Compito di Architettura dei Calcolatori - A.A. 2011-12
Prova di esame del 20 luglio 2012

Istruzioni: Spiegare chiaramente TUTTE le assunzioni che vengono effettuate per chiarire eventuali punti che si ritengono ambigui o non specificati.

1) [10 punti]
Illustrare i principi generali ed il flusso di controllo per la gestione di una "pipeline" per ottimizzare le prestazioni della CPU, discutendo sia il caso di salto incondizionato che di salto condizionato.

SVOLGIMENTO:

Si veda il cap.12 del libro di testo e dei lucidi presentati a lezione, in particolare i lucidi da 38 a 48 (compresi).

2) [10 punti]
Siano date due serie di N valori positivi in notazione *non complementata* a(i) e c(i) contenute in N celle di memoria consecutive a partire da quelle di indirizzo A e C. Scrivere un programma nel linguaggio Assembly di Virtual CPU che scrive nella cella r(i) il valore assoluto della differenza tra a(i) e c(i). I valori r(i) sono scritti in celle di memoria consecutive a partire da quella di indirizzo R.

I valori A, C, R, N sono contenuti nelle celle di memoria di indirizzo rispettivamente da 1 a 4. Assumere che i valori A, C, R, N sono tali da non creare mai problemi di sovrapposizione tra le celle di memoria

Commentare con adeguato dettaglio la logica seguita e le istruzioni usate

SVOLGIMENTO:

Si assume N>0. Ricordiamo che il linguaggio Assembly di Virtual CPU non possiede un'istruzione per il calcolo del valore assoluto. Dobbiamo allora distinguere i due casi:

- a(i) < c(i), nel qual caso il valore assoluto della differenza è dato da c(i)-a(i)
- a(i) ≥ c(i), nel qual caso il valore assoluto della differenza è dato da a(i)-c(i)

```
0    JMP 5         ; salta alla cella d'inizio del programma
; I DATI del programma, memorizzati in celle che sono usate come variabili di ingresso al programma
1                  ; contiene l'indirizzo del valore di a(i), cioè l'indirizzo di a(1) all'inizio
2                  ; contiene l'indirizzo del valore di c(i), cioè l'indirizzo di c(1) all'inizio
3                  ; contiene l'indirizzo del valore di r(i), cioè l'indirizzo di r(1) all'inizio
4                  ; contiene il numero di valori da confrontare
; IL PROGRAMMA
5    LOAD @@1      , si carica nell'accumulatore a(i)
6    SUB @@2       ; gli si sottrare c(i)
7    JL 16         ; se a(i)<c(i) si gestisce questo caso
8    STORE @@3     ; qui l'accumulatore contiene il valore assoluto corretto e lo si scrive in r(i)
9    DEC @4        ; si decrementa il numero di valori da confrontare
10   JZ 15         ; se non ce ne sono più si salta all'arresto altrimenti si prosegue
11   INC @1        , si incrementa l'indirizzo di a(i) per il passo successivo
12   INC @2        ; si incrementa l'indirizzo di c(i) per il passo successivo
13   INC @3        ; si incrementa l'indirizzo di r(i) per il passo successivo
14   JMP 5         ; si salta a ricominciare il passo successivo
15   HLT
; caso a(i)<(c(i)
16   LOAD @@2      ; si carica c(i) nell'accumulatore
17   SUB @@1       ; gli si sottrae a(i) calcolando il valore assoluto corretto per questo caso
18   JMP 8         ; si continua con la memorizzazione del valore assoluto
```

NB-1: Nel simulatore ENIAC di Virtual CPU l'istruzione JB, che fornisce il risultato del test di "minore" per valori rappresentati in notazione *non complementata*, non è corretta. Per questo motivo nella soluzione sopra esposta è stata usata JL che fornisce il risultato del test per valori rappresentati in notazione complementata. D'altro canto, poiché in Virtual CPU i valori sono rappresentati usando 24 bit, le due rappresentazioni sono equivalenti per valori positivi $\leq +2^{24-1}-1$. La soluzione sopra esposta (che può essere testata sul simulatore ENIAC) è pertanto valida in queste ipotesi. In sede di esame era ovviamente corretto usare JB.

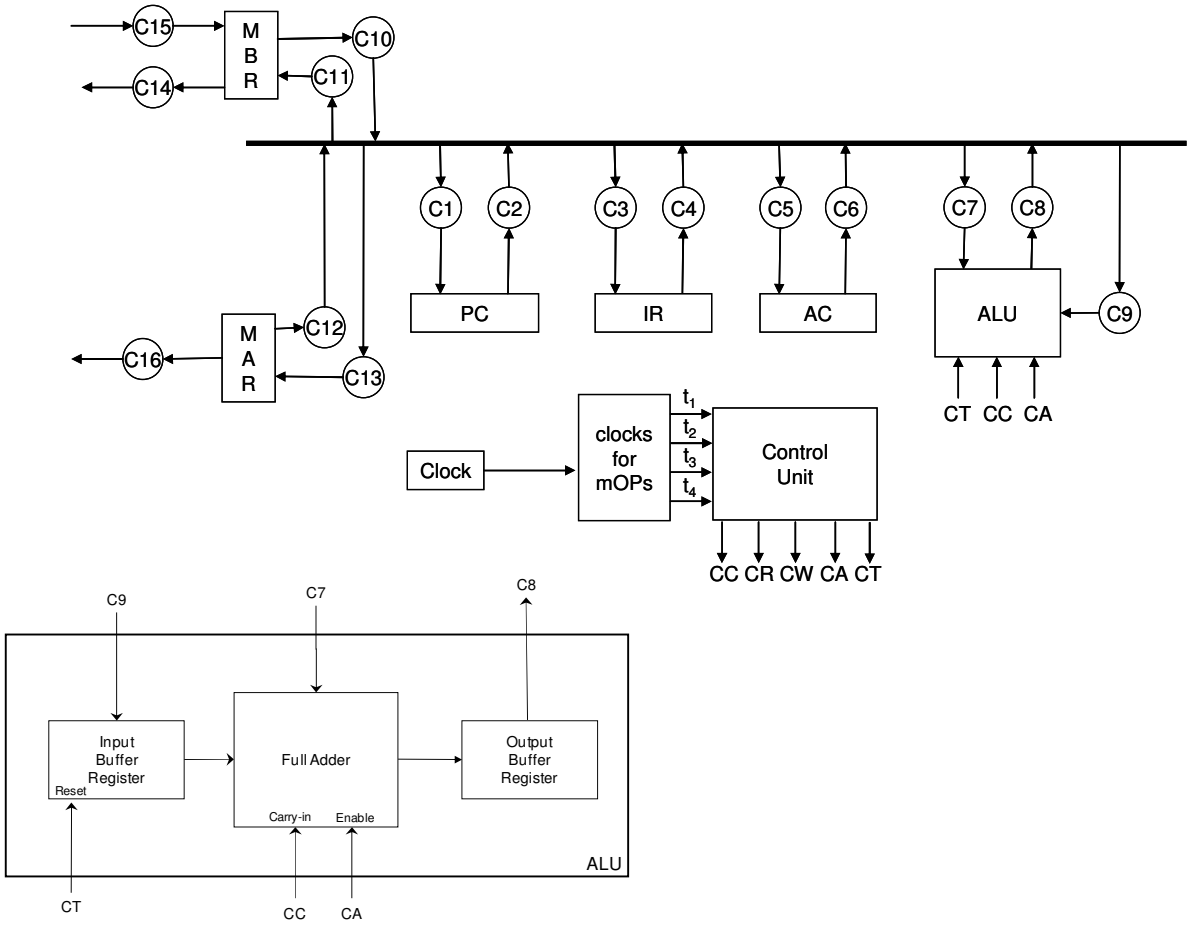
NB-2: In notazione non complementata il flag SI=1 (che indica che l'ultima operazione ha prodotto un risultato con MSB=1) non implica che il risultato dell'ultima operazione sia negativo. E' quindi scorretto usare il test JS o il

test JNS, così come è scorretto, quando si usano valori in notazione non complementata, usare qualunque istruzione che utilizza valori negativi (p.es. MUL -1)

NB-3: Anche se l'Assembly di Virtual CPU avesse un'istruzione per il calcolo del valore assoluto, essa darebbe risultati corretti per tutti i valori in ingresso solo quando applicata a valori rappresentati in notazione *complementata*, così come l'istruzione NEG fornisce risultati corretti per tutti i valori in ingresso solo nello stesso caso.

3) [10 punti]
Dato lo schema qua sotto della semplicissima CPU (VS0) – che si ricorda ha un formato istruzioni a 8 bit – scrivere e spiegare i microprogrammi necessari per l'esecuzione di tutte le fasi di ognuna delle seguenti istruzioni:

- M_ADD, caratterizzata dal bit b_5=0, che effettua l'addizione tra il valore presente in AC ed il valore contenuto nella successiva istruzione
- S_ADD N, caratterizzata dal bit b_5=1, che effettua l'addizione tra il valore presente in AC ed il valore N memorizzato nei bit b_4-b_0 dell'istruzione



SVOLGIMENTO:

Osserviamo preliminarmente che per la corretta esecuzione di queste due istruzioni l'unità di controllo deve essere in grado di distinguerle, così come deve essere in grado di distinguere tra loro tutte le istruzioni.

E' pertanto in primo luogo necessario che i bit b_7-b_6 di IR siano in ingresso all'unità di controllo (elemento circuitale che è erroneamente assente nello schema assegnato) ed è poi necessario che anche il bit b_5 sia in ingresso all'unità di controllo. Lo schema circuitale va quindi completato aggiungendo una connessione tra IR e la "Control Unit", specificando che si riferisce ai bit b_7-b_5 di IR.

Per la fase di esecuzione di S_ADD N è poi necessaria una nuova "porta di controllo" (Control Gate), denominata C17, tra IR e il bus, che consenta di far arrivare alla ALU soltanto i bit b_4-b_0 di IR che contengono il valore da sommare, altrimenti l'uso di C4 farebbe arrivare alla ALU anche il bit b_5 di IR.

Il micro-programma per la fase di *fetch* rimane sempre lo stesso che viene usato per tutte le istruzioni di VS0 ed è il seguente:

t1	MAR ← PC	C2, C13, CT	// il buffer interno della ALU va resettato con CT
t2	MBR ← memory	C16, C15, CR	
	PC + 1	C2, C7, CA, CC	// CC viene attivato per sommare +1
t3	PC ← ALU	C8, C1	
t4	IR ← MBR	C10, C3	

Per l'istruzione S_ADD N questo è sufficiente e può essere seguito dal micro-programma per la fase di *execute* di S_ADD N sotto descritto:

t1	ALU ← AC	C6, C9	// il 1° operando deve entrare su input con buffer
t2	AC + $IR_{b4\text{-}b0}$	C17, C7, CA	// con C17 solo i bit b_4-b_0 entrano nell'ALU
t3	AC ← ALU	C8, C5	

Per l'istruzione M_ADD è invece necessaria un'ulteriore fase di *fetch* dell'operando, effettuata dal seguente: micro-programma:

t1	MAR ← PC	C2, C13, CT	// il buffer interno della ALU va resettato con CT
t2	MBR ← memory	C16, C15, CR	
	PC + 1	C2, C7, CA, CC	// PC deve essere incrementato di nuovo per
t3	PC ← ALU	C8, C1	// il *fetch* della successiva istruzione

Dopo la fase di *fetch* dell'operando si prosegue con l'esecuzione del micro-programma per la fase di *execute* di M_ADD sotto descritto:

t1	ALU ← AC	C6, C9
t2	AC + MBR	C10, C7, CA
t3	AC ← ALU	C8, C5