, rispettivamente `1`, `0` e `1`. Per ciò, `1`, `0` e `1` saranno i bit di indice 2, 1 e 0 della parity word restituita. Assumiamo che il resto dei bit più significativi sia per default `0`. La parity word risultante è dunque
`0000 0000 … 0000 0101 = 0x5`.
Il parity bit della parity word è `0`.

Il programma assembly deve definire nella sezione dedicata al **data segment** l'array (`A`) e la lunghezza (`L`).

Nella sezione dedicata al **text segment**, il programma deve avere nel comparto ***main*** il caricamento dei dati da memoria, la chiamata ad una funzione **`parity`** definita di seguito e la stampa a terminale della stringa di bit della parity word prodotta dalla funzione (nell'esempio, `0000 0000 … 0101`). Il codice necessario per la *syscall* è spiegato in fondo. La stampa a terminale *non* deve avvenire all'interno della funzione ma nel comparto main chiamante.

La funzione **`parity`** accetta come argomenti:
– `$a0`: l'indirizzo base dell'array `A`;
– `$a1`: la lunghezza `L`;
e restituisce come risultato:
– `$v0`: la parity word (nell'esempio, `0x5`)
– `$v1`: il parity bit della parity word generata (nell'esempio, 0).

Note: Commentare *ogni riga di codice* avendo cura di spiegare a cosa servano i registri. Una soluzione ricorsiva sarà premiata con un bonus di 1 punto. Si ricorda che la syscall per la stampa di un intero come una stringa di bit prevede come argomenti `$v0` = 35 e `$a0` assegnato con la word da stampare.

Esercizio 1 (2020-10-27)

Sia data una CPU con processore a **4 GHz** e **8 CPI** (Clock Per Instruction) che adoperi indirizzi da 32 bit e memoria strutturata su due livelli di **cache** (L1, L2), il cui setup è come segue:
L1 è una cache **set-associativa** a **4 vie** con **8 set** e **blocchi da 4 word**; adopera una politica di rimpiazzo **LRU.**
L2 è una cache **direct-mapped** con **4 linee** e **blocchi da 16 word**.

1) Supponendo che all'inizio nessuno dei dati sia in cache, indicare quali degli accessi in memoria indicati di seguito sono HIT o MISS in ciascuna delle due cache. Per ciascuna **MISS** indicare se sia di tipo Cold Start (**Cold**), Capacità (**Cap**) o Conflitto (**Conf**). Utilizzare la tabella sottostante per fornire i risultati ed indicare la metodologia di calcolo più in basso.

	Address	800	824	828	832	790	1640	836	848	1240	1244	1248	1252
L1	**Block#**												
	Index												
	Tag												
	HIT/MISS												
	Miss type												
L2	**Block#**												
	Index												
	Tag												
	HIT/MISS												
	Miss type												

2) Calcolare le dimensioni in bit (compresi i bit di controllo ed assumendo che ne basti uno per la LRU) delle due cache: (a) L1 e (b) L2.

3) Assumendo che gli accessi in **memoria** impieghino **200 ns**, che gli **hit** nella cache **L1** impieghino **2 ns** e gli **hit** nella cache **L2** impieghino **40 ns**, calcolare (a) il **tempo totale** per la sequenza di accessi, (b) il tempo **medio** per la sequenza di accessi, e (c) **quante istruzioni** vengono svolte nel tempo medio calcolato.

Esercizio 2 (2021-03-31)

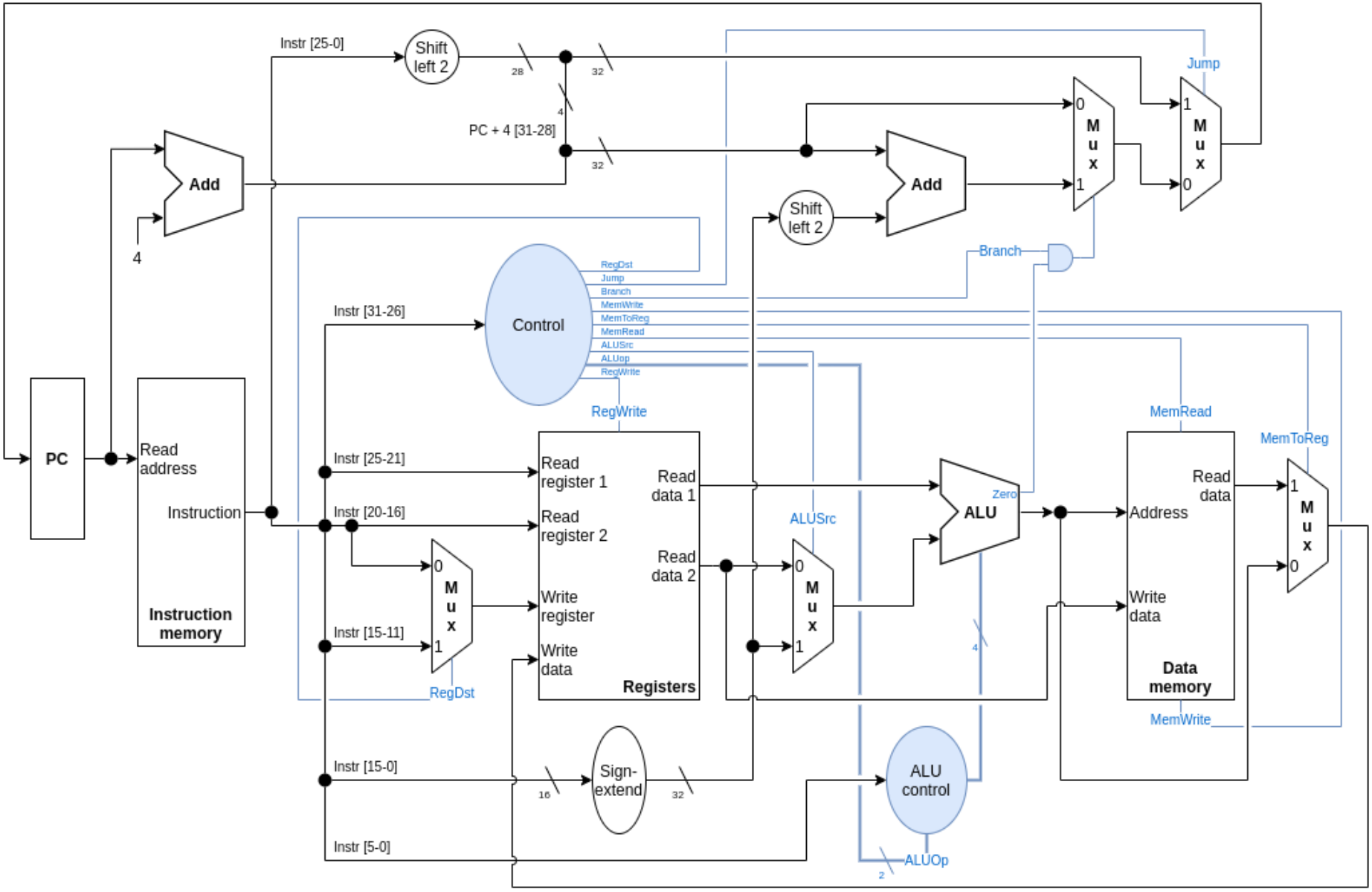
Considerare l'architettura MIPS a ciclo singolo nella figura in basso (ed in allegato).
Si vuole aggiungere l'istruzione di tipo R "sum register values with a word in memory"
`sum_w_mem $rs, $rt, $rd, mem_addr`
(laddove mem_addr è indicato dal campo di 5 bit tipicamente assegnato allo `shamt`). L'istruzione deve salvare nel registro `rd` la somma di
1) `$rs` **+**
2) `$rt` **+**
3) la word in memoria all'indirizzo `mem_addr × 4 + 0x1000 0000` (il calcolo fa sì che l'indirizzo ricada nel *data segment*).

Supponiamo, per esempio, che il valore nel registro `rs` sia `0x40`, il valore nel registro `rt` sia `0x8`, e che `mem_addr` sia `0x13`. Dunque, la word da caricare sarà all'indice `0x13 × 4 + 0x1000 0000 = 0x1000 004C`; supponiamo che tale word in memoria abbia valore `0x7`. Il risultato della somma
`0x40 + 0x8 + 0x7 = 0x4F`
verrà salvato nel registro `rd`.

1) Mostrare le **modifiche all'architettura** della CPU MIPS, avendo cura di aggiungere eventuali altri componenti necessari a realizzare l'istruzione. A tal fine, si può alterare la stampa del diagramma architetturale oppure ridisegnare i componenti interessati dalla modifica, avendo cura di indicare i fili di collegamento e tutti i segnali entranti ed uscenti. Indicare inoltre sul diagramma i **segnali di controllo** che la CU genera *per realizzare l'istruzione*.

2) Indicare il contenuto in bit della word che esprime l'istruzione
`sum_w_mem $a0, $s1, $t2, 0x1D`
compilando la tabella sottostante. Assumere che lo *OpCode* di `sum_w_mem` sia `0x33`. Nel campo `funct`, porre il sestetto di bit che indichi l'addizione.

3) Supponendo che l'accesso alle **memorie** impieghi **180 ns**, l'accesso ai **registri 60 ns**, le operazioni dell'**ALU e** dei **sommatori 120 ns**, e che gli altri ritardi di propagazione dei segnali siano trascurabili, indicare la durata totale del **ciclo di clock** tale che anche l'esecuzione della nuova istruzione sia permessa *spiegando i calcoli effettuati*.


Instr [25-0]
Shift left 2
28
32
4
PC + 4 [31-28]
32
Add
4
Add
Shift left 2
0
M
u
x
1
1
M
u
x
0
Jump
Branch
Control
RegDst
Jump
Branch
MemWrite
MemToReg
MemRead
ALUSrc
ALUop
RegWrite
Instr [31-26]
RegWrite
PC
Read address
Instruction
Instruction memory
Instr [25-21]
Instr [20-16]
Instr [15-11]
Instr [15-0]
Instr [5-0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
RegDst
ALUSrc
Zero
ALU
MemRead
MemToReg
Read data
Address
Write data
Data memory
MemWrite
Sign-extend
16
32
ALU control
ALUOp
2
4

Esercizio 3 (2020-09-11)

Considerare l'architettura MIPS a ciclo singolo nella figura in alto (ed in allegato). Si sospetta che la CPU abbia un **guasto**. In particolare, si ha il dubbio che la Control Unit sia malfunzionante e che pertanto produca il segnale di controllo **RegWrite** attivo se e solo se **non** è attivo il segnale **Jump**: RegWrite ≔ *not*(Jump).
Si assume che:

– MemToReg = 1 solo per le istruzioni di tipo load, altrimenti valga 0 (ossia, mai "don't care", X)
– RegDst = 1 solo per le istruzioni di tipo R, altrimenti valga 0 (ossia, mai "don't care" X)
– MemRead = 1 solo per le istruzioni di tipo load, altrimenti valga 0 (ossia, mai "don't care" X)
– Branch = 1 solo per le istruzioni di salto condizionato, altrimenti valga 0 (ossia, mai "don't care" X)

1) Completare la tabella sottostante ipotizzando che ci sia il guasto. Per evidenziare i segnali errati a causa del guasto, aggiungere un punto esclamativo accanto: es., **1!** laddove il segnale corretto (ossia, senza guasto) sia 0, e **0!** laddove il segnale corretto sia 1.

	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	Jump	ALUOp1	ALUOp0
Tipo R									**1**	**0**
lw									**0**	**0**
sw									**0**	**0**
beq									**0**	**1**
j									**X**	**X**

2) Realizzare un programma che scriva in **$s0** il valore **0xFFFFFFFF se il guasto occorre**. **Altrimenti**, $s0 deve contenere il valore **0**.
Spiegare con commenti il malfunzionamento atteso.

Esercizio 4 (2021-01-29)

Si consideri l'architettura MIPS con pipeline mostrata nella figura in basso (ed in allegato). Il programma qui di seguito (fornito sia in formato testo che come immagine) effettua il conteggio degli elementi pari (in `$v0`) e dispari (in `$v1`) presenti nell'array `M` di lunghezza `N`.

```
1  .globl main
2
3  .data
4  M: .word 0, 151, 4885, 200, 19, 87 # Array
5  N: .byte 6                          # Lungh.
6  # Ris.: $v0 = 2 (el. pari); $v1 = 4 (dispari)
7  .text
8  main: la   $a0, M          # Indice base array
9        lb   $a1, N          # Lunghezza array
10       sll  $t7, $a1, 2     # $t7: last index…
11       add  $t7, $t7, $a0   # … of the array
12       sub  $v0, $t7, $t7   # Init. conto pari
13       add  $v1, $v0, $v0   # Init. conto dispari
14 cycl: bge  $a0, $t7, exit # Esci se $a0 >= $t7
15       lw   $s0, ($a0)      # $s0: elem. array
16       sll  $s0, $s0, 31    # $s0 << 31 (LSB)
17       beq  $s0, $0, odd    # Se $s0 non è pari…
18       addi $v1, $v1, 1     # … incrementa $v1…
19       j    endC            # … altrimenti…
20 odd:  addi $v0, $v0, 1     # … incrementa $v0
21 endC: addi $a0, $a0, 4     # Avanza nell'array
22       j cycl               # Ricomincia
23 exit: add $a0, $v0, $zero # Carica $v0
24       addi $v0, $zero, 1   # Imposta st. int.
25       syscall              # Stampa $v0
26       addi $v0, $zero, 11 # Imposta st. char.
27       addi $a0, $0, 0x0A   # Carica linefeed
28       syscall              # Stampa linefeed
29       addi $v0, $zero, 1   # Imposta st. int.
30       add $a0, $v1, $zero # Carica $v1
31       syscall              # Stampa $v1
32       addi $v0, $zero, 10 # Esci
33       syscall
```

```
.globl main

.data
M: .word 0, 151, 4885, 200, 19, 87
N: .byte 6

.text
main: la   $a0, M
      lb   $a1, N
      sll  $t7, $a1, 2
      add  $t7, $t7, $a0
      sub  $v0, $t7, $t7
      add  $v1, $v0, $v0
cycl: bge  $a0, $t7, exit
      lw   $s0, ($a0)
      sll  $s0, $s0, 31
      beq  $s0, $0, odd
      addi $v1, $v1, 1
      j    endC
odd:  addi $v0, $v0, 1
endC: addi $a0, $a0, 4
      j cycl
exit: add $a0, $v0, $zero
      addi $v0, $zero, 1
      syscall
      addi $v0, $zero, 11
      addi $a0, $0, 0x0A
      syscall
      addi $v0, $zero, 1
      add $a0, $v1, $zero
      syscall
      addi $v0, $zero, 10
      syscall
```

Si supponga che *tutte le istruzioni impiegate fanno parte del set supportato dalla CPU in figura*, ossia non si fa uso di alcuna pseudoistruzione.

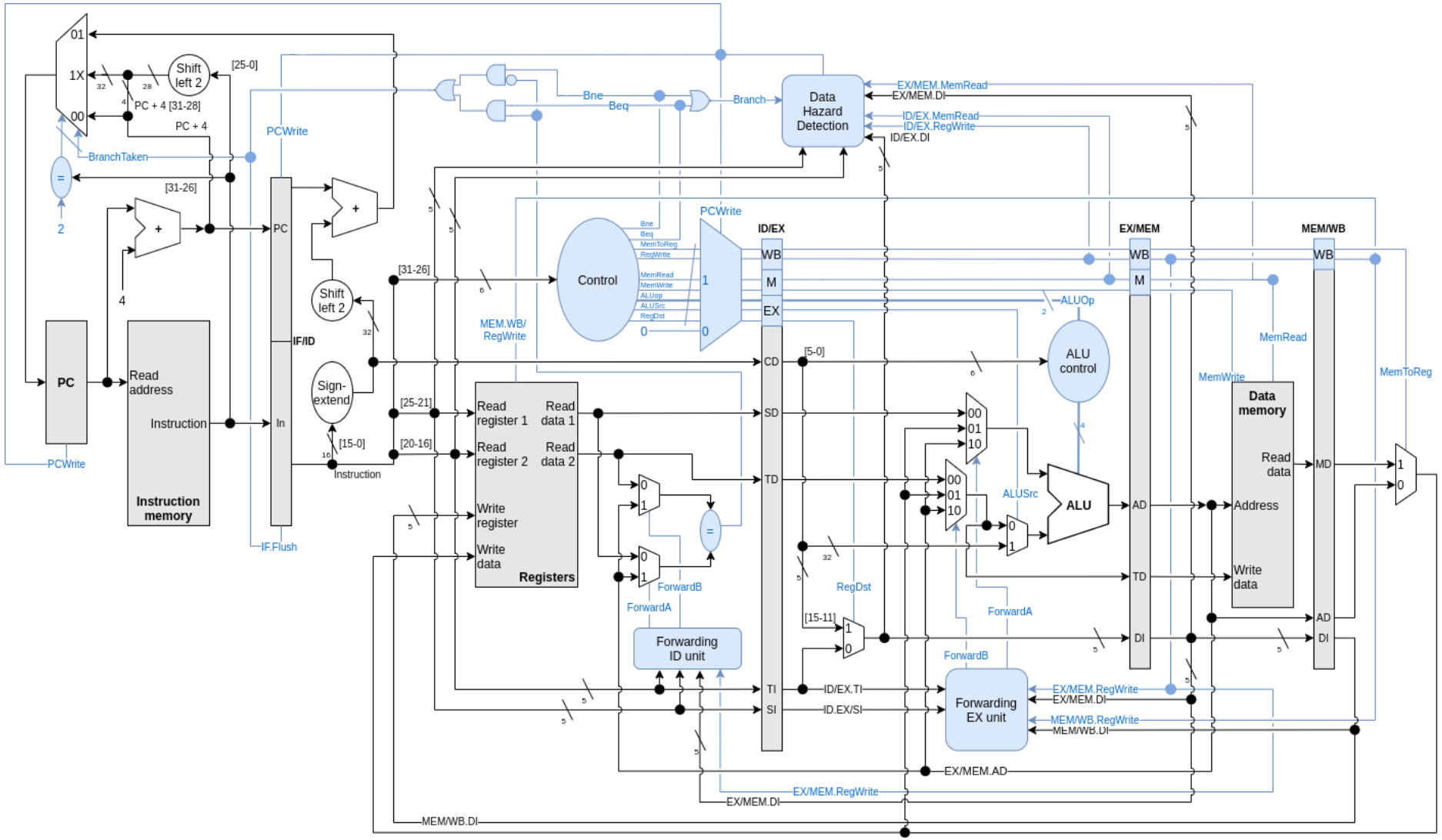
Si indichino (ignorando hazard che possano concernere la syscall):
1) le istruzioni tra le quali sono presenti **data hazard** – indicare i numeri di istruzione N ed M (visibili alla sinistra delle linee di codice in figura) ed il registro coinvolto $xY (nel formato N / M / $xY);

2) le istruzioni tra le quali sono presenti **control hazard** – indicare i numeri di istruzione N ed M (nel formato N / M);

3) quanti **cicli di clock** sono necessari ad eseguire il programma tramite **forwarding**, spiegando il calcolo effettuato;

4) quanti **cicli di clock** sarebbero necessari ad eseguire il programma **senza forwarding**, spiegando il calcolo effettuato;

5) quali sono, per ognuna delle cinque fasi, le **istruzioni** (o le bolle) in pipeline durante il **14° ciclo di clock** (con **forwarding**);
   IF:
   ID:
   EX:
   MEM:
   WB:


01
1X
00
Shift left 2
[25-0]
PC + 4 [31-28]
PC + 4
BranchTaken
=
2
[31-26]
+
4
PC
Read address
Instruction
Instruction memory
PCWrite
PCWrite
PC
IF/ID
In
IF.Flush
+
Shift left 2
Sign-extend
[15-0]
Instruction
[31-26]
[25-21]
[20-16]
Bne
Beq
Branch
Data Hazard Detection
EX/MEM.MemRead
EX/MEM.DI
ID/EX.MemRead
ID/EX.RegWrite
ID/EX.DI
Control
Bne
Beq
MemToReg
RegWrite
MemRead
MemWrite
ALUop
ALUSrc
RegDst
PCWrite
MEM.WB/ RegWrite
ID/EX
WB
M
EX
CD
SD
TD
TI
SI
EX/MEM
MEM/WB
ALUOp
ALU control
[5-0]
MemRead
MemWrite
MemToReg
Data memory
Read data
Address
Write data
ALU
ALUSrc
RegDst
[15-11]
ForwardA
ForwardB
Forwarding ID unit
Forwarding EX unit
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ID/EX.TI
ID.EX/SI
EX/MEM.RegWrite
EX/MEM.DI
MEM/WB.RegWrite
MEM/WB.DI
EX/MEM.AD
EX/MEM.RegWrite
EX/MEM.DI
MEM/WB.DI