Architettura LC-3[1]

Il Little Computer 3 (LC-3)

L'architettura LC-3 è a 16 bit, ovvero contiene 2^{16} (65.536) allocazioni di memoria di dimensione 16 bit, indirizzabili con una codifica a 16 bit. Ci sono 8 registri general purpose cui ci si riferisce con i nomi R0 … R7 che possono essere usati per scrivere e leggere dati temporanei. I dati sono conservati sempre in complemento a due, non c'è alcun supporto nativo per dati senza segno o in virgola mobile.
Le istruzioni si esprimono con una codifica a 16 bit, di cui i primi 4 bit rappresentano il codice operativo (opcode). L'Instruction Set Architecture definisce 15 delle 16 istruzioni disponibili tramite opcode. Le istruzioni aritmetiche comprendono la somma e le operazioni logiche di AND e NOT bit a bit; queste operazioni sono sufficienti per implementare un gran numero di operazioni aritmetiche più complesse e tutte le operazioni logiche, essendo <AND,NOT> un insieme logico completo. Sono fornite inoltre istruzioni di salto condizionato e incondizionato; le condizioni si basano sul segno (positivo, negativo o zero) dell'ultimo dato scritto nei registri. Istruzioni specifiche supportano l'utilizzo di sub-routine.
L'esecuzione delle istruzioni è regolata da una Macchina a Stati Finiti, mentre l'elaborazione viene eseguita da un datapath; il modello SIS di un LC-3 è pertanto una struttura FSMD.

Datapath

`[datapath.blif]`
Il datapath è l'unità di calcolo che contiene tutte le unità di elaborazione ed i registri necessari per l'esecuzione delle istruzioni nella CPU. L'unità esecutiva contiene i seguenti componenti principali:

- un banco di 8 registri, ciascuno di 16 bit;
- l'unità aritmetico-logica (ALU) che permette di effettuare addizioni e operazioni logiche svolte in complemento a due.

Oltre a questi due componenti principali fanno parte del datapath anche le unità funzionali che operano sull'informazione, i registri che memorizzano le informazioni, ed i bus che le trasportano da un punto all'altro.

[1] Il presente documento è una rivisitazione a fini didattici del progetto originale di Ambra Fausti.

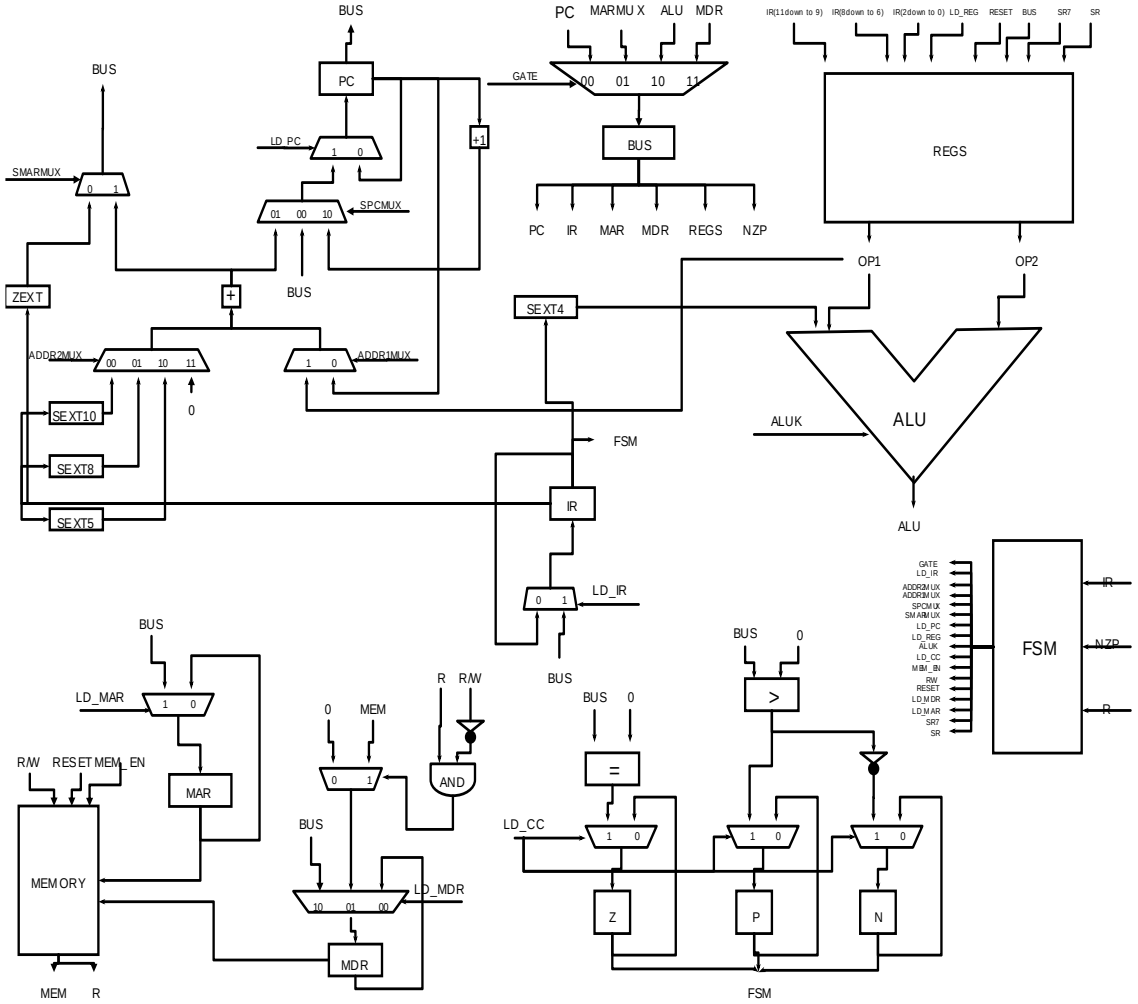
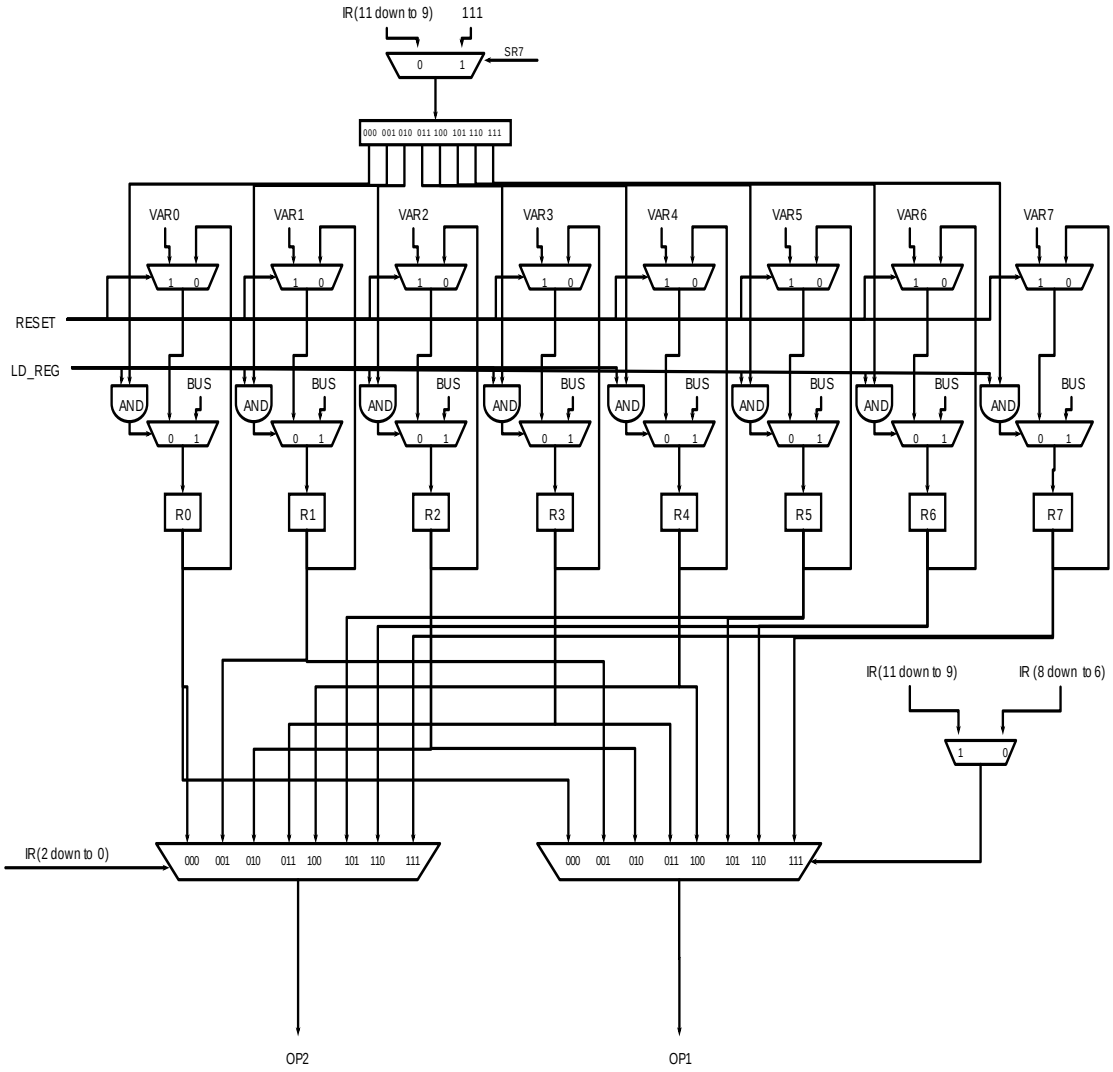


Figura 1: datapath

Banco di registri

`[regs.blif]`

Il banco di registri è composto da 8 registri di 16 bit ciascuno che consentono di salvare temporaneamente i risultati dell'elaborazione.



Input:

- **RESET**: viene usato come selettore del primo MUX di scrittura, prima di ciascun registro. Quando viene posto a 1 seleziona la variabile che dovrebbe essere inserita nel registro (***vedi …), altrimenti seleziona il valore attuale del registro per conservare l'informazione corrente;
- **IR(11 down to 9)**, **IR(2 down to 0)**, **IR(8 down to 6)**: sono i bit dell'instruction register *** (IR) che selezionano il registro in cui scrivere o da cui leggere in base all'istruzione;
- **LD_REG**: viene posto come primo operatore dell'operazione logica AND prima del secondo MUX di scrittura di ciascun registro e ha la funzione di decidere se scrivere o meno all'interno del registro;
- **SR7**: viene usato come selettore del MUX posto prima del decoder. Quando vale 1 seleziona l'ingresso costituito dai bit 111, che specificano l'indirizzo di R7 (***vedi …), altrimenti seleziona i bit dell'IR che specificano l'indirizzo del registro in cui scrivere;

- **BUS**: contiene il valore che si vuole memorizzare nel registro richiesto e per tale motivo viene messo come ingresso del secondo MUX di scrittura, prima di ciascun registro;
- **SR**: viene usato come selettore del MUX, posto prima del primo MUX di lettura, che ha come ingressi IR(8 down to 6) e IR(11 down to 9) e viene attivato in presenza di specifiche istruzioni.

Output:

- OP1, OP2: sono gli operandi che verranno passati alla ALU come elementi su cui effettuare le operazioni logiche e aritmetiche.

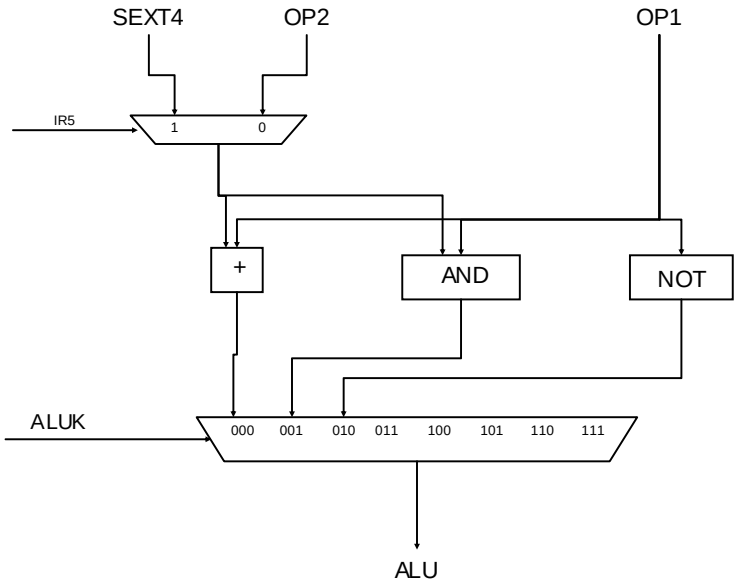
Componenti:

- **MUX SR7** [`mux1s3.blif`]: ha come ingressi IR(11 down to 9) e 111, e come selettore SR7. Quando SR7 vale 1 seleziona l'ingresso costituito dai bit 111, che specificano l'indirizzo di R7 (***vedi …), altrimenti seleziona i bit dell'IR che specificano l'indirizzo del registro in cui si vuole scrivere;
- **DECODER** [`decoder.blif`]: ha come unico ingresso il risultato del MUX iniziale e in base a questo alza una delle 8 uscite mentre le sette rimanenti rimangono poste a 0;
- **PRIMO MUX SCRITTURA** [`mux1s16.blif`]: posto prima di ciascun registro, ha come ingressi il valore della variabile che si vuole inserire, il valore del registro in quell'istante e RESET come selettore. Quando RESET viene posto a 1 seleziona la variabile che dovrebbe essere inserita nel registro secondo le specifiche del programma (***vedi …), altrimenti seleziona il valore attuale del registro in modo tale da non perderlo ad ogni ciclo di clock. L'output di tale MUX viene collegato all'ingresso del secondo MUX di scrittura
- **AND** [`and.blif`]: svolge l'operazione logica AND tra LD_REG e il valore del decoder corrispondente al registro. L'uscita viene usata come selettore del secondo MUX di scrittura;
- **SECONDO MUX SCRITTURA** [`mux1s16.blif`]: posto prima di ciascun registro, ha come ingressi il valore del BUS, l'uscita del primo MUX di scrittura e come selettore l'uscita dell'operazione logica AND; quando questa vale 1 viene selezionato il valore del BUS contenete l'informazione da salvare all'interno del registro, altrimenti seleziona l'uscita del primo MUX corrispondente al valore del registro al tempo t oppure al valore della variabile del programma da inserire. L'uscita viene collegata direttamente all'ingresso del registro;
- **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **R7** [`reg16.blif`]: registri a 16 bit all'interno dei quali vengono temporaneamente memorizzati i risultati dell'elaborazione. Gli indirizzi di ciascun registro sono espressi in 3 bit;
- **MUX SR** [`mux1s3.blif`]: posizionato prima del primo MUX di lettura, ha come ingressi IR(8 down to 6) e IR(11 down to 9), e come selettore SR. L'uscita viene usata come selettore del primo MUX di lettura e specifica l'indirizzo del registro che si vuole leggere;

- **PRIMO MUX LETTURA** [`mux3s16.blif`]: ha come ingressi tutti i valori dei registri e come selettore l'uscita del MUX SR. Il valore del registro selezionato viene posto come uscita e andrà a costituire il valore di OP1, necessario per diverse operazioni;
- **SECONDO MUX LETTURA** [`mux3s16.blif`]: ha come ingressi tutti i valori dei registri e come selettore IR(2 down to 0). Il valore del registro selezionato viene posto come uscita e andrà a costituire il valore di OP2, necessario per diverse operazioni.

Arithmetic Logic Unit

Questo modulo ha il compito di processare l'informazione.



Input:

- **SEXT**: posto come input di R2MUX;
- **OP2**: posto come input di R2MUX;
- **OP1**: collegato direttamente alle diverse operazioni aritmetiche e logiche;
- **SR2MUX**: ha la funzione di selettore di R2MUX;
- **ALUK**: (a 3 bit) viene usato come selettore in ALUMUX; seleziona il risultato dell'operazione desiderata;

Output:

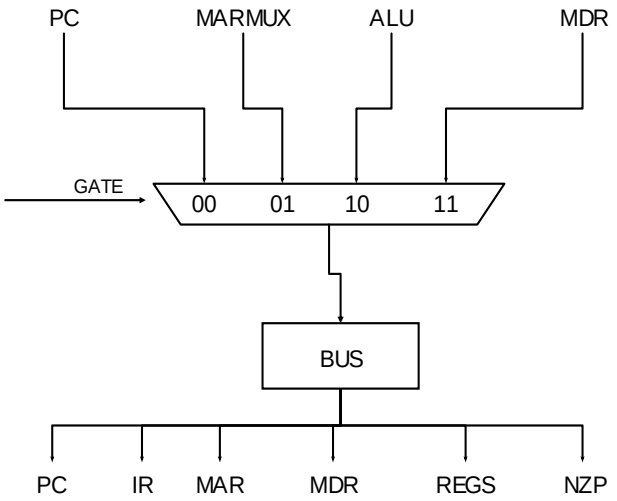
L'output della ALU viene collegato direttamente al BUS.

Componenti:

- **R2MUX** [`mux1s16.blif`]: ha come ingressi SEXT ed OP2, e come selettore il bit 5 di IR. Ha la funzione di decidere se porre come primo operatore delle successive operazioni l'offset specificato dall'IR (SEXT) oppure OP2, valore letto dal banco di registri;
- **ADD** [`add16.blif`]: esegue l'operazione di somma in complemento a due tra l'output del R2MUX e OP1;
- **AND** [`and16.blif`]: esegue l'operazione logica AND tra l'output di R2MUX e OP1;
- **NOT** [`not16.blif`]: esegue l'operazione logica NOT su OP1;
- **ALUMUX** [`mux3s16.blif`]: seleziona il risultato dell'oprazione da passare in uscita all'ALU. Sebbene il selettore ALUK abbia una codifica a 3 bit, solo 3 codici sono occupati: 000-ADD, 001-AND e 010-NOT. Tutti gli altri casi sono liberi per l'implementazione di altre operazioni algebriche e logiche.

Bus driver

Questo modulo ha la funzione di arbitro del BUS. Il BUS è un mezzo utilizzato per condividere dati: per questo si può accedervi sia in lettura che in scrittura da più moduli. Letture simultanee non creano problemi, perché non portano a corruzione il segnale di uscita. Non si devono però verificare scritture simultanee, o il risultato che viaggia su bus diventa non prevedibile e non deterministico.
Il modo migliore per implementare il BUS in SIS è tramite l'utilizzo di un registro.

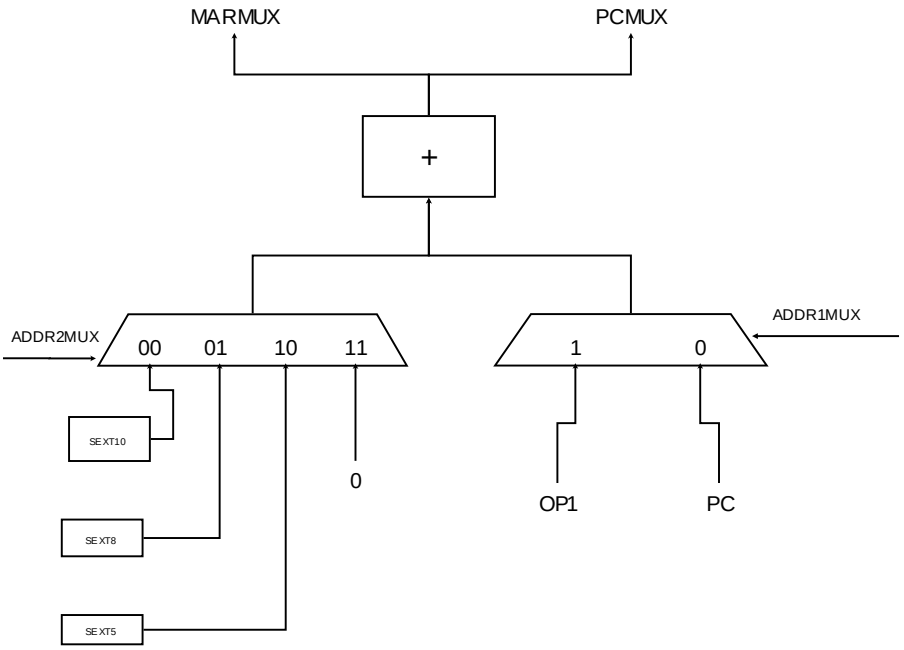


Componenti:

- **BUSMUX** [`mux2s16.blif`]: prende come ingressi PC, MARMUX, ALU, MDR e come selettore GATE. Quando GATE vale 00 scrive nel registro del BUS il valore di PC, quando GATE vale 01 scrive il valore di MARMUX, quando GATE vale 10 scrive il valore di ALU e quando GATE vale 11 scrive il valore di MDR;
- **BUS** [`reg16.blif`]: ha la funzione di memorizzare il valore fornitogli da BUSMUX e riportarlo al ciclo di clock successivo a PC, IR, MAR, MDR, REGS e NZP.

Adder

Questo modulo serve ad evitare di coinvolgere la ALU nell'esecuzione di somme che riguardano il Program Counter (PC) necessarie per i salti all'interno del programma o semplicemente per fornire l'indirizzo dell'istruzione successiva.



Il sommatore vero e proprio ha due input provenienti dai risultati dei MUX, inseriti per selezionare quali operatori vanno sommati. Il risultato della somma viene poi collegato al PCMUX e al MARMUX.

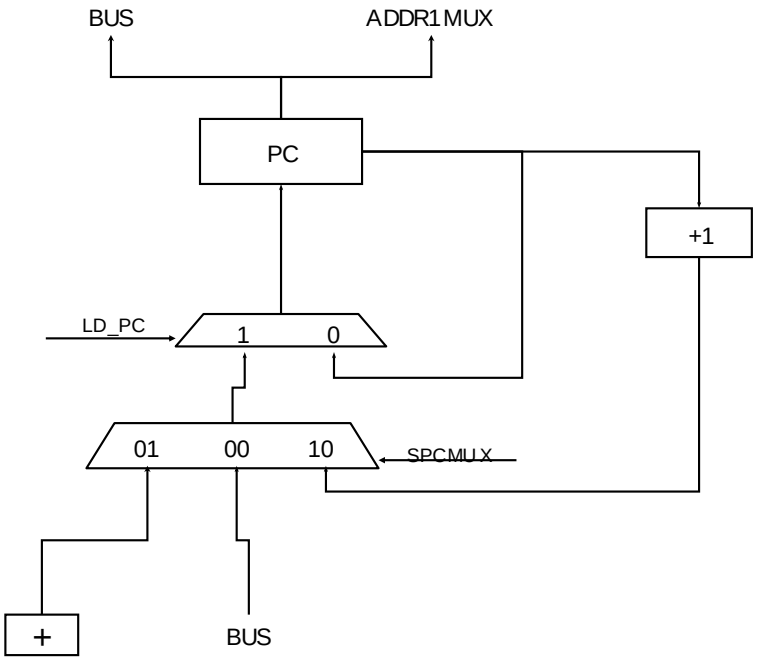
Componenti:

- **SEXT**: questo modulo estende di segno il numero in complemento a due che gli viene fornito dall'IR, scartando i bit in eccesso. Nel datapath ci sono quattro moduli di questo tipo che differiscono per il numero di bit che vengono mantenuti:
  - SEXT10 [`sext10.blif`]: riceve in ingresso 11 bit, estende di segno il valore e fornisce in uscita 16 bit collegati a ADDR2MUX;

  - SEXT8 [`sext8.blif`]: riceve in ingresso 9 bit, estende di segno il valore e fornisce in uscita 16 bit collegati a ADDR2MUX;
  - SEXT5 [`sext5.blif`]: riceve in ingresso 6 bit, estende di segno il valore e fornisce in uscita 16 bit collegati a ADDR2MUX;
  - SEXT4 [`sext4.blif`]: riceve in ingresso 5 bit, estende di segno il valore e fornisce in uscita 16 bit collegati a SR2MUX della ALU.
- **ADDR1MUX** [`mux1s16.blif`]: seleziona il primo addendo tra PC ed il registro letto nel banco di registri;
- **ADDR2MUX** [`mux1s16.blif`]: seleziona il secondo addendo tra SEXT10, SEXT8, SEXT5 e 0.

Program Counter

Il Program Counter (PC) ha il ruolo di fornire al MAR l'indirizzo dell'istruzione che dovrà entrare nella fase di fetch all'inizio del ciclo di clock successivo. Effettua questa operazione attraverso il bus driver.



Componenti:

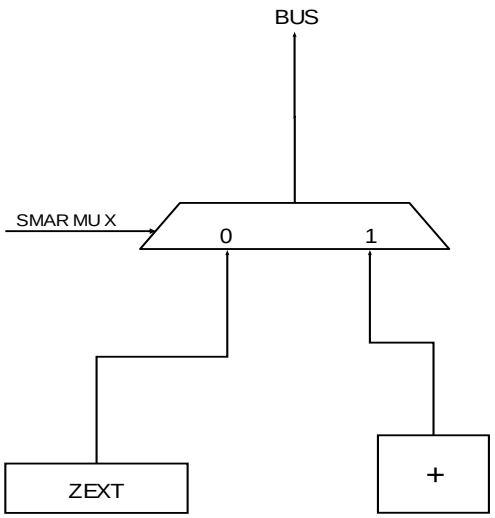
- **PCMUX1** [`mux2s16.blif`]: ha come ingressi il risultato di ADDER, il BUS, il valore di PC incrementato di 1 e come selettore SPCMUX (2 bit). L'uscita è collegata direttamente al MUX prima di PC;
- **PCMUX2** [`mus1s16.blif`]: ha come ingressi l'output di PCMUX1, il valore di PC e come selettore LD_PC. Quando quest'ultimo vale 0 seleziona il valore di PC per permettere di conservare il suo valore ad ogni ciclo di clock dell'istruzione

corrente, altrimenti seleziona il valore di PCMUX1 se deve memorizzare l'indirizzo di una nuova istruzione;

- **+1** [`add16.blif`]: incrementa di 1 il valore di PC per ottenere l'indirizzo dell'istruzione successiva (in assenza di salti);
- **REGISTRO** [`reg16.blif`]: memorizza l'indirizzo dell'istruzione successiva. L'output del registro è collegato al BUS DRIVER, a +1 e a ADDR1MUX.

Marmux

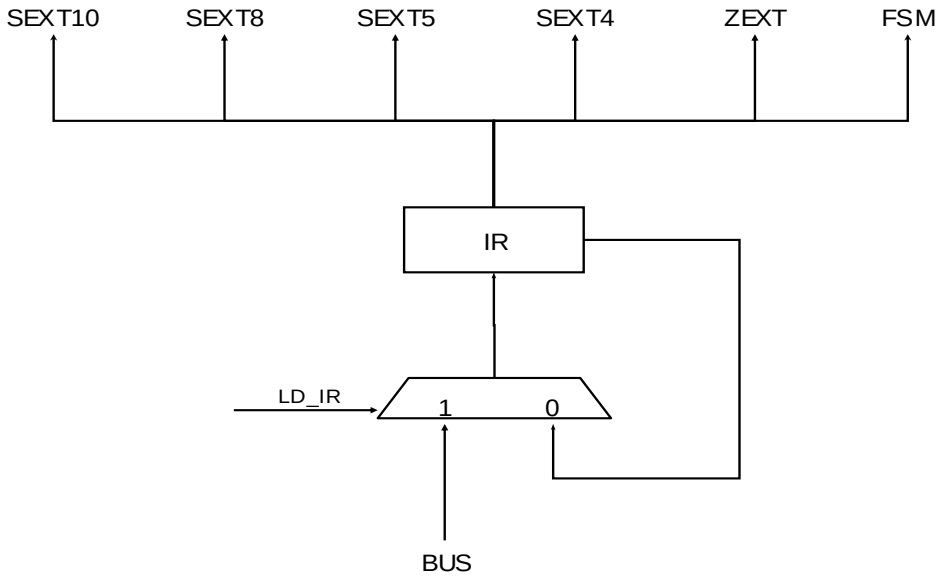
Questo modulo controlla quale delle due sorgenti fornirà al MAR l'indirizzo appropriato durante l'esecuzione di una LOAD o di una STORE. Uno dei due input al MARMUX è ottenuto aggiungendo il PC incrementato o un registro base ad un valore fornito dall'IR. L'output è collegato direttamente al BUS DRIVER.



Gli input di MARMUX [`mux1s16.blif`] sono +, ZEXT, che tiene i 7 bit più significativi dell'input che riceve da IR e rimpiazza i restanti con zero, e SMARMUX, che funziona come selettore. Quando SMARMUX vale 1 seleziona il risultato di ADD altrimenti seleziona ZEXT.

Instruction Register

Il ruolo di questo modulo è quello di memorizzare l'istruzione che viene eseguita e decodificata durante tutto il ciclo di decodifica (recupero degli operandi) ed esecuzione. Il suo output è collegato a uno dei moduli che estendono con o senza segno il numero di bit che si desidera mantenere ed al controllore.

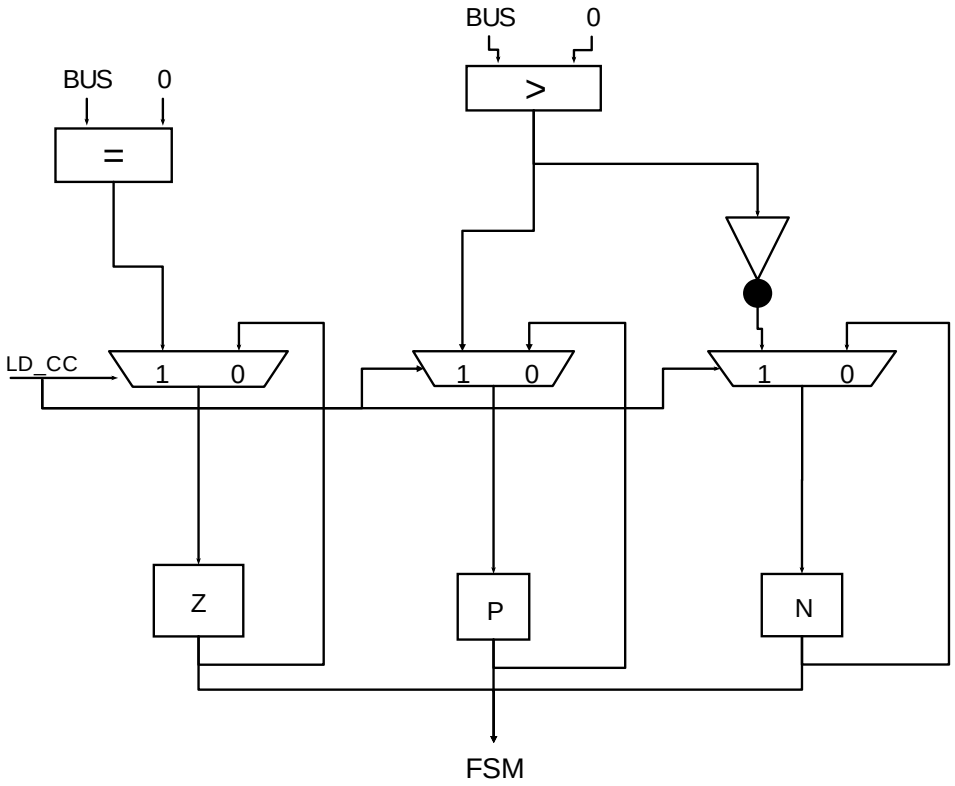


Componenti:

- **MUX_IR** [`mux1s16.blif`]: prende in ingresso il valore di BUS, il valore di IR e LD_IR come selettore. Quando questo vale 0 mette in uscita il valore di IR, in modo tale che mantenga il suo valore corrente, altrimenti prende il valore di BUS che fornisce una nuova istruzione;
- **REGISTRO** [`reg16.blif`]: memorizza il valore dell'istruzione corrente.

NZP logic

Questo modulo ha la funzione di dire se il valore che gli si passa in ingresso è negativo, positivo o nullo (il suo nome infatti sta per Negative, Zero, Positive).
Il suo input è collegato al BUS DRIVER.

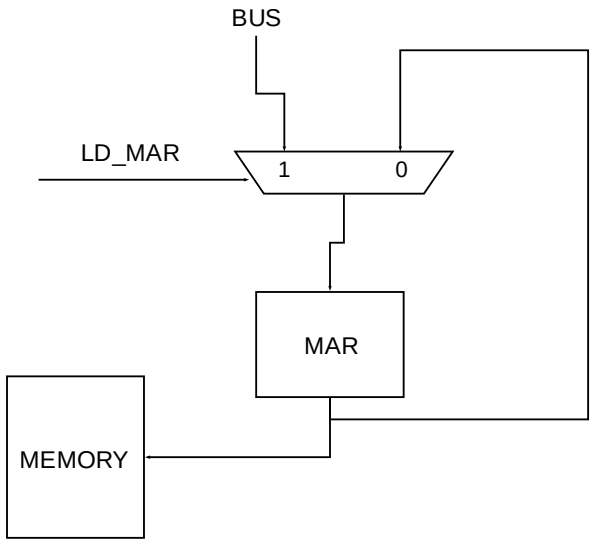


Componenti:

- **EQUAL** [`equal16.blif`]: controlla se il valore proveniente dal BUS DRIVER è pari a 0;
- **GREATER** [`greaterc2.blif`]: controlla se il valore proveniente dal BUS DRIVER è maggiore di 0;
- **NOT** [`not1.blif`]: nega il risultato di GREATER;
- **MUX** [`mux1s.blif`]: posizionati prima di ciascun registro, ciascun MUX ha come ingresso il valore del registro prima del quale è stato posto, il risultato dell'operazione e tutti e tre hanno come selettore LD_CC che decide se voler aggiornare i valori di N, Z, P oppure no;
- **N** [`reg.blif`]: posto a 1 se il valore in ingresso è negativo, altrimenti a 0;
- **Z** [`reg.blif`]: posto a 1 se il valore in ingresso è pari a 0, altrimenti a 0;
- **P** [`reg.blif`]: posto a 1 se il valore in ingresso è positivo, altrimenti a 0.

Memory Address Register (MAR)

Questo modulo rappresenta il registro della memoria che salva gli indirizzi: viene usato per accedere in lettura o in scrittura ad una determinata locazione di memoria. Una delle funzioni principali del MAR riguarda la possibilità di accedere all'indirizzo di memoria contenente la prossima istruzione da eseguire. Il suo input è collegato a BUS DRIVER mentre il suo output è collegato alla memoria.



Componenti:

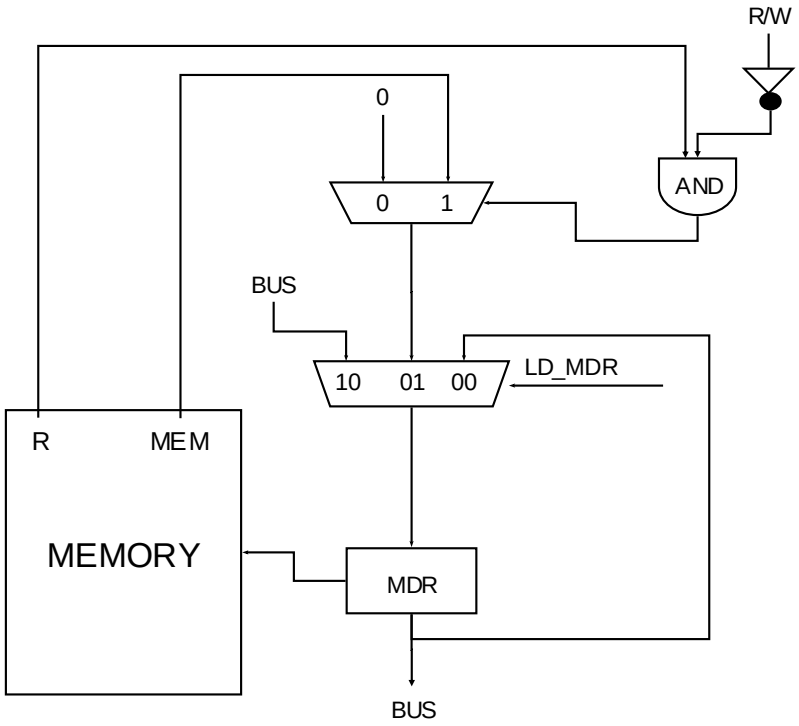
- **MUX** [`mux1s16.blif`]: posto prima di MAR, prende come input il valore di BUS, il valore di IR e LD_IR come selettore. Quando questo vale 0 mette in uscita il valore di MAR, in modo tale che mantenga il suo valore per tutta la durata dell'istruzione, altrimenti prende il valore di BUS nel caso in cui si voglia eseguire una nuova istruzione;
- **REGISTRO** [`reg16.blif`]: memorizza l'indirizzo della cella di memoria.

Il codice di questo modulo trasferisce solo i quattro bit meno significativi alla memoria. Il motivo di tale scelta sta nel fatto che all'interno della memoria ci sono 16 registri che svolgono la funzione di locazione di memoria. Per decidere se leggere o scrivere questi quattro bit verranno collegati rispettivamente ad un MUX, svolgendo la funzione di selettore, ad un decoder, al fine di attivare il bit che permette la scrittura sulla locazione richiesta. Se fossero stati inseriti tutti i 16 bit del MAR, considerando che un multiplexer con selettore di n bit ha bisogno di 2^n ingressi e che un decoder con un ingresso di n bit deve avere 2^n output, sarebbero state necessarie 65536 locazioni di memoria!
Si utilizzano i bit meno significativi perché il PC manda al MAR l'indirizzo dell'istruzione da eseguire e poi viene incrementato di 1 per puntare quindi all'istruzione successiva: prendendo i quattro bit più significativi questo incremento non verrebbe notato e il MAR sarebbe sempre indirizzato alla stessa locazione di memoria, quindi verrebbe eseguita ripetute volte la medesima istruzione.

Memory Data Register (MDR)

Questo componente contiene i dati che devono essere scritti in memoria oppure i dati che sono stati letti dalla memoria: per questo avrà bisogno di più componenti che

permettano che il suo input provenga dalla memoria o dal bus e che il suo output finisca o nella memoria o nel bus.

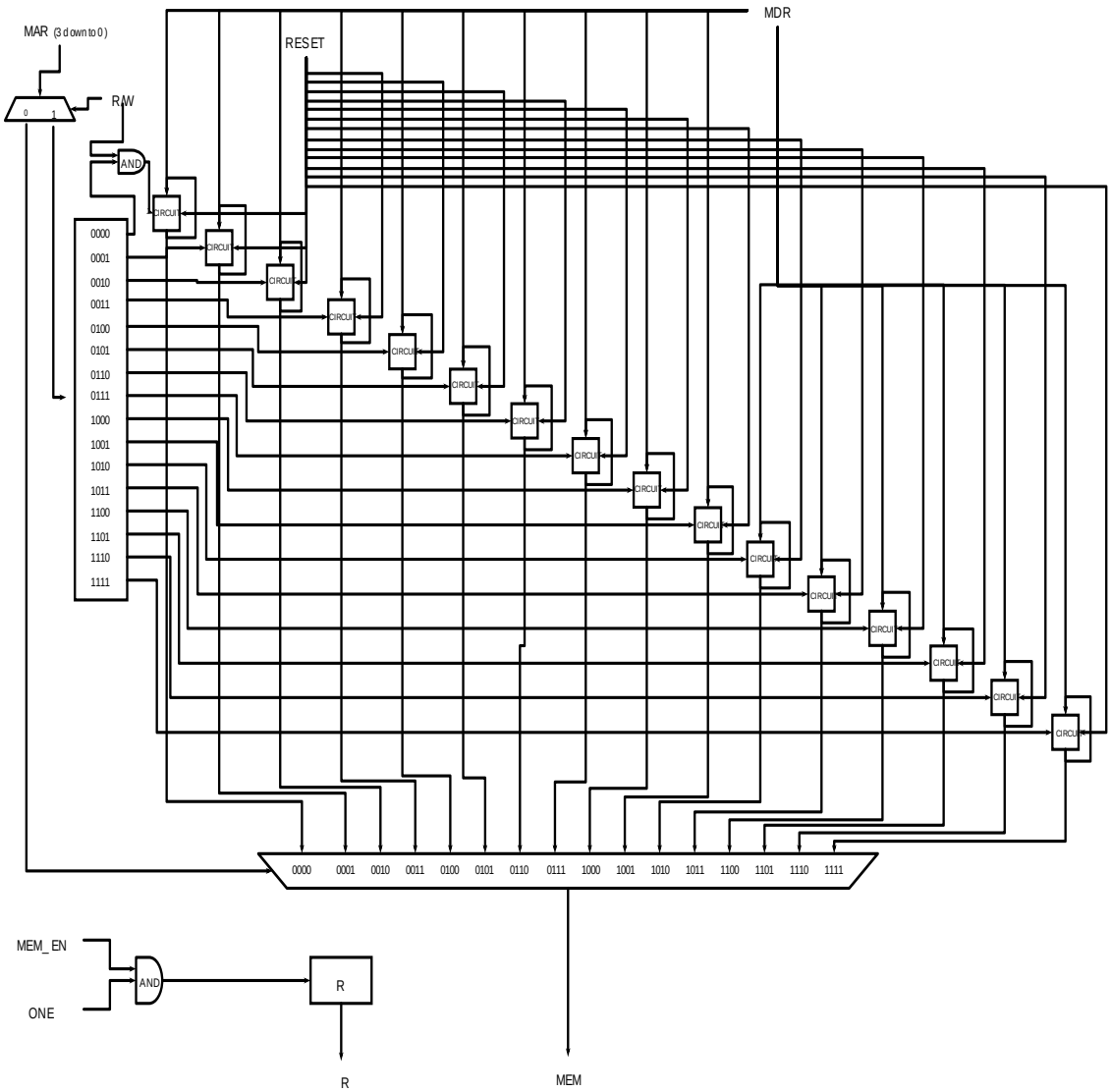


Componenti:

- **AND** [`and.blif`] ha come ingressi R ("ready", indica che la memoria è pronta per trasferire o ricevere il dato) e R/W negato (posto a 0 quando si vuole leggere e a 1 quando si vuole scrivere in memoria). Quando entrambi valgono 1 vuol dire che la memoria è pronta per trasferire o ricevere il dato e che si vuole leggere il dato proveniente da questa quindi l'operazione logica pone l'uscita a 1;
- **PRIMO MUX** [`mux1s16.blif`] prende in ingresso il valore letto all'interno della memoria, la costante 0 e come selettore l'output dell'operazione logica AND. Quando il selettore vale 1 vuol dire che si vuole inserire il valore letto in MDR, altrimenti viene inserita la costante;
- **SECONDO MUX** [`mux2s16.blif`] prende in ingresso il valore del BUS, l'output del PRIMO MUX, il valore corrente di MDR e come selettore LD_MDR (2 bit). Quando LD_MDR vale 10 seleziona il valore del BUS, quando vale 01 seleziona il valore in uscita del PRIMO MUX mentre quando vale 00 seleziona il suo valore al tempo t in modo tale da non perdere il dato, in mancanza di operazioni su di esso;
- **REGISTRO** [`reg16.blif`] prende in ingresso l'uscita del secondo MUX e ha la funzione di memorizzare i dati che devono essere scritti in memoria oppure i dati che devono essere letti dalla memoria. L'output è collegato sia a MEMORY che al BUS DRIVER.

Memoria

La memoria ha lo scopo di conservare le informazioni e permetterne il recupero durante l'esecuzione del programma; è organizzata come una tabella di celle, ciascuna delle quali è identificata univocamente da un indirizzo.
La memoria puo' eseguire soltanto due operazioni: lettura (LOAD) e scrittura (STORE). La lettura consiste nel trasferire a MDR il contenuto della cella il cui indirizzo è specificato in MAR; la scrittura consiste nel trasferire il valore di MDR alla cella di memoria il cui indirizzo è specificato in MAR.
Per simulare le celle di memoria si utilizzano 16 registri di 16 bit ciascuno.

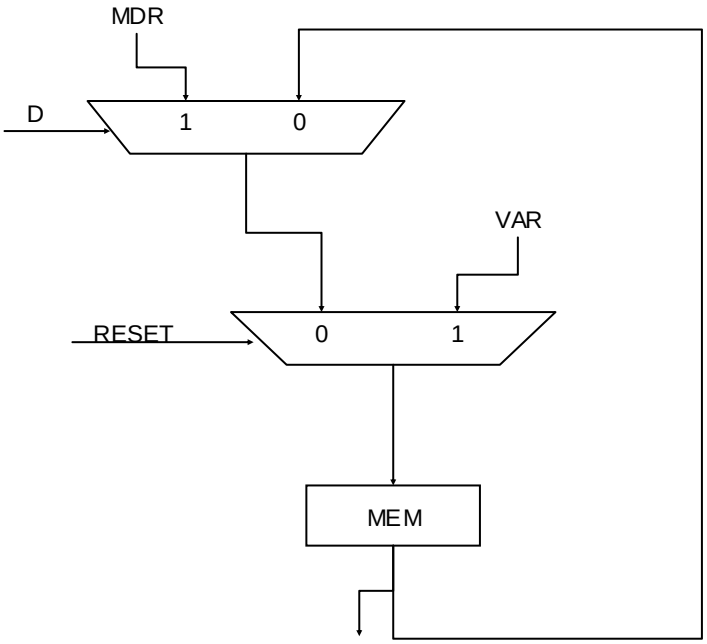


Componenti:

- **AND** [`and.blif`]: prende come ingessi MEM_EN (segnale fornito dal controllore, attivato quando si vuole usare la memoria) e la costante 1. Il risultato dell'operazione logica viene inserito all'interno del registro R.

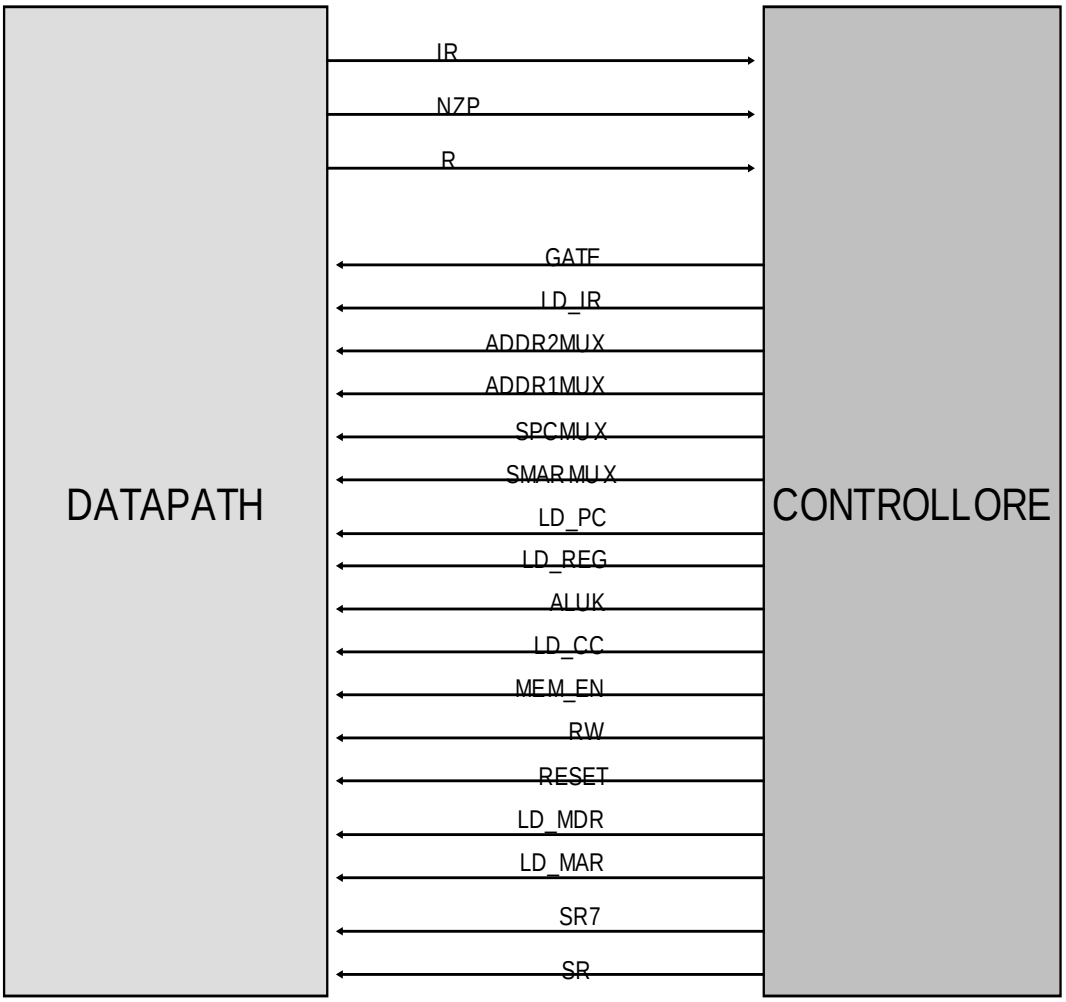
- **REGISTRO** [`reg.blif`]: usato per salvare il risultato dell'operazione logica AND perché è necessario che la risposta della memoria "ready" non arrivi nello stesso ciclo di clock in cui viene attivato MEM_EN ma in quello successivo, quando quindi sarà pronto il dato da leggere oppure l'operazione di scrittura terminata;
- **DEMUX** [`demux1s4.blif`]: riceve in ingresso i 4 bit meno significativi del MAR e come selettore R/W; quando il selettore vale 1 è richiesta un operazione di scrittura quindi l'input viene fatto passare sulla seconda uscita e collegato al DECODER mentre la prima uscita viene posta a 0000. Se invece R/W vale 0 i 4 bit di MAR vengono messi sulla prima uscita e posti come selettore di un multiplexer e la seconda uscita viene posta a 0000;
- **MUX DI LETTURA** [`mux4s16.blif`]: prende in ingresso i valori dei sedici registri contenuti all'interno di CELL e come selettore i quattro bit della prima uscita del DEMUX. Ha lo scopo di selezionare quale cella di memoria leggere;
- **DECODER** [`decoder4.blif`]: prende come input i quattro bit che escono dalla seconda uscita del DEMUX. In base al valore dell'ingresso viene posta a 1 una delle sedici uscite mentre tutte le altre sono poste a 0. Grazie a questa proprietà del decoder è possibile scrivere sulla sola cella di memoria selezionata dal bit alzato;
- **AND** [`and.blif`]: posto prima di MEM0, questo componente serve per correggere il funzionamento del DECODER. Infatti quando il selettore R/W che entra nel DEMUX vale 0 vuol dire che è in corso un'operazione di lettura: la seconda uscita del DEMUX però rimane posta a 0000 e viene comunque collegata al DECODER ponendo quindi il primo bit d'uscita a 1. Si rischia quindi di andare a scrivere sulla prima cella di memoria un dato sbagliato. Per rimediare è stato quindi collegato il primo bit in uscita dal DECODER all'ingresso dell'operazione logica AND e come secondo ingresso invece R/W: quando entrambi i bit valgono 1 vuol dire che si vuole scrivere sulla prima cella di memoria e quindi l'uscita viene posta a 1 mentre in tutti gli altri casi o è in corso un'operazione di scrittura (che però non riguarda la prima cella di memoria) oppure è in corso un'operazione di lettura, l'uscita viene posta a 0;
- **CELL** [`cell.blif`] composta a sua volta da 3 componenti:
    - **PRIMO MUX**: prende come ingressi il valore di MDR, il valore corrente del REGISTRO della cella di memoria e come selettore il bit d'uscita del DECODER corrispondente alla cella di memoria. Quando questo bit è alzato significa che si vuole scrivere il valore di MDR all'interno della cella di memoria, altrimenti mantiene il valore attuale selezionando l'uscita del REGISTRO;
    - **SECONDO MUX**: prende in ingresso l'output del PRIMO MUX, il valore di una variabile e come selettore RESET. La variabile servirà per inserire all'interno della cella di memoria un'istruzione da eseguire. RESET viene posto a 1 quindi quando si vuole inserire un programma nella memoria, selezionando in ciascuna cella il valore della variabile contenente l'istruzione. Se invece RESET vale 0 si passa l'output del PRIMO MUX;

- **REGISTRO**: memorizza il valore che gli viene fornito dal SECONDO MUX e collega l'uscita sia al PRIMO MUX si all'uscita della cella di memoria, che verrà poi collegata al MUX DI LETTURA.



Controllore (FSM)

Il controllore è composto dalle strutture usate dal processore per regolamentare l'esecuzione e fa interagire tutti i componenti e decide il flusso di esecuzione. È definito come una Macchina a Stati Finiti (FSM), che ad ogni ciclo di clock attiva i segnali per far eseguire determinate operazioni che regolamenta il flusso di esecuzione.
L'unità più importante per l'esecuzione di un processore è data dalle istruzioni: sono sequenze di bit che identificano operazioni, operandi e locazioni di memoria (o registri) coinvolte. Tutto il comportamento del processore sarà regolamentato dalla lettura di istruzioni, che compongono programmi e procedure.
Altri due componenti fondamentali per il controllore sono il Program Counter (PC) e l'Instruction Register (IR). L'IR contiene l'istruzione che è in esecuzione e determina le operazioni che devono essere portate avanti dal processore. Il PC invece contiene l'indirizzo della prossima istruzione che deve essere eseguita, una volta che l'istruzione corrente è terminata.



Input:

- **IR**: contiene l'istruzione da eseguire;
- **N**, **Z**, **P**: condizioni necessarie per le istruzioni di salto;
- **R**: svolge la funzione di risposta della memoria.

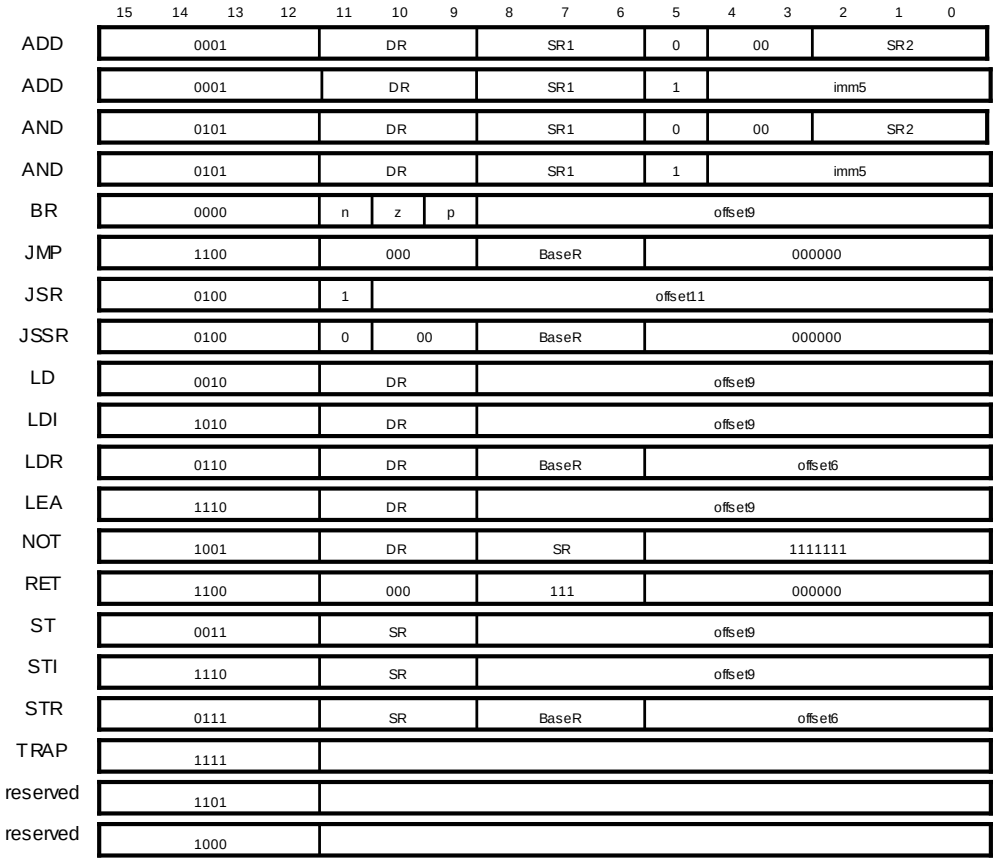
Output:

- **GATE**: selettore del MUX prima del BUS;
- **LD_IR**: selettore del MUX prima di IR;
- **ADDR2MUX**: selettore di ADDR2;
- **ADDR1MUX**: selettore di ADDR1;
- **SMARMUX**: selettore di MARMUX;
- **SPCMUX**: selettore di PCMUX;
- **LD_PC**: selettore del MUX prima di PC;
- **LD_REG**: primo operatore dell'operazione logica NAD all'interno del banco dei registri;
- **ALUK**: selettore del MUX all'interno della ALU;
- **LD_CC**: selettore di ciascun MUX posto prima di N,Z,P;

- **MEM_EN**: primo operatore dell'operazione logica AND all'interno della memoria;
- **R/W**: selettore del DEMUX all'interno della memoria, secondo operatore dell'operazione logica AND prima della prima cella di memoria e secondo operatore dell'operazione logica prima del registro MDR;
- **RESET**: selettore del primo MUX posto prima di ciascun registro all'interno del banco di registri e selettore del secondo MUX prima di ciascun registro all'interno della cella di memoria;
- **LD_MDR**: selettore del MUX prima di MDR;
- **LD_MAR**: selettore del MUX prima di MAR;
- **SR7**: selettore del MUX posto prima del decoder all'interno del banco di registri;
- **SR**: selettore del MUX posto prima del SECONDO MUX SCRITTURA all'interno del banco di registri.

Le istruzioni sono l'unità fondamentale dell'esecuzione di un processore. Sono sequenze di bit come normali dati, ma vengono interpretate come codifiche di operazioni con i rispettivi operandi, registri e locazioni di memoria coinvolti.
Ogni istruzione è costituita da due parti fondamentali:

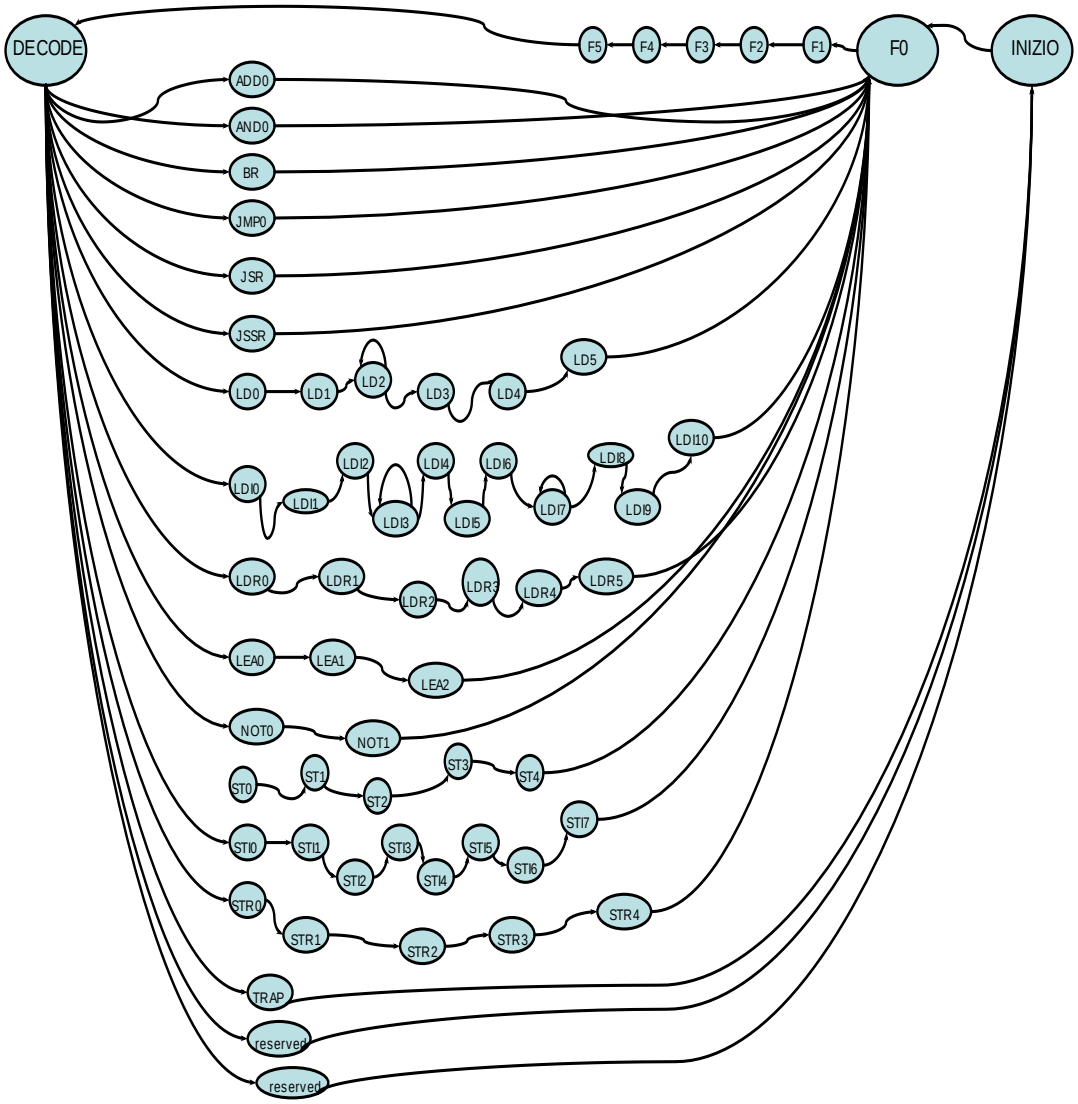
- ***opcode***: codice di lunghezza fissata (4 bit, da 15 a 12) che identifica l'operazione;
- ***operands***, operandi coinvolti nell'esecuzione dell'operazione. Questi rimanenti 12 bit possono contenere flag, registri (identificati con 3 bit), offset o indirizzi di memoria.



La sequenza di passi di esecuzione è detta Instruction Cycle e ogni passo è detto *fase*.
Le fasi fondamentali sono tre:

- **FETCH**: fase in cui il processore recupera l'istruzione da eseguire dalla memoria e la carica nell'IR; per fare questo userà il PC, che tiene traccia della successiva istruzione da eseguire e che sarà incrementato alla fine dell'operazione di lettura da memoria. Questa fase è comune a tutte le istruzioni;
- **DECODE**: questa fase esamina gli operandi contenuti nell'istruzione per identificare il flusso di esecuzione che dovrà seguire la macchina a stati. Alcuni operandi possono essere recuperati e salvati nei registri;
- **EXECUTE**: fase in cui avviene la vera e propria esecuzione di un'istruzione. In alcuni casi segue il salvataggio del risultato in memoria o in un registro.

Una volta terminata la terza fase, l'esecuzione ripartirà dalla lettura di una nuova istruzione da memoria: il PC sarà infatti aggiornato al suo indirizzo.



INIZIO

Questo stato è necessario all'inizio dell'esecuzione per caricare il programma in memoria ed assegnare ai registri i valori desiderati. Per qualsiasi valore in ingresso l'output RESET viene posto a 1 e tutti gli altri posti a 0.

```
-------------------- INIZIO F0 00000000000000000100000
```

FETCH

È comune a tutte le istruzioni, si occupa infatti di recuperare l'istruzione da eseguire dalla memoria. Per far questo, i passi fondamentali sono:

- Caricamento del PC nel registro MAR, che conterrà l'indirizzo dell'istruzione corrente;
- Incremento del PC, che conterrà l'indirizzo dell'istruzione successiva;
- Lettura da memoria: l'istruzione sarà restituita nel registro MDR;

- Salvataggio dell'istruzione corrente nell'IR

La fase di fetch ha quindi bisogno di 6 stati, chiamati F0, F1, F2, F3, F4, F.5.
Si entra all' interno di questi stati progressivamente, indipendentemente dai valori in ingresso.

- Output F0: GATE=00 (seleziona il valore di PC), SPCMUX=10 (seleziona +1 e quindi incrementa PC), LD_PC=1 (permette al valore incrementato di essere memorizzato in PC), il resto posto a 0. Stato prossimo F1;
- Output F1: LD_MAR=1, il resto posto a 0. Stato prossimo F2;
- Output F2: R/W=0 (per lettura), MEM_EN =1 (per accedere alla memoria), il resto posto a 0. Stato prossimo F3;
- Output F3: LD_MDR=01 (per inserire in MDR il valore riportato dalla memoria), il resto posto a 0. Stato prossimo F4;
- Output F4: GATE=11 (per ricevere il valore di MDR), il resto posto a 0. Stato prossimo F5;
- Output F5: LD_IR= 1 (per memorizzare il valore ricevuto dal BUS in IR), il resto posto a 0. Stato prossimo DECODE.

```
-------------------- F0 F1 00000001010000000000000
-------------------- F1 F2 00000000000000000000100
-------------------- F2 F3 00000000000000010000000
-------------------- F3 F4 00000000000000000001000
-------------------- F4 F5 11000000000000000000000
-------------------- F5 DECODE 00100000000000000000000
```

DECODE

Questo stato è stato usato per riconoscere quale tipo di istruzione eseguire. Gli output di questo stato sono funzione dell'opcode e verranno separatamente per ciascuna istruzione.

ADD

L'istruzione ADD rappresenta la somma tra due operandi. L'opcode è **0001** ed è riportata in due versioni, distinguibili tramite un flag (il bit 5): se il flag vale 0, la somma è tra i valori contenuti in due registri, mentre se il flag vale 1 la somma è tra il valore di un registro e una costante. Questo bit viene utilizzato come selettore di R2MUX all'interno della ALU. In entrambi i casi il risultato sarà salvato in un registro. Infine, andranno aggiornati i codici di condizione (P, N, Z). Gli stati totali sono 2.

```
if (bit[5] == 0)
     DR = SR1 + SR2;
else
     DR = SR1 + SEXT4;
```

Lo stato prossimo è ADD0 e l'output è: GATE=10 (seleziona il risultato fornito dalla ALU), ALUK = 000 (seleziona il risultato dell'operazione ADD).

```
0001---------------- DECODE ADD0 10000000000000000000000
```

Lo stato prossimo è ADD1 e l'output è: LD_CC=1 (permette di controllare se il valore del bus è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
-------------------- ADD0 ADD1 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- ADD1 F0 00000000000000000000000
```

AND

L'istruzione AND rappresenta l'operazione logica AND bit a bit tra i due operandi a 16 bit. L'opcode è **0101** ed è riportata in due versioni, distinguibili tramite un flag (il bit 5): se il flag vale 0, la somma è tra i valori contenuti in due registri, mentre se il flag vale 1 la somma è tra il valore di un registro e una costante. Questo bit viene utilizzato come selettore di R2MUX all'interno della ALU. In entrambi i casi il risultato sarà salvato in un registro. Infine, vanno aggiornati i codici di condizione (P, N, Z). Gli stati totali sono 2.

```
if (bit[5] == 0)
     DR = SR1 and SR2;
else
     DR = SR1 and SEXT4;
```

Lo stato prossimo è AND0 e l'output è: GATE=10 (seleziona il risultato fornito dalla ALU), ALUK = 001 (seleziona il risultato dell'operazione AND).

```
0101---------------- DECODE AND0 10000000000001000000000
```

Lo stato prossimo è AND1 e l'output è: LD_CC=1 (permette di controllare se il valore del BUS è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
-------------------- AND0 AND1 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- AND1 F0 00000000000000000000000
```

BR

L'istruzione BR rappresenta il salto condizionato. Ha come opcode **0000**. I primi tre bit dopo l'opcode sono utilizzati come flag che indicano proprietà (N, P, Z) dell'ultimo valore che è stato calcolato dalla ALU. Tali flag sono utilizzati per esprimere le condizioni di salto. L'esecuzione sarà la seguente: si confrontano i tre flag con i valori dei registri che memorizzano i codici di condizione, se in almeno un caso sia il flag che il registro sono a

uno, il programma salta all'indirizzo ottenuto negli ultimi bit dell'istruzione. Il risultato è un unico stato BR0.

```
if ((n and N) or (z and Z) or (p and P))
     PC = PC + SEXT8;
```

Se bit[11-9]=100 e bit[3]=1, oppure bit[11-9]=010 e bit[2]=1, oppure bit[11-9]=001 e bit[1]=1 allora lo stato prossimo è BR0 e l'output è : ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8), SPCMUX=01 (seleziona il risultato di ADD), LD_PC=1 (permette che il valore di PCMUX venga memorizzato nel PC). Altrimenti lo stato prossimo è F0 e tutti i bit in uscita sono posti a 0.

```
0000100---------1--- DECODE BR0 00001000110000000000000
0000010----------1-- DECODE BR0 00001000110000000000000
0000001-----------1- DECODE BR0 00001000110000000000000
0000100---------0--- DECODE F0 00000000000000000000000
0000010----------0-- DECODE F0 00000000000000000000000
0000001-----------0- DECODE F0 00000000000000000000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- BR0 F0 00000000000000000000000
```

JMP e RET

L'istruzione JMP rappresenta il salto non condizionato e, in un caso particolare, il ritorno da procedure. Ha come opcode **1100**. L'esecuzione sarà la seguente: si salta senza condizione all'indirizzo specificato dal contenuto del registro base identificato dai bit da 8 a 6. La RET è il caso particolare in cui il registro base è R7; tale registro viene infatti usato per tenere traccia del PC prima di chiamate a procedura, e la RET consiste proprio nell'istruzione di ritorno da procedura. Il risultato è un unico stato JMP0.

```
PC = BaseR;
```

L'output è ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=11 (seleziona 0), SPCMUX=01 (seleziona il risultato di ADD), LD_PC=1 (permette che il valore di PCMUX venga memorizzato nel PC);

```
1100---------------- DECODE JMP0 00011100110000000000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- JMP0 F0 00000000000000000000000
```

JSR e JSSR

L'istruzione JSR rappresenta il salto a procedura e ha come opcode **0100**. È l'istruzione che consente di eseguire chiamate e di invocare funzioni e procedure. L'esecuzione sarà la seguente: si incrementa PC e lo si salva nel registro R7, quindi si carica l'indirizzo che corrisponde alla prima istruzione della procedura in PC. Tale indirizzo si ottiene

dall'offset specificato nel registro base indicato se il bit[11] è 0 (JSRR), oppure sommando il PC all'offset specificato nell'istruzione se tale bit è posto a 1 (JSR). Il risultato è un unico stato per ciascuna istruzione: JSR0 E JSSR0.

```
R7=PC;
if (bit[11] == 0)
     PC= BaseR;
else
     PC= PC + SEXT10 ;
```

Se il bit[11] è posto a 0 lo stato prossimo è JSSR0 e l'output è: ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=11 (seleziona 0), SPCMUX=01 (seleziona il risultato di ADD), LD_PC=1 (permette che il valore di PCMUX venga memorizzato nel PC), LD_REG=1 (permette di accedere ai registri), SR7(seleziona il registro R7).

```
0100-----0---------- DECODE JSRR0 00011100111000000000010
```

Se il bit[11] è posto a 1 lo stato prossimo è JSR0 e l'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=00 (seleziona SEXT10), SPCMUX=01 (seleziona il risultato di ADD), LD_PC=1 (permette che il valore di PCMUX venga memorizzato nel PC)), LD_REG=1 (permette di accedere ai registri), SR7(seleziona il registro R7).

```
0100-----1---------- DECODE JSR0 00000000111000000000010
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- JSRR0 F0 00000000000000000000000
-------------------- JSR0 F0 00000000000000000000000
```

LD

L'istruzione LD è una lettura da memoria: l'indirizzo da cui leggere si calcola sommando un offset (specificato nell'istruzione) al PC; il risultato viene poi salvato in un registro. Ha come opcode **0010**. Il risultato consiste in 6 stati: LD0, LD1, LD2, LD3, LD4, LD5.

```
DR = mem[PC + SEXT8];
```

Lo stato prossimo è LD0. L'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8) , SMARMUX=1 (seleziona il risultato di ADD), GATE=01 (seleziona il valore di MARMUX).

```
0010---------------- DECODE LD0 01001010000000000000000
```

Lo stato prossimo è LD1. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- LD0 LD1 00000000000000000000100
```

Lo stato prossimo è LD2. L'output è: R/W=0 (consente la lettura), MEM_EN (permette di accedere alla memoria).

```
-------------------- LD1 LD2 00000000000000010000000
```

Se R=1 lo stato prossimo è LD3 e l'output è: LD_MDR=01(seleziona il valore proveniente dalla memoria).
Se R=0 lo stato prossimo è LD2 e tutti i bit dell'output hanno valore 0.

```
------------------1 LD2 LD3 00000000000000000001000
------------------0 LD2 LD2 00000000000000000000000
```

Lo stato prossimo è LD4. L'output è: GATE=11 (inserisce nel BUS il valore di MDR).

```
------------------- LD3 LD4 11000000000000000000000
```

Lo stato prossimo è LD5. L'output è: LD_CC=1 (permette di controllare se il valore del bus è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
------------------- LD4 LD5 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
------------------- LD5 F0 00000000000000000000000
```

LDI

L'istruzione LDI è una lettura indiretta da memoria. Ha come opcode **1010**. L'esecuzione sarà la seguente: si somma a PC l'offset specificato nell'istruzione, nei bit meno significativi, si va a leggere la locazione di memoria con questo indirizzo e il valore letto sarà l'indirizzo della locazione di memoria di cui vogliamo il valore. Il risultato di tale lettura sarà salvato in un registro. Il risultato consiste di 11 stati.

```
DR = mem[ mem[PC + SEXT8] ];
```

Lo stato prossimo è LDI0. L'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8), SMARMUX=1 (seleziona il risultato di ADD).

```
1010---------------- DECODE LDI0 00001010000000000000000
```

Lo stato prossimo è LDI1. L'output è: GATE=01 (seleziona il valore di MARMUX).

```
------------------- LDI0 LDI1 01000000000000000000000
```

Lo stato prossimo è LDI2. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
------------------- LDI1 LDI2 00000000000000000000100
```

Lo stato prossimo è LDI3. L'output è: R/W=0 (consente la lettura), MEM_EN=1 (permette di accedere alla memoria).

```
------------------- LDI2 LDI3 00000000000000010000000
```

Se R=1, lo stato prossimo è LDI4 e l'output è: LD_MDR=01 (seleziona il valore proveniente dalla memoria).
Altrimenti se R=0, lo stato prossimo è LDI3 e i bit dell'output sono posti a 0.

```
------------------1 LDI3 LDI4 00000000000000000001000
------------------0 LDI3 LDI3 00000000000000000000000
```

Lo stato prossimo è LDI5. L'output è: GATE=11 (inserisce nel registro BUS il valore di MDR).

```
------------------- LDI4 LDI5 11000000000000000000000
```

Lo stato prossimo è LDI6. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
------------------- LDI5 LDI6 00000000000000000000100
```

Lo stato prossimo è LDI7. L'output è: R/W=0 (consente la lettura), MEM_EN=1 (permette di accedere alla memoria).

```
------------------- LDI6 LDI7 00000000000000010000000
```

Se R=1, lo stato prossimo è LDI8 e l'output è: LD_MDR=01 (seleziona il valore proveniente dalla memoria).
Altrimenti se R=0, lo stato prossimo è LDI7 e i bit dell'output sono posti a 0.

```
------------------1 LDI7 LDI8 00000000000000000001000
------------------0 LDI7 LDI7 00000000000000000000000
```

Lo stato prossimo è LDI9. L'output è: GATE=11 (inserisce nel registro BUS il valore di MDR).

```
------------------- LDI8 LDI9 11000000000000000000000
```

Lo stato prossimo è LDI10. L'output è: LD_CC=1 (permette di controllare se il valore del bus è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
------------------- LDI9 LDI10 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
------------------- LDI10 F0 00000000000000000000000
```

LDR

L'istruzione LDR è una lettura da memoria con registro base. Ha come opcode **0110**. L'esecuzione è la seguente: si calcola l'indirizzo da leggere sommando a un registro base l'offset specificato nei bit meno significativi dell'istruzione, il valore letto da memoria sarà poi salvato in un registro. Si assegnano quindi i codici di condizione. Il risultato consiste di 6 stati.

```
DR = mem[ BaseR + SEXT5 ];
```

Lo stato prossimo è LDR0 e l'output è: GATE= 01 (seleziona il valore di MARMUX), ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=01 (seleziona SEXT5), SMARMUX=1 (seleziona il risultato di ADD).

```
0110---------------- DECODE LDR0 01010110000000000000000
```

Lo stato prossimo è LDR1. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- LDR0 LDR1 00000000000000000000100
```

Lo stato prossimo è LDR2. L'output è: R/W=0 (consente la lettura), MEM_EN=1 (permette di accedere alla memoria).

```
-------------------- LDR1 LDR2 00000000000000010000000
```

Se R=1, lo stato prossimo è LDR3 e l'output è: LD_MDR=01 (seleziona il valore proveniente dalla memoria).
Altrimenti se R=0, lo stato prossimo è LDR2 e i bit dell'output sono posti a 0.

```
-------------------1 LDR2 LDR3 00000000000000000001000
-------------------0 LDR2 LDR2 00000000000000000000000
```

Lo stato prossimo è LDR4. L'output è: GATE=11 (inserisce nel BUS il valore di MDR).

```
-------------------- LDR3 LDR4 11000000000000000000000
```

Lo stato prossimo è LDR5. L'output è: LD_CC=1 (permette di controllare se il valore del bus è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
-------------------- LDR4 LDR5 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- LDR5 F0 00000000000000000000000
```

LEA

L'istruzione LEA effettua il caricamento di un indirizzo: andrà a scrivere in un registro il valore del PC incrementato di un certo offset (specificato nei bit meno significativi dell'istruzione). Ha come opcode **1110**. Il risultato consiste di 3 stati.

```
DR = PC  +  SEXT8;
```

Lo stato prossimo è LEA0. L'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8) , SMARMUX=1 (seleziona il risultato di ADD).

```
1110---------------- DECODE LEA0 00001010000000000000000
```

Lo stato prossimo è LEA1 e l'output è: GATE= 01 (seleziona il valore di MARMUX).

```
-------------------- LEA0 LEA1 01000000000000000000000
```

Lo stato prossimo è LEA2 e l'output è: LD_REG=1 (permette di accedere al banco di registri).

```
-------------------- LEA1 LEA2 00000000001000000000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- LEA2 F0 00000000000000000000000
```

NOT

L'istruzione calcola il NOT bit a bit del valore salvato in un registro. Ha come opcode **1001**. Gli stati totali sono 2.

```
DR = NOT( SR ) ;
```

Lo stato prossimo è NOT0. L'output è: GATE=10 (mette il risultato della ALU in BUS), ALUK=010 (esegue l'operazione logica NOT all'interno dalla ALU).

```
1001---------------- DECODE NOT0 10000000000010000000000
```

Lo stato prossimo è NOT1. L'output è: LD_CC=1 (permette di controllare se il valore del bus è positivo, negativo o nullo e di inserire i risultati nei rispettivi registri), LD_REG=1 (permette di accedere al banco di registri).

```
-------------------- NOT0 NOT1 00000000001000100000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- NOT1 F0 00000000000000000000000
```

ST

ST è la prima delle istruzioni usate per scrivere in memoria. Ha come opcode **1001**. L'esecuzione è la seguente: si calcola l'indirizzo su cui scrivere sommando al PC l'offset contenuto nei bit meno significativi dell'istruzione, quindi si recupera il valore da scrivere da uno dei registri e si fa partire l'operazione di scrittura. Gli stati totali sono 5.

```
mem[ PC + SEXT8 ] = SR;
```

Lo stato prossimo è ST0. L'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8), SMARMUX=1 (seleziona il risultato di ADD), GATE=01 (seleziona il valore di MARMUX).

```
0011---------------- DECODE ST0 01001010000000000000000
```

Lo stato prossimo è ST1. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- ST0 ST1 00000000000000000000100
```

Lo stato prossimo è ST2. L'output è: ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=11 (seleziona 0), SMARMUX=1 (seleziona il risultato di ADD), GATE=01 (seleziona il valore di MARMUX).

```
-------------------- ST1 ST2 01011110000000000000001
```

Lo stato prossimo è ST3. L'output è: LD_MDR=10 (seleziona il valore proveniente dal BUS).

```
-------------------- ST2 ST3 00000000000000000010000
```

Lo stato prossimo è ST4. L'output è: R/W=1 (consente la scrittura), MEM_EN=1 (permette di accedere alla memoria).

```
-------------------- ST3 ST4 00000000000000011000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- ST4 F0 00000000000000000000000
```

STI

È l'istruzione di scrittura indiretta in memoria. Ha come opcode **1011**. L'esecuzione è la seguente: si somma al PC l'offset contenuto nei bit meno significativi dell'istruzione. La locazione di memoria a tale indirizzo contiene l'indirizzo della locazione su cui scrivere, quindi si recupera il valore da scrivere da uno dei registri e si fa partire l'operazione di scrittura. Gli stati totali sono 8.

```
mem [ mem [ PC + SEXT8 ] ]  = SR ;
```

Lo stato prossimo è ST0. L'output è: ADDR1MUX=0 (seleziona il valore di PC), ADDR2MUX=01 (seleziona SEXT8) , SMARMUX=1 (seleziona il risultato di ADD), GATE=01 (seleziona il valore di MARMUX).

```
1011---------------- DECODE STI0 01001010000000000000000
```

Lo stato prossimo è STI1. L'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- STI0 STI1 00000000000000000000100
```

Lo stato prossimo è STI2 e l'output è: R/W=0 (consente la lettura), MEM_EN (permette di accedere alla memoria).

```
-------------------- STI1 STI2 00000000000000010000000
```

Se R=1 lo stato prossimo è STI3 e l'output è LD_MDR=01 (per inserire in MDR il valore riportato dalla memoria).
Se R=0 lo stato prossimo è STI2 e tutti i bit in uscita sono posti a 0.

```
-------------------1 STI2 STI3 00000000000000000001000
-------------------0 STI2 STI2 00000000000000000000000
```

Lo stato prossimo è STI4 e l'output è: GATE=11 (seleziona il valore di MDR).

```
-------------------- STI3 STI4 11000000000000000000000
```

Lo stato prossimo è STI5 e l'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- STI4 STI5 00000000000000000000100
```

Lo stato prossimo è STI6 e l'output è: ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=11 (seleziona 0), SMARMUX=1 (seleziona il risultato di ADD),

GATE=01 (seleziona il valore di MARMUX), LD_MDR=10 (per inserire in MDR il valore del BUS).

```
-------------------- STI5 STI6 01011100000000000010001
```

Lo stato prossimo è STI7 e l'output è: R/W=1 (consente la scrittura), MEM_EN (permette di accedere alla memoria).

```
-------------------- STI6 STI7 00000000000000011000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- STI7 F0 00000000000000000000000
```

STR

È l'istruzione di scrittura in memoria con registro base. Ha come opcode **0111**. L'esecuzione è la seguente: per ottenere l'indirizzo di memoria su cui scrivere si somma al valore di un registro l'offset contenuto nei bit meno significativi dell'istruzione, quindi si recupera il valore da scrivere da un secondo registro e si fa partire l'operazione di scrittura. Il risultato consiste di 5 stati.

```
mem [ BaseR + SEXT8 ] = SR;
```

Lo stato prossimo è STR0 e l'output è: GATE=01 (seleziona il valore di MARMUX), ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=10 (seleziona SEXT5), SMARMUX=1 (seleziona il risultato di ADD).

```
0111---------------- DECODE STR0 01010110000000000000000
```

Lo stato prossimo è STR1 e l'output è: LD_MAR=1 (permette di memorizzare il valore del BUS in MAR).

```
-------------------- STR0 STR1 00000000000000000000100
```

Lo stato prossimo è STR2 e l'output è: GATE=01 (seleziona il valore di MARMUX), ADDR1MUX=1 (seleziona il valore del registro letto), ADDR2MUX=11 (seleziona 0), SMARMUX=1 (seleziona il risultato di ADD).

```
-------------------- STR1 STR2 01011110000000000000001
```

Lo stato prossimo è STR3 e l'output è: GATE=01 (seleziona il valore di MARMUX), LD_MDR=10 (per inserire in MDR il valore del BUS).

```
-------------------- STR2 STR3 01000000000000000010000
```

Lo stato prossimo è STR4 e l'output è: R/W=1 (consente la scrittura), MEM_EN (permette di accedere alla memoria).

```
-------------------- STR3 STR4 00000000000000011000000
```

Per qualsiasi valore in ingresso tutti i bit hanno il valore 0 lo stato prossimo è F0.

```
-------------------- STR4 F0 00000000000000000000000
```

TRAP e reserved

Agli opcode **1000** e **1101** non è assegnata alcuna istruzione. All'opcode **1111** è invece assegnata TRAP, che indica un errore all'interno dell'esecuzione. In presenza di questi tre casi lo stato prossimo è INIZIO e non F0 perché occorre inserire un nuovo programma e tutti i bit dell'output sono posti a 0.

```
1000---------------- DECODE INIZIO 00000000000000000000000
1111---------------- DECODE INIZIO 00000000000000000000000
1101---------------- DECODE INIZIO 00000000000000000000000
```

Esecuzione di un programma

Un programma consiste di una serie di istruzioni, che vanno eseguite per raggiungere il risultato richiesto. Perché un programma venga eseguito è necessario che le istruzioni di cui è composto vengano caricate in memoria. È anche probabile che il programma richieda operazioni tra dati contenuti nei registri, è quindi necessario caricare i valori iniziali all'interno di ciascun di questi.
Per inserire un programma all'interno del codice bisogna rispettare i seguenti passi:

1. GENERARE LE ISTRUZIONI: scrivere un modello `.blif` che generi il codice binario corrispondente all'istruzione;
2. INSERIRE LE ISTRUZIONI IN MEMORIA: le istruzioni create vengono inserite all'interno delle celle di memoria collegando i valori a VAR nel file `memory.blif`.
3. INSERIRE I DATI ALL'INTERNO DEI REGISTRI: vengono create anche qui delle variabili con il valore desiderato che poi verranno inserite all'interno del registro richiesto nel file `regs.blif`.

Ovviamente il programma non può essere inserito ad ogni ciclo di clock, perché si andrebbero a sovrascrivere dei valori contenuti nelle locazioni di memoria o nei registri che rappresentavano il risultato di un'istruzione, necessari quindi per l'esecuzione del programma. È stato creato così l'input RESET che viene inserito sia nella memoria che nel banco di registri e permette di lasciare che il valore delle variabili entri nel registro o nella cella di memoria corrispondente solo quando è posto a 1.
Nella FSM questo bit viene alzato solo nello stato INIZIO, corrispondente quindi allo stato in cui si caricano i dati del programma.

Esempio

Si vuole implementare in LC3 un programma con il seguente pseudo-codice:

```
int gf (int x){
          int f, h;
          f = x;
          do {
               f--;
               h = x;
               do {
                    h = h - f;
                    if (h = = 0) return (f);
               } while (h > 0);
          } while (1)
}
```

Di seguito si riporta una simulazione di funzionamento ponendo x a 6.

```
x = 6
f = x = 6
     (1° while)
     f-- => f = 5
     h = x = 6

          (2° while)
          h = h - f = 6 - 5 = 1
          h ≠ 0
          h = h - f = 1 - 5 = -4
          h < 0 => esco dal 2° while

     f-- => f = 4
     h = x = 6

          h = h - f = 6 - 4 = 2
          h ≠ 0
          h = h - f = 2 - 4 = -2
          h < 0 => esco dal 2° while

     f-- => f = 3
     h = x = 6

          h = h - f = 6 - 3 = 3
          h ≠ 0
          h = h - f = 3 - 3 = 0
          h = 0 => esco dal 2° while

     esco dal 1° while
return f = 3
```

L'implementazione di questo programma in codice binario prevede dieci istruzioni da salvare in memoria a partire dall'indirizzo 0. Per ogni istruzione è proposto l'indirizzo, cosa deve fare, istruzione in assembler e istruzione in binario (che è quella che verrà salvata in memoria).
I registri sono usati in questo modo:

- R2 contiene l'indirizzo di memoria contenente il valore di x (in questo caso deve essere inizializzato a 12);
- R3, R4 e R5 vengono usati per salvare i valori intermedi rispettivamente di f, h e –f durante il calcolo.

Indirizzo	Funzionamento	Assembler	Binario
0	Legge da memoria il valore di f, che è all'indirizzo contenuto in R2 (cioè 12 – R2 quindi dovrà valere 12), e lo salva in R3 `R3 = mem(R2) = f`	LDR R3 R2 000000	0110011010000000
1	Si somma -1 a R3 per calcolare f--; il risultato si salva in R3 `R3 = R3+(-1) = f--`	ADD R3 R3 1 11111	0001011011111111
2	Viene letto da memoria il valore di h, cioè 6, che viene poi salvato in R4 `R4 = mem(R2) = h`	LDR R4 R2 000000	0110100010000000
3	Viene negato R3 (cioè si calcola -f) e si salva il risultato in R5 `R5 = -R3`	NOT R5 R3 1 11111	1001101011111111
4	Viene sommato 1 a R5 (per calcolare correttamente -f) `R5 = R5+1 = -f`	ADD R5 R5 1 00001	0001101101100001
5	Viene sommato R4 a R5 (h+(-f)) e salvato in R4 `R4 = R4+R5 = h-f`	ADD R4 R4 000 R5	0001100100000101
6	Se è 0, salta all'istruzione 9	BR 010 000000010	0000010000000010

	`PC = PC + 2`		
7	Se è positivo, salta all'istruzione 5 `PC = PC - 3`	BR 001 111111101	0000001111111101
8	Se è negativo, salta all'istruzione 1 `PC = PC - 8`	BR 111 111111000	0000111111111000
9	Il risultato viene scritto nella locazione di memoria $(2^8) + 1$ `mem(R2+1) = R3`	STR R3 R2 000001	0111011010000001

Quindi alla fine si hanno:

- nove istruzioni da mettere in memoria. Quindi le locazioni di memoria dalla 0 alla 9 contengono le istruzioni scritte sopra;
- il registro R2 deve contenere 12, cioè l'indirizzo della locazione di memoria che conterrà il valore di x;
- la locazione di memoria con indirizzo 12 deve contenere il valore di x cioè 6.

Alla fine dell'esecuzione si troverà il valore 3 nella locazione di memoria 13. La gestione della memoria quindi è questa:

0	LDR
1	ADD
2	LDR
3	NOT
4	ADD
5	ADD
6	BR
7	BR
8	BR
9	STR
10	(non usata)
11	(non usata)
12	X = 6
13	Risultato = 3
14	(non usata)

GENERARE LE ISTRUZIONI

Le istruzioni nel codice vengono definite come variabili. Ad esempio, la prima istruzione di questo programma è LDR R3 R2 000000 che in codice binario corrisponde a 0110011010000000.

```
.model LDRA
.outputs A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1
A0
.names A15
.names A14
1
.names A13
1
.names A12
.names A11
.names A10
1
.names A9
1
.names A8
.names A7
1
.names A6
.names A5
.names A4
.names A3
.names A2
.names A1
.names A0
.end
```

Codice di `instrldra.blif`

INSERIRE LE ISTRUZIONI IN MEMORIA

Bisogna collegare VAR a ciascuna delle istruzioni come spiegato precedentemente.

```
.subckt LDRA A15=LDRA15 A14=LDRA14 A13=LDRA13 A12=LDRA12
A11=LDRA11 A10=LDRA10 A9=LDRA9 A8=LDRA8 A7=LDRA7 A6=LDRA6
A5=LDRA5 A4=LDRA4 A3=LDRA3 A2=LDRA2 A1=LDRA1 A0=LDRA0

.subckt CELL D=S00 RESET=RESET MDR15=MDR15 MDR14=MDR14
MDR13=MDR13 MDR12=MDR12 MDR11=MDR11 MDR10=MDR10 MDR9=MDR9
MDR8=MDR8 MDR7=MDR7 MDR6=MDR6 MDR5=MDR5 MDR4=MDR4 MDR3=MDR3
MDR2=MDR2 MDR1=MDR1 MDR0=MDR0 VAR15=LDRA15 VAR14=LDRA14
VAR13=LDRA13 VAR12=LDRA12 VAR11=LDRA11 VAR10=LDRA10
VAR9=LDRA9 VAR8=LDRA8 VAR7=LDRA7 VAR6=LDRA6 VAR5=LDRA5
```

```
VAR4=LDRA4 VAR3=LDRA3 VAR2=LDRA2 VAR1=LDRA1 VAR0=LDRA0
MEM15=MEM015 MEM14=MEM014 MEM13=MEM013 MEM12=MEM012
MEM11=MEM011 MEM10=MEM010 MEM9=MEM09 MEM8=MEM08 MEM7=MEM07
MEM6=MEM06 MEM5=MEM05 MEM4=MEM04 MEM3=MEM03 MEM2=MEM02
MEM1=MEM01 MEM0=MEM00
```

Questo pezzo di codice va inserito in `memory.blif`

INSERIRE I DATI NEI REGISTRI

Il registro R2 deve contenere 12, cioè l'indirizzo della locazione di memoria che conterrà il valore di x. Si crea quindi la variabile 12 in `twelve.blif`

```
.model TWELVE
.outputs A15 A14 A13 A12 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1
A0
.names A15
.names A14
.names A13
.names A12
.names A11
.names A10
.names A9
.names A8
.names A7
.names A6
.names A5
.names A4
.names A3
1
.names A2
1
.names A1
.names A0
.end
```

All'interno di `regs.blif` si richiama la variabile con `.subckt` e la si collega al primo MUX di scrittura.

```
.subckt TWELVE A15=TWELVE15 A14=TWELVE14 A13=TWELVE13
A12=TWELVE12 A11=TWELVE11 A10=TWELVE10 A9=TWELVE9
A8=TWELVE8 A7=TWELVE7 A6=TWELVE6 A5=TWELVE5 A4=TWELVE4
A3=TWELVE3 A2=TWELVE2 A1=TWELVE1 A0=TWELVE0
```

```
.subckt MUX1S16 S=RESET A15=TWELVE15 A14=TWELVE14
A13=TWELVE13 A12=TWELVE12 A11=TWELVE11 A10=TWELVE10
A9=TWELVE9 A8=TWELVE8 A7=TWELVE7 A6=TWELVE6 A5=TWELVE5
A4=TWELVE4 A3=TWELVE3 A2=TWELVE2 A1=TWELVE1 A0=TWELVE0
B15=R215 B14=R214 B13=R213 B12=R212 B11=R211 B10=R210
```

```
B9=R29 B8=R28 B7=R27 B6=R26 B5=R25 B4=R24 B3=R23 B2=R22
B1=R21 B0=R20 O15=RR215 O14=RR214 O13=RR213 O12=RR212
O11=RR211 O10=RR210 O9=RR29 O8=RR28 O7=RR27 O6=RR26 O5=RR25
O4=RR24 O3=RR23 O2=RR22 O1=RR21 O0=RR20
```

A questo punto bisogna solo simulare il progetto per controllarne il corretto funzionamento. Dal terminale si apre SIS e si eseguono i seguenti comandi:

```
read_blif lc3.blif
source execute.script
```

`Execute.blif` è uno script che ci permette di non dare il comando simulate ogni volta. Durante il debugging per facilitare il controllo sono stati posti in uscita i valori di:

```
BUS3 BUS2 BUS1 BUS0 PC0 MAR0 IR15 IR14 IR13 IR12 IR11 IR10
IR9 IR8 IR7 IR6 IR5 IR4 IR3 IR2 IR1 IR0 N Z P MEM1315
MEM1314 MEM1313 MEM1312 MEM1311 MEM1310 MEM139 MEM138
MEM137 MEM136 MEM135 MEM134 MEM133 MEM132 MEM131 MEM130
R215 R214 R213 R212 R211 R210 R29 R28 R27 R26 R25 R24 R23
R22 R21 R20 R315 R314 R313 R312 R311 R310 R39 R38 R37 R36
R35 R34 R33 R32 R31 R30 R415 R414 R413 R412 R411 R410 R49
R48 R47 R46 R45 R44 R43 R42 R41 R40 R515 R514 R513 R512
R511 R510 R59 R58 R57 R56 R55 R54 R53 R52 R51 R50
```

Il risultato del programma è contenuto in MEM13.
Alla fine della simulazione l'output è il seguente:

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di IR è 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 corrispondente all'istruzione STR

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di R2 è 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0. Infatti il valore di R2 è stato inizializzato a 12 e non è mai stato toccato durante il programma.

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di R3 è 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1. Infatti il valore di R3 è quello che ci dice il risultato e che alla fine verrà messo in MEM13.

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di R4 è 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0. È necessario che il suo valore sia pari a 0 altrimenti non sarebbe possibile saltare all'istruzione STR.

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di R5 è 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1. R5 deve valere -3 perché due istruzioni prima R4 valeva 3: sono stati sommati questi due valori e il risultato è stato messo in R4, mentre R5 non è stato modificato.

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```

Il valore di MEM13 è 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1. Questo era il risultato atteso.

```
Network simulation:
Outputs: 1 0 1 0 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
```