Fabrizio Angiulli

MACCHINA MULTICICLO

1. ARCHITETTURA

Di seguito si riportano le caratteristiche dell'architettura della macchina multiciclo.

Modello di memoria
Una RAM da 64k locazioni (indirizzi a 16 bit) a 16 bit.

Registri
32 registri ad uso generale (GPR – General Purpose Register) a 16 bit.
1 registro flag Z ad 1 bit (vale 1 se il risultato dell'ALU è pari a zero e 0 altrimenti)

Tipi di dati
Numeri interi a 16 bit.

Istruzioni

Istruzioni: Modello di esecuzione registro-memoria
Nelle istruzioni di trasferimento ed elaborazione è presente come primo operando un registro che funge da sorgente e/o destinazione, mentre il secondo operando può essere un registro o una locazione di memoria; queste istruzioni rispettano la seguente sintassi: <*OP*> <*operando1*>, <*operando2*>.

Istruzioni: Modalità di indirizzamento
Sono disponibili quattro modalità di *indirizzamento*, di seguito specificate con riferimento all'istruzione LD (load) di caricamento dalla memoria:

i1) LD Ri, Rj *a registro* (in RTL: Rj → Ri);
i2) LD Ri, #X *immediato* (X è un intero a 16 bit; in RTL: X → Ri);
i3) LD Ri, X *diretto* (X è un indirizzo a 16 bit; in RTL: M[X] → Ri);
i4) LD Ri, X(Rj) *indicizzato* (X è un indirizzo a 16 bit; in RTL: M[X+Rj] → Ri).

Notiamo che ponendo X=0, è possibile ottenere come caso particolare dell'indirizzamento indicizzato una quinta modalità, ovvero l'indirizzamento indiretto a registro:

i4') LD Ri, 0(Rj) *indiretto a registro* (in RTL: M[0+Rj] → Ri, ovvero M[Rj] → Ri).

Instruzioni: Repertorio

Sono disponibili le seguenti istruzioni.

Trasferimento da e verso la memoria:

LD	load	tutti gli indirizzamenti
ST	store	solo indirizzamento diretto e indicizzato

Aritmetiche e confronto:

ADD	addition	tutti gli indirizzamenti
SUB	subtraction	tutti gli indirizzamenti
CMP	compare	tutti gli indirizzamenti (calcola op1 – op2 e aggiorna flag senza modificare i registri)

Salti condizionati ed incondizionati:

JP X	jump	nessuno (assumiamo diretto), salto incondizionato
JE X (JZ X)	jump if equal	nessuno (assumiamo diretto), salta se zero
JNE X (JNZ X)	jump if not equal	nessuno (assumiamo diretto), salta se non zero

Istruzioni: Formato

Per la codifica delle istruzioni in memoria scegliamo un formato con le seguenti caratteristiche:

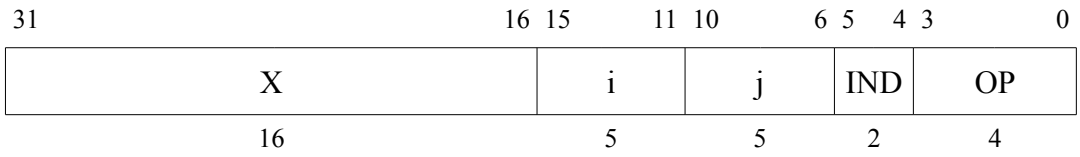
- ***lunghezza fissa***: 32 bit (in memoria occupa due locazioni consecutive);
- ***formato unico*** per tutte le categorie di istruzioni; quindi ***campi*** fissi: occupano sempre la stessa posizione ed hanno sempre la stessa lunghezza;
- le componenti ***operazione***, ***tipo di dato*** e ***indirizzamento*** sono **indipendenti** (proprietà nota come **ortogonalità**): quindi campi separati per ognuno di questi elementi.

L'ortogonalità permette di semplificare notevolmente la logica di controllo, perchè parte dei segnali di comando possono essere prelevati direttamente dall'IR senza la necessità di inviare la corrispondente informazione all'U.C. (alcuni campi dell'IR sono utilizzati per comandare una o più componenti dell'U.O. mediante collegamenti diretti, ovvero collegamenti che non passano dall'U.C.).

L'alternativa diametralmente opposta al formato ortogonale consiste nell'enumerare tutte le possibili combinazioni di operazioni, registri e indirizzamenti ed assegnare ad ogni combinazione un codice operativo univoco. Sebbene praticabile, questa soluzione ha lo svantaggio di far sì che la decodifica dell'istruzione sia completamente a carico dell'U.C. (con conseguente complicazione della relativa logica ed aumento delle sue dimensioni).

Si noti che in questo formato non compare il tipo di dato (es. Floating-point o Intero, per attivare rispettivamente l'ALU Floating-point o Intera) perchè è presente un solo tipo di dato (intero a 16 bit).

L'Instruction Register è a 32 bit ed organizzato nei seguenti ***campi***:



In particolare, il campo IND identifica il tipo di indirizzamento utilizzato dall'istruzione e può assumere i seguenti valori:

IND	significato
00	*a registro*
01	*immediato*
10	*diretto*
11	*indicizzato*

Il campo OP identifica il tipo di istruzione e può assumere i seguenti valori:

OP	significato
0001	LD
0010	ST
0011	ADD
0100	SUB
0101	CMP
0110	JP
0111	JE/JZ
1000	JNE/JNZ

I campi IND e OP, che indichiamo complessivamente come campo codice operativo COP = IND:OP (il carattere ':' è utilizzato per evidenziare la separazione in campi della sequenza di bit), costituiscono i (sei) segnali istruzione I da inviare all'U.C.
Di seguito si riportano tutti i codici operativi assegnati:

COP	Istruzione
00:0001	LD Ri, Rj
01:0001	LD Ri, #X
10:0001	LD Ri, X
11:0001	LD Ri, X(Rj)
10:0010	ST Ri, X
11:0010	ST Ri, X(Rj)
00:0011	ADD Ri, Rj
01:0011	ADD Ri, #X
10:0011	ADD Ri, X
11:0011	ADD Ri, X(Rj)
00:0100	SUB Ri, Rj
01:0100	SUB Ri, #X
10:0100	SUB Ri, X
11:0100	SUB Ri, X(Rj)
00:0101	CMP Ri, Rj
01:0101	CMP Ri, #X
10:0101	CMP Ri, X
11:0101	CMP Ri, X(Rj)
10:0110	JP X
10:0111	JE X
10:1000	JNE X

Tutti i COP non specificati sono non assegnati e sono disponibili per estensioni future del repertorio di istruzioni.

Un'istruzione occupa due locazioni consecutive di memoria. Assumiamo che la parte meno significativa (bit 0-15) dell'IR sia posta nella locazione di indirizzo più basso.

Esempio di rappresentazione in memoria di un programma assembly della macchina

Si consideri il seguente frammento di programma ad alto livello, che calcola la somma degli elementi di un array V composto da 100 elementi:

```
int s = 0;
for (int i = 0; i < 100, i++)
        s += V[i];
```

Si assuma che:

- la traduzione in linguaggio assembly del precedente frammento di programma sia posta in memoria a partire dalla locazione di indirizzo 2000;
- l'array V è posto in memoria a partire dalla locazione di indirizzo 19999;
- le variabili *s* ed *i* siano memorizzate nei registri R0 ed R1.

Di seguito si riporta il corrispondente frammento in linguaggio assembly:

```
        LD    R0, #0      ; s = 0
        LD    R1, #0      ; i = 0
LOOP:   ADD   R0, V(R1)   ; s += V[i]
        ADD   R1, #1      ; i++
        CMP   R1, #100    ; i == 100 ?
        JNE   LOOP
```

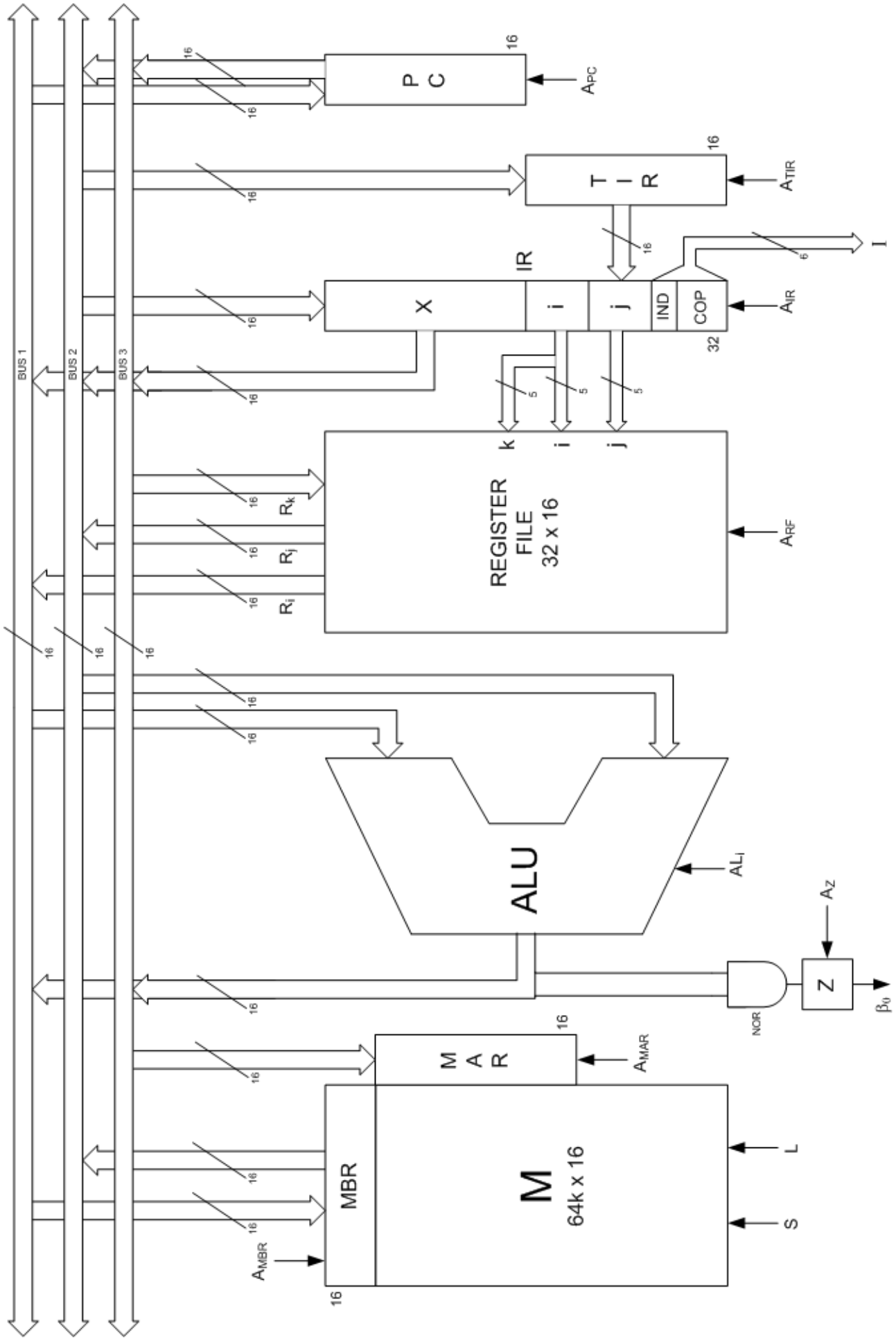
e la relativa rappresentazione in memoria (il carattere 'x' indica un bit il cui valore non è significativo; si può assumere uguale a 0):

Indirizzo	Contenuto locazione	Istruzione assembly
...	...	
2000:	00000:xxxxx:01:0001	LD R0, #0
2001:	0	
2002:	00001:xxxxx:01:0001	LD R1, #0
2003:	0	
2004:	00000:00001:11:0011	ADD R0, V(R1)
2005:	19999_{10}	
2006:	00001:xxxxx:01:0011	ADD R1, #1
2007:	1	
2008:	00001:xxxxx:01:0101	CMP R1, #100

2009:	100_{10}	
2010:	xxxxx:xxxxx:10:1010	JNE LOOP
2011:	2004_{10}	
...	...	
19999:	V[0]	
20000:	V[1]	
	...	
20098:	V[99]	

Si consideri il seguente alternativo frammento di programma. Che funzione viene svolta? Qual è la corrispondente rappresentazione in memoria?

```
        LD    R0, #0        ; s = 0
        LD    R1, #100      ; contatore
        LD    R2, #V        ; indirizzo partenza array
LOOP2:  ADD   R0, 0(R2)
        ADD   R2, #1
        SUB   R1, #1
        JE    LOOP2
```


BUS 1
BUS 2
BUS 3
P C
A_PC
T I R
A_TIR
IR
X
i
j
IND
COP
A_IR
32
I
k
i
j
REGISTER FILE 32 x 16
A_RF
R_k
R_j
R_i
ALU
AL_I
A_Z
Z
NOR
β_U
M A R
A_MAR
MBR
M 64k x 16
A_MBR
L
S
16
5
6

2. PROGETTO DELL'UNITA' OPERATIVA

Componenti principali dell'U.O.:

- 3 BUS: la disponibilità di più bus consente di aumentare il *"parallelismo" del percorso dati* (*data path*); infatti, due bus possono essere utilizzati per portare in ingresso all'ALU due operandi ed il terzo bus per inviare il risultato dell'ALU ad un registro, il tutto in un solo ciclo di clock;
- *Register File – RF* (o *Banco dei Registri*): blocco funzionale contenente i registri ad uso generale; permette di leggere il contenuto di due registri e di modificare il contenuto di un altro registro nello stesso ciclo di clock; nello schema di riferimento, il RF invia i registri da leggere sui bus 1 e 2 e preleva il registro da scrivere dal bus 3;
- ALU intera: utilizzata per effettuare le elaborazioni; nello schema di riferimento, preleva i due ingresso dai bus 1 e 2 ed invia il risultato al bus 3.

In generale ogni componente può essere collegato a tutti i bus. Si limiterà il numero di tali collegamenti cercando di rispettare il flusso dei dati programmato per il data path (*bus 1 e 2 "sorgenti"* e *bus 3 "destinazione*) ed aggiungendo collegamenti ogni qualvolta si renda necessario.

L'IR è a 32 bit e suddiviso nei campi illustrati in precedenza. In particolare i campi i e j, il cui scopo è quello di specificare i registri Ri e Rj coinvolti nell'istruzione, sono utilizzati per comandare direttamente il RF sfruttando il formato istruzione ortogonale.
Poichè la macchina adotta il modello di esecuzione registro-memoria, il campo i dell'IR identifica sia un registro sorgente che un registro destinazione e quindi il suo contenuto viene inviato agli ingressi i e k del RF (che individuano, rispettivamente, uno dei due registri da leggere del RF e l'unico registro da scrivere del RF), mentre il campo j dell'IR viene inviato in ingresso al campo j del RF (che individua l'altro dei due registri da leggere del RF).

Il registro TIR a 16 bit viene utilizzato per memorizzare temporaneamente la parte meno significativa del registro IR durante la fase di fetch. Si ricorda che la sovrascrittura del campo COP determina la fine della fase di fetch ed il passaggio all'esecuzione vera e propria dell'istruzione (fase di execute). Si tratta quindi dell'ultimo passaggio da eseguire durante la fase di fetch.
Si noti che si sarebbe potuto scegliere di rappresentare l'istruzione in memoria ponendo nella locazione di indirizzo più basso il campo X, eliminando così la necessità del registro TIR. Sebbene questa soluzione sia praticabile per il formato istruzione corrente, non è adatta per formati istruzione a lunghezza variabile, in cui è necessario ispezionare innanzitutto il campo COP al fine di determinare la lunghezza effettiva dell'istruzione corrente.

La scelta del ***periodo di clock*** è determinata dalla microistruzione più lunga, ovvero quella che tiene impegnato il percorso combinatorio con il maggior ritardo complessivo. Nel caso specifico occorre tener conto del ritardo dell'ALU, del bus e del Register File (somma dei ritardi dei sottosistemi di lettura e scrittura), nonchè del ritardo dell'U.C.
Si assume che il tempo di accesso alla memoria sia inferiore al ritardo complessivo, in modo che un'operazione di lettura/scrittura possa essere completata in un unico ciclo di clock (diversamente, occorre introdurre uno o più cicli di wait in corrispondenza degli accessi alla memoria, oppure aumentare la durata del periodo di clock).

CODICE RTL DELLE ISTRUZIONI E TRADUZIONE IN MICROISTRUZIONI

FETCH (COP = 00:0000)

PC → MAR, PC+1 → PC; <μ_1>
M[MAR] → MBR, PC → MAR, PC+1 → PC; <μ_2>
MBR → TIR, M[MAR] → MBR; <μ_3>
TIR → $IR_{0\text{-}15}$, MBR → X; <μ_4>

Anzichè progettare il PC come registro funzione a incremento, sfruttiamo la disponibilità di 3 bus per incrementare il PC mediante la ALU. Il registro X coincide con i bit 16-31 del campo IR ($X = IR_{16\text{-}31}$). Il registro TIR viene utilizzato per caricare il campo COP nell'ultimo micropasso della microsequenza associata alla FETCH.

<μ_1> W3_PC, R3_MAR, A_MAR, W2_PC, R2_ALU, ALU_INC2, W1_ALU, R1_PC, A_PC
<μ_2> L, A_MBR, W3_PC, R3_MAR, W2_PC, R2_ALU, ALU_INC2, W1_ALU, R1_PC, A_PC
<μ_3> W2_MBR, R2_TIR, A_TIR, L, A_MBR
<μ_4> W2_MBR, R2_X, A_IR

Istruzione LD Ri, Rj

Rj → Ri; <μ_5>

Il registro Rj viene inviato mediante il bus 2 alla ALU, che lo restituisce invariato in uscita sul bus 3 e viene quindi scritto nel RF. Occorre abilitare il RF in scrittura (A_RF asserito) per permettere la scrittura del registro Ri (ovvero Rk).

<μ_5> W2_RF, R2_ALU, ALU_EQ2, W3_ALU, R3_RF, A_RF

Istruzione LD Ri, #X

X → Ri; <μ_6>

Il contenuto del campo X dell'IR, che in questo caso rappresenta un numero intero (operando immediato), viene inviato al RF mediante il bus 3.

<μ_6> W3_X, R3_RF, A_RF

Istruzione LD Ri, X

X → MAR; <μ_7>
M[MAR] → MBR; <μ_8>
MBR → Ri; <μ_9>

Il contenuto della locazione di memoria di indirizzo X viene letto dalla RAM e scritto nel RF.

<μ_7> W3_X, R3_MAR, A_MAR
<μ_8> L, A_MBR
<μ_9> W2_MBR, R2_ALU, ALU_EQ2, W3_ALU, R3_RF, A_RF

Istruzione LD Ri, X(Rj)

X + Rj → MAR; <μ_{10}>
M[MAR] → MBR; <μ_8>
MBR → Ri; <μ_9>

Il contenuto della locazione di memoria di indirizzo X+Rj viene letto dalla RAM e scritto nel RF. L'indirizzo effettivo di memoria viene calcolato utilizzando la ALU: il campo X di IR viene letto dal bus 1, il registro Rj dal bus 2 e la loro somma inviata al MAR dalla ALU mediante il bus 3.

<μ_{10}> W1_X, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_MAR, A_MAR

Istruzione ST Ri, X

X → MAR, Ri → MBR; <μ_{11}>
MBR → M[MAR]; <μ_{12}>

Il contenuto del registro Ri viene scritto in memoria nella locazione di indirizzo X. Il MAR (campo X mediante bus 3) e l'MBR (registro Ri mediante bus 1) vengono caricati nello stesso ciclo di clock.

<μ_{11}> W1_RF, R1_MBR, A_MBR, W3_X, R3_MAR, A_MAR
<μ_{12}> S

Istruzione ST Ri, X(Rj)

X + Rj → MAR; <μ_{10}>
Ri → MBR; <μ_{13}>
MBR → M[MAR]; <μ_{12}>

Il contenuto del registro Ri viene scritto in memoria nella locazione di indirizzo X+Rj. Il MAR (campo X mediante bus 3) e l'MBR (registro Ri mediante bus 1) vengono caricati nello stesso ciclo di clock. Questa volta i 3 bus vengono impegnati dal calcolo dell'indirizzo effettivo (primo micropasso) e quindi il caricamento dell'MBR richiede un ulteriore ciclo di clock. Complessivamente la store indicizzata risulta più lenta di quella diretta.

<μ_{10}> W1_X, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_MAR, A_MAR
<μ_{13}> W1_RF, R1_MBR, A_MBR
<μ_{12}> S

Istruzione ADD Ri, Rj

Ri + Rj → Ri, NOR(Ri + Rj) → Z; <μ_{14}>

Calcola Ri + Rj e memorizza il risultato in Ri (ovvero Rk). Ri arriva all'ALU mediante il bus1, Rj mediante il bus 2 e la somma viene presentata al RF mediante il bus 3. Il flag Z viene modificato di conseguenza (comando A_Z asserito).

<μ_{14}> W1_RF, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_RF, A_RF, A_Z

Istruzione ADD Ri, #X

Ri + X → Ri, NOR(Ri + X) → Z; <μ_{15}>

Somma Ri (via bus 1) e campo X (via bus 2; operando immediato) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

<μ_{15}> W1_RF, R1_ALU, W2_X, R2_ALU, ALU_ADD, W3_ALU, R3_RF, A_RF, A_Z

Istruzione ADD Ri, X

X → MAR; <μ_7>
M[MAR] → MBR; <μ_8>
Ri + MBR → Ri, NOR(Ri + MBR) → Z; <μ_{16}>

Somma il contenuto del registro Ri (via bus 1) e della locazione di memoria di indirizzo X (provienente dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

<μ_7> W3_X, R3_MAR, A_MAR
<μ_8> L, A_MBR
<μ_{16}> W2_MBR, R2_ALU, W1_RF, R1_ALU, ALU_ADD, W3_ALU, R3_RF, A_RF, A_Z

Istruzione ADD Ri, X(Rj)

X + Rj → MAR; <μ_{10}>
M[MAR] → MBR; <μ_8>
Ri + MBR → Ri, NOR(Ri + MBR) → Z; <μ_{17}>

L'indirizzo effettivo (X+Rj) della locazione di memoria da addizionare viene calcolato mediante l'ALU. Somma il contenuto del registro Ri (via bus 1) e della locazione di memoria di indirizzo X+Rj (provienente dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

<μ_{10}> W1_X, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_MAR, A_MAR
<μ_8> L, A_MBR
<μ_{17}> W1_RF, R1_ALU, W2_MBR, R2_ALU, ALU_ADD, W3_ALU, R3_RF, A_RF, A_Z

Istruzione SUB Ri, Rj

Ri – Rj → Ri, NOR(Ri – Rj) → Z; $<\mu_{18}>$

Calcola Ri – Rj e memorizza il risultato in Ri (ovvero Rk). Ri arriva all'ALU mediante il bus1, Rj mediante il bus 2 e la somma viene presentata al RF mediante il bus 3. Il flag Z viene modificato di conseguenza (comando A_Z asserito).

$<\mu_{18}>$ W1_RF, R1_ALU, W2_RF, R2_ALU, ALU_SUB, W3_ALU, R3_RF, A_RF, A_Z

Istruzione SUB Ri, #X

Ri – X → Ri, NOR(Ri – X) → Z; $<\mu_{19}>$

Sottrae da Ri (via bus 1) il campo X (via bus 2; operando immediato) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

$<\mu_{19}>$ W1_RF, R1_ALU, W2_X, R2_ALU, ALU_SUB, W3_ALU, R3_RF, A_RF, A_Z

Istruzione SUB Ri, X

X → MAR; $<\mu_{7}>$
M[MAR] → MBR; $<\mu_{8}>$
Ri – MBR → Ri, NOR(Ri – MBR) → Z; $<\mu_{20}>$

Sottrae dal contenuto del registro Ri (via bus 1) il contenuto della locazione di memoria di indirizzo X (proveniente dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

$<\mu_{7}>$ W3_X, R3_MAR, A_MAR
$<\mu_{8}>$ L, A_MBR
$<\mu_{20}>$ W2_MBR, R2_ALU, W1_RF, R1_ALU, ALU_SUB, W3_ALU, R3_RF, A_RF, A_Z

Istruzione SUB Ri, X(Rj)

X + Rj → MAR; $<\mu_{10}>$
M[MAR] → MBR; $<\mu_{8}>$
Ri – MBR → Ri, NOR(Ri – MBR) → Z; $<\mu_{21}>$

L'indirizzo effettivo (X+Rj) della locazione di memoria da sottrarre viene calcolato mediante l'ALU. Sottrae dal contenuto del registro Ri (via bus 1) il contenuto della locazione di memoria di indirizzo X+Rj (proveniente dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

<μ_{10}> W1_X, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_MAR, A_MAR
<μ_8> L, A_MBR
<μ_{21}> W1_RF, R1_ALU, W2_MBR, R2_ALU, ALU_SUB, W3_ALU, R3_RF, A_RF, A_Z

Istruzione CMP Ri, Rj

NOR(Ri – Rj) → Z; <μ_{22}>

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Calcola Ri – Rj e non memorizza il risultato in Ri=Rk (comando A_Z non asserito). Ri arriva all'ALU mediante il bus1, Rj mediante il bus 2. Il flag Z viene modificato di conseguenza.

<μ_{22}> W1_RF, R1_ALU, W2_RF, R2_ALU, ALU_SUB, A_Z

Istruzione CMP Ri, #X

NOR(Ri – X) → Z; <μ_{23}>

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Sottrae da Ri (via bus 1) il campo X (via bus 2; operando immediato). Aggiorna il flag Z, ma non il RF.

<μ_{23}> W1_RF, R1_ALU, W2_X, R2_ALU, ALU_SUB, A_Z

Istruzione CMP Ri, X

X → MAR; <μ_7>
M[MAR] → MBR; <μ_8>
NOR(Ri – MBR) → Z; <μ_{24}>

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Sottrae dal contenuto del registro Ri (via bus 1) quello della locazione di memoria di indirizzo X (proviene dall'MBR via bus 2). Aggiorna il flag Z, ma non il RF.

<μ_7> W3_X, R3_MAR, A_MAR
<μ_8> L, A_MBR
<μ_{24}> W2_MBR, R2_ALU, W1_RF, R1_ALU, ALU_SUB, A_Z

Istruzione CMP Ri, X(Rj)

X + Rj → MAR; <μ_{10}>
M[MAR] → MBR; <μ_8>
NOR(Ri – MBR) → Z; <μ_{25}>

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. L'indirizzo effettivo (X+Rj) della locazione di memoria da sottrarre viene calcolato mediante l'ALU. Sottrae dal contenuto del registro Ri (via bus 1) quello della locazione di memoria di indirizzo X+Rj (provienente dall'MBR via bus 2). Aggiorna il flag Z, ma non il RF.

<μ_{10}> W1_X, R1_ALU, W2_RF, R2_ALU, ALU_ADD, W3_ALU, R3_MAR, A_MAR
<μ_{8}> L, A_MBR
<μ_{25}> W1_RF, R1_ALU, W2_MBR, R2_ALU, ALU_SUB, A_Z

Istruzione JP X

X → PC; <μ_{26}>

Carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

<μ_{26}> W1_X, R1_PC, A_PC

Istruzione JE X / JZ X

if Z = 1
then **X → PC;** <μ_{26}>
else **φ;** <μ_{0}>
fi

Se Z vale 1 carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

Istruzione JNE X / JNZ X

if Z = 0
then **X → PC;** <μ_{26}>
else **φ;** <μ_{0}>
fi

Se Z vale 0 carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

3. PROGETTO DELL'UNITA' DI CONTROLLO

Numero di segnali istruzione:
Campo COP = IND:OP dell'IR, per un totale di 6 bit.

Numero di segnali condizione:
Flag Z, per un totale di un 1 bit.

Numero di segnali di stato:
La più lunga microsequenza consta di 4 micropassi, per un totale di 2 bit di stato.

Numero di segnali di comando:
Elencare i segnali di comando e determinarne il numero.

Esercizio: progettare la parte di controllo microprogrammata della macchina a registri.

4. MODIFICA DEL FORMATO ISTRUZIONE

Modifichiamo ora il formato istruzione della macchina con i seguenti obiettivi:

1. Evitare spreco di memoria, ovvero che l'istruzione occupi più spazio di quello strettamente necessario;
2. Aggiungere esplicitamente l'indirizzamento indiretto a registro.

Partiamo dagli indirizzamenti. La macchina dove ora supportare le seguenti cinque modalità di indirizzamento:

i1) LD Ri, Rj *a registro*
i2) LD Ri, #X *immediato*
i3) LD Ri, X *diretto*
i4) LD Ri, X(Rj) *indicizzato*
i5) LD Ri, (Rj) *indiretto a registro*

Due bit non sono più sufficienti per distringuere gli indirizzamenti. Per mantenere l'ortogonalità del formato istruzione si rende necessario un terzo bit. Dove lo recuperiamo?

Proviamo a riorganizzare i campi dell'IR. Notiamo innanzitutto che i campi i, j e X non sono richiesti contemporaneamente da tutti gli indirizzamenti. In particolare:

- Il campo X non è richiesto dagli indirizzamenti *i1* e *i5*, che invece richiedono i campi i e j. Le istruzioni che fanno uso di questi indirizzamenti potrebbero occupare una sola locazione di memora anzichè due locazioni;
- Gli indirizzamenti *i2* e *i3* richiedono il campo X ma non il campo j;
- L'indirizzamento *i4* richiede tutti e tre i campi.

Riprogettiamo il campo IND a 2 bit:

IND	**significato**
00	*a registro*
10	*indiretto a registro*
01	*indicizzato*
11	*immediato*
11	*diretto*

Le prime due configurazioni del campo X sono relative ai due indirizzamenti che non usano il campo X. Esse hanno in comune il fatto di avere il primo bit posto a 0 (si assuma che il bit posto più a destra sia quello meno significativo), mentre le altre configurazioni hanno sempre il primo bit posto ad 1.
Le configurazione successiva (01) è relativa all'indirizzamento che usa tutti e tre i campi.
Rimane una configurazione (11) per codificare i due indirizzamenti che non usano il campo j. Possiamo in questo caso sfruttare tale configurazione come prefisso comune di una nuova configurazione a 3 bit, il cui terzo bit discrimina tra indirizzamento immediato e diretto. Questo terzo bit viene preso dal campo j, che con queste istruzioni non è utilizzato.
Arriviamo quindi al seguente nuovo campo IND a 3 bit:

IND	**significato**
x00	*a registro*
x10	*indiretto a registro*
x01	*indicizzato*
011	*immediato*
111	*diretto*

Tale campo occupa i bit 4, 5 e 6 dell'IR. Il terzo bit di IND (il bit 6 di IR) è condiviso con il campo j ed è significativo solo quando il campo j non viene utilizzato. Il campo COP sarà formato ora dai 7 bit meno significativi dell'IR. I segnali istruzione da inviare all'U.C. passano quindi da 6 a 7. Per alcune istruzioni il 7° bit istruzione non sarà significativo.

Il primo bit del campo IND ha anche un ulteriore significato: specifica la lunghezza dell'istruzione. Se vale 0 l'istruzione occupa 16 bit, se vale 1 invece occupa 32 bit. Chiameremo LEN questo bit (bit 4 dell'IR).

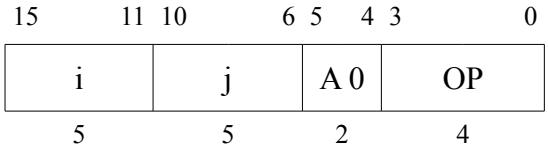
In conclusione, siamo passati da un formato istruzione di lunghezza fissa e formato unico ad un formato istruzione di ***lunghezza variabile*** e ***formato non unico***, ma variabile a seconda della categiora dell'istruzione o dell'indirizzamento.

Per esempio, nel caso specifico se il bit 4 (LEN) dell'istruzione vale 0, allora l'istruzione occupa 16 bit. Diversamente occupa 32 bit.
Inoltre, in base alla configurazione dei bit 4 e 5 dell'istruzione, i campi di cui si compone l'istruzione cambiano (se 00 o 10, sono presenti i campi i e j, ma non il campo X; se 01, sono presenti i campi i, j e X; se 11, sono presenti i campi i ed X, ma non il campo j).

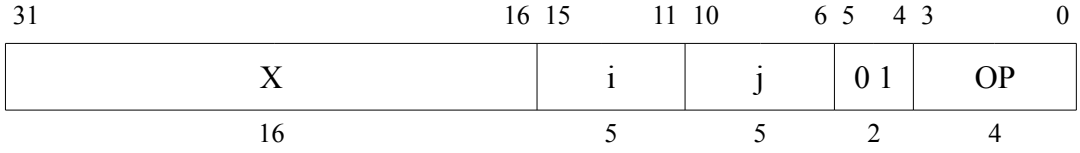
Il formato istruzione utilizza inoltre un ***codice operativo espandibile***, ovvero un campo COP di lunghezza variabile. Un codice operativo espandibile si basa sull'uso di *prefissi*, ovvero particolari configurazioni (in genere dei bit meno significativi del campo COP) da cui è possibile risalire alla lunghezza effettiva del campo COP.

Per esempio, nel caso specifico se i bit 4 e 5 dell'istruzione valgono entrambi 1, allora il campo COP occupa 7 bit. Diversamente, il campo COP occupa 6 bit.

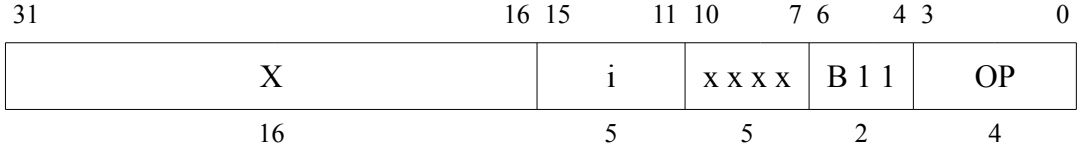
Di seguito sono riporati i diversi formati che l'istruzione può assumere:



<OP> in {LD, ST, ADD, SUB, CMP}:
A = 0: <OP> Ri, Rj (ST non ammessa)
A = 1: <OP> Ri, (Rj)



<OP> in {LD, ST, ADD, SUB, CMP}:
<OR> Ri, X(Rj)



<OP> in {LD, ST, ADD, SUB, CMP}:
B=0: <OP> Ri, #X (ST non ammessa)
B=1: <OP> Ri, X

<OP>=JUMP (OP=0110; in questo caso il campo i ed il bit B non sono significativi)
JUMP X

<OP>={JE,JNE} (usare OP unico per JE/JNE, es. OP=0111, e discriminare tra le due usando bit B; in questo caso il campo i non è significativo)
B=0: JNE X
B=1: JE X

NUOVA FASE FETCH

Modifichiamo la fase di fetch in modo da gestire le istruzioni di lunghezza variabile. Al fine di non sprecare cicli di clock durante il caricamento dell'istruzione, aggiungiamo un segnale condizione sul bit 4 dell'MBR, segnale che utilizzeremo per testare il valore del campo LEN quando leggiamo dalla memoria la prima locazione relativa alla prossima istruzione da eseguire.

PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR;
if $MBR_4 = 0$
then $MBR \rightarrow IR_{0\text{-}15}$;
else M[MAR] → MBR, MBR → TIR, PC + 1 → PC;
$MBR \rightarrow X$, $TIR \rightarrow IR_{0\text{-}15}$;
fi

La nuova fetch impiega tre cicli di clock per caricare un'istruzione che occupa 1 locazione di memoria (MBR_4 = LEN = 0) e quattro cicli di clock per caricare un'istruzione che occupa 2 locazioni di memoria (MBR_4 = LEN = 1).

Per eseguire correttamente la fetch occorre poter caricare $IR_{0\text{-}15}$ direttamente dall'MBR, aggiungendo un collegamento dal bus 2 verso $IR_{0\text{-}15}$ ed un multiplexer (pilotato da un nuovo segnale di comando) in cui confluiscono il nuovo collegamento e l'uscita del TIR.

Si noti che nella parte **else** della precedente microsequenza viene modificato il valore del segnale di controllo $\mathbf{MBR_4}$ da cui dipende l'esecuzione del relativo **if**. Nonostante il segnale condizione da cui dipende l'**if** non sia stabile per tutta la durata dell'**else**, in questo particolare caso è possibile comunque tradurre la sequenza in microistruzioni sfruttando il fatto che la parte **then** è composta da un solo micropasso, considerando la precedente microsequenza equivalente a quella di seguito illustrata:

PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR;
if $MBR_4 = 0$
  then **MBR → $IR_{0\text{-}15}$;** (qui si passa ad eseguire un'istruzione)
  else **M[MAR] → MBR, MBR → TIR, PC + 1 → PC;**
fi
MBR → X, TIR → $IR_{0\text{-}15}$;

Nel caso una tale trasformazione non fosse possibile è necessario stabilizzare il segnale condizione trasferendo l'MBR nel TIR. Ad esempio:

PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR;
MBR → TIR;
if $TIR_4 = 0$
  then **TIR → $IR_{0\text{-}15}$;**
  else **M[MAR] → MBR, PC + 1 → PC;**
    MBR → X, TIR → $IR_{0\text{-}15}$;
fi

oppure:

PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR;
MBR → TIR, M[MAR] → MBR; (anticipa la lettura della seconda locazione dell'istruzione; se non necessaria verrà ignorata)
if $TIR_4 = 0$
  then **TIR → $IR_{0\text{-}15}$;**
  else **PC + 1 → PC, MBR → X, TIR → $IR_{0\text{-}15}$;** (aggiungere collegamenti mancanti)
fi

ALTRI SCHEMI DI FETCH

La fetch che abbiamo appena visto presuppone che esistano uno o più bit nel COP dell'istruzione che permettono di risalire alla lunghezza dell'istruzione. In ogni caso, il COP nella sua interezza permette di determinare questa informazione e quindi lo schema di fetch precedente è sempre applicable.

Una soluzione semplice, nonchè diametralmente opposta alla precedente, consiste nel caricare durante la fetch solo la prima parte dell'istruzione (di lunghezza sufficiente a contenere il campo

COP) e poi lasciare a carico di ogni istruzione il corretto completamento della fase di fetch.
In quest'ultima ipotesti, si potrebbe arrivare ad un allungamento delle microsequenze associate ad alcune istruzioni, oppure, in altri casi, ad una riduzione del numero di cicli di clock complessivi a scapito di una qualche complicazione del microcodice dovuta alla commistione delle fasi di fetch ed execute (ad esempio, potrebbe non essere più necessario allocare nell'IR tutti i campi previsti nel formato istruzione perchè gestiti direttamente da ogni istruzione, l'istruzione dovrebbe farsi carico di incrementare il PC, od altro ancora).

Per esempio, l'istruzione *ADD Ri, X(Rj)* potrebbe essere associata alla seguente microsequenza:

Fetch:
PC → MAR, PC + 1 → PC;
M[MAR] → MBR;
MBR → $IR_{0\text{-}15}$;
ADD Ri, X(Rj):
PC → MAR;
M[MAR] → MBR, PC + 1 → PC;
MBR → X;
X + Rj → MAR;
M[MAR] → MBR;
Ri + MBR → Ri, NOR(Ri + MBR) → Z;

Oppure, ottimizzando:

Fetch:
PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR; (carica il MAR con l'indirizzo della locazione successiva)
M[MAR] → MBR, MBR → $IR_{0\text{-}15}$; (anticipa la lettura della seconda locazione dell'istruzione; se non necessaria verrà ignorata)
ADD Ri, X(Rj):
MBR + Rj → MAR; (l'indirizzo X è nell'MBR e non nel campo X dell'IR, campo che non viene più utilizzato; occorre collegare l'uscita dell'MBR al bus 1 in modo da poter effettaure la somma in un ciclo di clock)
M[MAR] → MBR, PC + 1 → PC; (incrementa il PC sfruttando il ciclo di accesso alla RAM)
Ri + MBR → Ri, NOR(Ri + MBR) → Z;

In questo caso, l'istruzione *ADD Ri, X(Rj)* richiede un ciclo di clock in meno rispetto al progetto discusso in precedenza (6 cicli invece di 7). Si noti che si sarebbe potuto ottenere lo stesso risultato anche con il precedente schema di fetch, assumendo che l'IR non contenga il campo X, ma piuttosto che l'eventuale operando X sia disponibile nell'MBR all'inizio dell'esecuzione dell'istruzione, come mostrato nel seguito:

Fetch:
PC → MAR, PC + 1 → PC;
M[MAR] → MBR, PC → MAR;
if MBR_4 = 0
then MBR → $IR_{0\text{-}15}$;
else M[MAR] → MBR, MBR → $IR_{0\text{-}15}$, PC + 1 → PC;

fi
ADD Ri, X(Rj):
MBR + Rj → MAR;
M[MAR] → MBR;
Ri + MBR → Ri, NOR(Ri + MBR) → Z;

Si tratta in ogni caso di una soluzione adatta per lo specifico formato e che comunque crea commistione tra le fasi di fetch ed execute.

Sebbene tutte queste soluzioni siano applicabili, è preferibile mantenere separata la fase di fetch da quella di execute vera e propria.

5. ULTERIORI MODIFICHE

1) Aggiungere all'U.O. il registro FLAG composto dai 4 bit Z, S, C e O:

Z: zero (uscita dell'ALU uguale a zero)
S: sign (segno dell'uscita dell'ALU)
C: carry (riporto in uscita dell'ALU)
O: overflow (supero di capacità dell'ALU)

2) Progettare un formato unico per le seguenti istruzioni di salto:

J<flag>
JN<flag>
<flag> in {Z, S, C, O}

3) Effettuare la selezione del bit di flag da testare direttamente nella parte di controllo sfruttando il nuovo formato per le istruzioni di salto.