|  |
| --- |
| **Spazio di I/O**   * Spazio di 64K celle (dette porte), non tutte implementate (l'implementazione è a cura delle interfacce montate sul calcolatore, poche). * **Un’interfaccia** (qua a lato) si presenta come una memoria RAM: ho un piedino di select /s prodotto da una maschera. * Le variabili /ior e /iow hanno la stessa funzione di /mr e /mw, rispettivamente. Sono identiche ma non uguali. * Le variabili che non vanno nella maschera finiscono in ingresso come indirizzi interni se l’interfaccia presenta più di una porta. Nel nostro esempio ne ha due, quindi ci servirà una singola variabile logica. * **Osservazioni sulle interfacce dal lato bus**:   + In una RAM si può leggere e scrivere in una qualunque locazione. Un’interfaccia, invece, può supportare solo operazioni di lettura o solo operazioni di scrittura. Chiaramente uno dei due fili di comando, in questi casi, risulta superfluo. Nella maggior parte dei casi le interfacce presenteranno porte di entrambi i tipi.   Struttura di un'interfaccia con tavola di verità per ripassare   * + Se un’interfaccia presenta una sola porta allora non ho bisogno delle variabili di indirizzo (mi basta solo il select) * **Osservazioni sulle interfacce dal lato dispositivo**:   + Gli ingressi e le uscite variano da interfaccia a interfaccia e verranno descritti più avanti.   + **A cosa ci serve avere delle interfacce?**     - I dispositivi hanno velocità molto diverse tra loro, e spesso sono più lenti del processore (visto che si interfacciano col mondo esterno). Se io collegassi i dispositivi direttamente al bus il processore dovrebbe conoscere le proprietà del dispositivo cosa insana realizzare i processori tenendo conto dei dispositivi esistenti, che cambiano nel tempo in modo rapido)     - Le modalità di trasferimento dei dati sono molto diverse. Alcuni dispositivi trasferiscono un bit alla volta, altri gruppi di bit.   L’interfaccia permette di avere temporizzazioni omogenee e trasferimento di dati omogeneo.     * Vediamo il nostro spazio di I/O, con due interfacce:   + Entrambe hanno due porte (lo vediamo dalla variabile )   + Entrambe sono interfacce sia di ingresso che di uscita.   + Abbiamo 16 fili (non 24) che entrano nello spazio di I/O. 15 fili vanno nella maschera per generare il select.   + Nell’esempio supponiamo di avere le porte della prima interfaccia agli offset ‘H03C8, ‘H03C9, e le porte della seconda interfaccia agli offset ‘H0060, ‘H0064. La maschera ci permette di generare i valori di /s1 ed /s2   + **Attenzione**: non per forza la cifra meno significativa è quella utilizzata per distinguer le porte. Nella seconda interfaccia utilizziamo la terza cifra meno significativa per indicare la porta desiderata!   + ‘H03C8: 0000 0011 1100 1000   + ‘H03C9: 0000 0011 1100 1001   Cifre significative con cui identifico la porta dell’interfaccia   * + ‘H0060: 0000 0000 0110 0000   + ‘H0064: 0000 0000 0110 0100 |
| **Processore**  Il processore è una RSS sincronizzata che presenta un certo numero di registri. Abbiamo:   * Registri a supporto delle uscite, quindi:   + D7\_D0, per i fili di dati   + A23\_A0, per i fili di indirizzo   + /MR e /MW, per interagire con la memoria principale   + /IOR e /IOW, per interagire con il sottosistema di I/O * Un registro DIR per gestire la porta tristate per i fili di dati. Se DIR è uguale a zero la porta tristate è in alta impedenza, altrimenti è in conduzione. * Il registro STAR con cui indichiamo lo stato interno della rete. * Un registro dei flag, che conterrà flag come CF, SF, ZF, OF. * I registri accumulatori AH e AL * Il registro puntatore DP (indirizzamento di memoria indiretto) * Il registro puntatore SP per la pila * Il registro puntatore IP, per ricordarsi ogni volta l’istruzione successiva da eseguire. In alcuni casi potrebbe essere incrementato più volte (dipende dalla dimensione dell’istruzione, che può essere superiore al singolo byte). * Cinque registri a supporto delle operazioni di lettura e scrittura:   + APP0, APP1, APP2, APP3, che contengono il valore letto o da scrivere   + NUMLOC, registro contatore utilizzato nelle operazioni di lettura e scrittura (per capire quando abbiamo finito). * Registri a supporto dell’esecuzione delle istruzioni:   + OPCODE, registro contenente gli 8 bit con formato e istruzione.   + SOURCE e DEST\_ADDR, registri contenenti dati sugli operandi. Non è detto che siano usati sempre. Ricordiamoci che del sorgente ci interessa il contenuto, mentre del destinatario l’indirizzo. Più avanti è presente un pdf di Corsini che spiega i vari casi relativamente ai formati.   + MJR, registro per i salti (spiegato qualche lezione fa) * **Fasi del processore**:     *Fase di fetch*   * + **Fase di reset**: inizializzazione del contenuto dei registri (F, IP, ma anche altri).     - IP <- ‘HFF0000     - F <- 0     - Tutti i registri che reggono variabili di comando dovranno essere inizializzati in modo coerente: in particolare tutti gli attivi bassi dovranno assumere come valore 1.     - Dobbiamo anche mettere in alta impedenza la porta tri-state. Segue DIR inizializzato a 0, cambiamo il suo valore solo quando dovremo scrivere sul bus (ricordiamoci che quando la porta tristate del processore si comporta in un modo quella presente dall’altra parte – per esempio quella in memoria principale – dovrà comportarsi in modo opposto).     - STAR verrà inizializzato col primo stato della fase di fetch.   + **Fase di fetch**:     - Preleva un byte dalla memoria, all’indirizzo IP     - Incrementa IP (punta al byte successivo)     - Controlla che il bit appena letto corrisponda all’OPCODE di una delle istruzioni note.     - Il contenuto del byte letto deve essere copiato nel registro OPCODE. Inoltre dobbiamo valutare il formato dell’istruzione (tre bit più significativi). Valutare il formato significa decidere cosa fare (per ogni formato abbiamo una certa procedura)       * Per alcuni formati dovremo incrementare IP opportunamente (in base al numero di bit da leggere)       * In alcuni formati dobbiamo procurarci l’indirizzo dell’operando destinatario, e inserirlo in DEST\_ADDR. Nel formato F3 l’indirizzo già sta in DP, mi basta copiarlo in DEST\_ADDR. Nei formati F6 e F7 devo andarlo a leggere in memoria: incremento di 3 il valore di IP, leggo 3 bit e li sposto nel processore.       * In altri formati, F0 ed F1, il processore non fa niente in fase di fetch.     - Ultima cosa è la lettura del contenuto di OPCODE (i cinque bit meno significativi): devo capire qual è l’istruzione da seguire.   + **Fase di esecuzione**     - Il processore esegue l’istruzione che ha decodificato.     - Torna nella fase di fetch a meno che non stia eseguendo l’istruzione di HLT.   + **Stato di blocco**: raggiunto nei seguenti casi:     - OPCODE non valido;     - istruzione HLT.   Non è possibile uscire da questo stato: l’unico modo è fare reset.   * **Letture e scritture in memoria e spazio di I/O**:   + Il processore legge in memoria durante la fase di fetch.   + Durante la fase di esecuzione il processore dovrà leggere e scrivere in memoria (RET, MOV) o nello spazio di I/O (IN, OUT)   + **Letture e scritture in memoria**:     - Ricordarsi della struttura della RAM statica (multiplexer, demultiplexer, variabili di comando, RC per le variabili di pilotaggio).     - Ricordarsi la temporizzazione delle RAM statiche in lettura e scrittura     - **Cosa significano queste cose dal punto di vista del processore?**   Vediamo il ciclo di lettura:     * + - * Nel primo stato imposto l’indirizzo uguale a qualcosa, pongo DIR uguale a 0 (finchè non ho dati sensati rimane a 0). Pongo MR\_ a 0 e indico il passaggio alla stato successivo (micro-istruzione a singolo step).       * Il secondo stato è detto stato di wait, utilizzato per dare alla memoria tempo per rispondere. D’ora in poi lo daremo per scontato, in modo tale da ottenere descrizioni più semplici.       * Il terzo stato è quello in cui abbiamo dati affidabili: li campioniamo in quale registro e riportiamo MR\_ a 1.       * **Letture successive**: nell’ultimo stato non alzo MR\_ e pongo un nuovo indirizzo da leggere.       * **Domanda**: posso alzare DIR a 1 nell’ultimo stato assieme a MR\_? No, perché le tristate della memoria sono ancora in conduzione e ci mettono più tempo ad alzarsi rispetto alle tristate del processore. Se alzo insieme i due valori creo, per poco tempo, una situazione in cui entrambe le tristate sono attive.   Vediamo il ciclo di scrittura:     * + - * Operazione distruttiva. Sappiamo che dobbiamo attendere la stabilizzazione di /s e degli indirizzi prima di portare giù /mw.       * I dati, ricordiamo, devono essere corretti a cavallo del fronte di salita.       * Primo stato: setto l’indirizzo, pongo i dati che voglio salvare in ingresso, alzo DIR e indico il passaggio allo stato successivo.       * Secondo e terzo stato: stati di attesa.       * Terzo stato: abbasso DIR, abbiamo scritto.       * **Posso scrivere in D7\_D0 nel terzo stato?** No, altrimenti non sarebbe rispettata la regola di temporizzazione principale (dati costanti a cavallo del fronte di salita).       * **Posso portare DIR a 0 direttamente nel secondo stato?** No, altrimenti i dati non rimangono stabili.       * **Posso porre un nuovo indirizzo nel terzo stato?** No, devo aspettare il clock successivo.   + **Letture nello spazio di I/O**:   Vediamo la lettura     * + - Le letture sono simili, MA NON UGUALI, alle lettura in memoria.     - Differenza: MR\_ può essere portato a 0 nello stesso stato in cui setto gli indirizzi. La lettura in memoria è un’operazione non distruttiva, se gli indirizzi ballano non è un problema. Nello spazio di I/O anche la lettura può essere distruttiva: se io leggo da un’interfaccia un dato questa si regola col suo dispositivo per svolgere operazioni di scrittura. Quindi posso avere dati appena letti subito sovrascritti.     - Nell’ultimo stato posso assegnare il valore in uscita (d7\_d0) a qualche registro. Alzo IOR\_ e sono obbligato a farlo: in caso di nuove lettura siamo obbligati a finire un ciclo e aprirne un altro.   Vediamo la scrittura:     * + - Nel ciclo di scrittura in memoria l’assegnamento a D7\_D0 poteva essere spostato nello stato successivo. Nella scrittura nello spazio di I/O dobbiamo avere i dati già pronti a cavallo del fronte di discesa di IOW\_: questo perché alcune interfacce memorizzano sul fronte di discesa, e non su quello di salita. Segue che D7\_D0 deve essere assegnato per forza nel primo stato e che non possa essere spostato. * **Accessi per più di un byte alla memoria**:   + Il processore deve poter leggere operandi a 2 e 3 byte.   + Fa comodo strutturarsi di micro-sottoprogrammi di lettura/scrittura modulari. Questi sottoprogrammi possono essere riferiti ogni volta che dobbiamo leggere più di un byte.   + **Registri utilizzati**:     - Il registro MJR per salvare il micro-indirizzo di ritorno, cioè lo stato su cui devo tornare dopo aver eseguito il micro-sottoprogramma.     - I registri APP0, APP1, APP2, APP3 per salvare i byte letti o da scrivere.     - Il registro NUMLOC come contatore del numero di byte da leggere/scrivere.   + **Lettura**     - I micro-sottoprogrammi sono i seguenti:       * readB (1 byte)       * readW (2 byte)       * readM (3 byte)       * readL (4 byte, non lo utilizzeremo)     - I parametri in ingresso sono A23\_A0, che contiene il primo indirizzo in memoria (modificato dai micro-sottoprogrammi), DIR a zero e il valore di MJR (dobbiamo indicare da dove ricominciare dopo aver terminato l’esecuzione del micro-sottoprogramma).     - Codice del microprogramma:   // PER L’ESECUZIONE  *Chiamata di un micro-sottoprogramma*  *Dati in ingresso*  Sx: **begin** … A23\_A0<=un\_indirizzo; MJR<=Sx+1; STAR<=read**B**; **end**  Sx+1: **begin** … <utilizzo di APP0> **end**  *Utilizzo dei dati letti dal micro-sottoprogramma. Questi dati sono recuperati dai registri di supporto*  *Inizializzazione delle operazioni di lettura. Abbasso MR\_ visto che voglio svolgere l’operazione, indico il numero di locazioni da leggere in NUMLOCK e passo alla prima lettura*  // MICROSOTTOPROGRAMMA PER LETTURE IN MEMORIA  readB: **begin** MR\_<=0; DIR<=0; NUMLOC<=1; STAR<=read0; **end**  readW: **begin** MR\_<=0; DIR<=0; NUMLOC<=2; STAR<=read0; **end**  readM: **begin** MR\_<=0; DIR<=0; NUMLOC<=3; STAR<=read0; **end**  readL: **begin** MR\_<=0; DIR<=0; NUMLOC<=4; STAR<=read0; **end**  read0: **begin** APP0<=d7\_d0; A23\_A0<=A23\_A0+1; NUMLOC<=NUMLOC-1;  STAR<=(NUMLOC==1) ? read4 : read1; **end**  read1: **begin** APP1<=d7\_d0; A23\_A0<=A23\_A0+1; NUMLOC<=NUMLOC-1;  STAR<=(NUMLOC==1) ? read4 : read2; **end**  read2: **begin** APP2<=d7\_d0; A23\_A0<=A23\_A0+1; NUMLOC<=NUMLOC-1;  STAR<=(NUMLOC==1) ? read4 : read3; **end**  read3: **begin** APP3<=d7\_d0; A23\_A0<=A23\_A0+1; STAR<=read4; end  read4: **begin** MR\_<=1; STAR<=MJR; **end**  *Operazioni di lettura. Salvo il byte letto nell’apposito registro APPx. Dopo aver letto e spostato nel processore mi chiedo se ho svolto il numero di letture necessarie. Se ho finito abbaso MR\_ e indico come stato successivo quello posto all’inizio in MJR.*   * + **Scrittura**     - I micro-sottoprogrammi sono i seguenti:       * writeB (1 byte)       * writeW (2 byte)       * writeM (3 byte)   Ricordarsi a cosa serve DIR. Se la porta tristate ha 1 la “levetta” è alzata.   * + - * writeL (4 byte)     - I parametri in ingresso sono A23\_A0, che contiene il primo indirizzo in memoria su cui lavorare (verrà modificato dai micro-sottoprogrammi), DIR a 0 (ci pensa il sottoprogramma ad alzarlo e poi ad abbassarlo), APPj (j=0…3) con i byte da scrivere, MJR (come prima).     - Codice del microsottoprogramma:   *Solite cose fatte prima, cambiano solo gli ingressi e i micro-sottoprogrammi chiamati*  // PER L’ESECUZIONE  Sx: **begin** … APP1<=dato\_16\_bit[15:8]; APP0<=dato\_16\_bit[7:0];  A23\_A0<=un\_indirizzo; MJR<=Sx+1; STAR<=write**W**; **end**  Sx+1: **begin** … <prosecuzione del programma dopo la scrittura> **end**  // MICROSOTTOPROGRAMMA PER SCRITTURE IN MEMORIA  // *PIU’ NOIOSO, PER SVOLGERE OGNI SINGOLA SCRITTURA SERVE UN CICLO IN PIU*  *// Metto il primo indirizzo su cui scrivo, alzo DIR perché devo svolgere operazioni di lettura, indico il numero di scritture da fare, infine cambio stato per svolgere la prima scrittura.*  *// In ogni ciclo di scrittura prima abbasso MW\_, poi lo rialzo (ogni volta)*  writeB: **begin** D7\_D0<=APP0; *DIR<=1; NUMLOC<=1;* STAR<=write0; **end**  writeW: **begin** D7\_D0<=APP0; *DIR<=1; NUMLOC<=2;* STAR<=write0; **end**  writeM: **begin** D7\_D0<=APP0; *DIR<=1; NUMLOC<=3;* STAR<=write0; **end**  writeL: **begin** D7\_D0<=APP0; *DIR<=1; NUMLOC<=4;* STAR<=write0; **end**  write0: **begin** MW\_<=0; STAR<=write1; **end**  write1: **begin** MW\_<=1; STAR<=(NUMLOC==1)?write11:write2; **end**  write2: **begin** D7\_D0<=APP1; A23\_A0<=A23\_A0+1; NUMLOC<=NUMLOC-1; STAR<=write3; **end**  write3: **begin** MW\_<=0; STAR<=write4; **end**  write4: **begin** MW\_<=1; STAR<=(NUMLOC==1)?write11:write5; **end**  write5: **begin** D7\_D0<=APP2; A23\_A0<=A23\_A0+1; NUMLOC<=NUMLOC-1; STAR<=write6; **end**  Ricordare l'ordine delle cifre nei vari bit.  write6: **begin** MW\_<=0; STAR<=write7; **end**  write7: **begin** MW\_<=1; STAR<=(NUMLOC==1)?write11:write8; **end**  write8: **begin** D7\_D0<=APP3; A23\_A0<=A23\_A0+1; STAR<= write9; **end**  write9: **begin** MW\_<=0; STAR<= write10; **end**  write10: **begin** MW\_<=1; STAR<= write11; **end**  write11: **begin** *DIR<=0;* STAR<=MJR; **end**  **Ogni volta**:   * Indico l’indirizzo su cui voglio scrivere (nel caso della prima scrittura è indicato come parametro in ingresso) e i valori da porre sui fili di dati D7\_D0. * Dopo aver posto questi dati, E NON PRIMA, abbasso MW\_ a 0. * Al ciclo successivo rialzo e verifico se ho scritto il numero di byte richiesti. * Se ho finito vado all’ultimo stato del micro-sottoprogramma: riabbasso la levetta della porta tristate e pongo come stato successivo quello indicato in MJR all’inizio. * **Promemoria**: negli indirizzi più bassi si pongono le cifre meno significative. |
| **Descrizione in Verilog del processore** |
| Vediamo la descrizione in Verilog del processore. Nel corso della descrizione faremo uso di reti combinatorie in grado di semplificarci alcune operazioni. Non ci dedicheremo alla descrizione accurata di queste funzioni: sono estremamente noiose, ma non impossibili da scrivere per noi.  //-----------------------------------------------------------------  // DESCRIZIONE COMPLETA DEL PROCESSORE  //-----------------------------------------------------------------  module Processore(d7\_d0,a23\_a0,mr\_,mw\_,ior\_,iow\_,clock,reset\_);  input clock,reset\_;  **inout [7:0] d7\_d0;**  output [23:0] a23\_a0;  output mr\_,mw\_;  output ior\_,iow\_;  // REGISTRI OPERATIVI DI SUPPORTO ALLE VARIABILI DI USCITA E ALLE  // VARIABILI BIDIREZIONALI E CONNESSIONE DELLE VARIABILI AI REGISTRI  **reg** DIR;  *Definisco i registri di supporto alle variabili di uscita e definisco come valore di queste uscite quelli dei corrispondenti registri*  *Evidenziate in rosso due righe con sintassi nuova: con inout indichiamo fili che possono fungere sia come ingressi che come uscite, con ‘HZZ indichiamo l’alta impedenza (in questo caso se DIR == 0)*  **reg** [7:0] D7\_D0;  **reg** [23:0] A23\_A0;  **reg** MR\_,MW\_,IOR\_,IOW\_;  **assign** mr\_ = MR\_;  **assign** mw\_ = MW\_;  **assign** ior\_ = IOR\_;  **assign** iow\_ = IOW\_;  **assign** a23\_a0 = A23\_A0;  **assign d7\_d0=(DIR==1)?D7\_D0:'HZZ; //FORCHETTA**  *Definisco i registri a supporto delle operazioni di lettura e scrittura, oltre ai registri accumulatori che utilizziamo per gestire operazioni aritmetiche.*  // REGISTRI OPERATIVI INTERNI  *Al più lavoriamo su 4 byte, quindi bastano 3 bit.*  **reg** [2:0] NUMLOC;  **reg** [7:0] AL,AH,F,OPCODE,SOURCE,APP3,APP2,APP1,APP0;  *I registri a 24 bit contengono indirizzi*  **reg** [23:0] DP,IP,SP,DEST\_ADDR;  // REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI  **reg** [6:0] STAR,MJR;  *Con 7 bit rappresentiamo 86 stati interni diversi!*  **parameter** fetch0=0, .... write11=86;  // RETI COMBINATORIE NON STANDARD  **function** valid\_fetch;  *Prende in ingresso un byte e restituisce 1 se quel byte è l’OPCODE di un’istruzione nota, 0 altrimenti*  **input** [7:0] opcode;  ... ...  **endfunction**  **function** [6:0] first\_execution\_state;  *Prende in ingresso 7 bit, interpretato come OPCODE valido, restituisce la codifica del primo stato interno dell’esecuzione dell’istruzione relativa.*  **input** [7:0] opcode;  ... ...  **endfunction**  *Simulo lo ALU interna al processore. Interpreto i 3 byte passati come un OPCODE, un operando sorgente e un operando destinatario. Restituisco il risultato su 8 bit dell’elaborazione svolta. Posso porre come OPCODE add, sub, and, or …*  **function** [7:0] alu\_result;  **input** [7:0] opcode,operando1,operando2;  ... ...  **endfunction**  **function** [3:0] alu\_flag;  *Prendo in ingresso gli stessi byte di alu\_result e aggiorno in flag rendendoli consistenti. Restituisco 4 bit che consistono nei 4 flag significativi (quelli che ci interessano di più) del registro F.*  **input** [7:0] opcode,operando1,operando2;  ... ...  **endfunction**  **function** jmp\_condition;  *Prendo in ingresso il contenuto del registro OPCODE e quello del registro dei flag. Restituisco 1 se:*   * *La condizione di JMP è incondizionata* * *La condizione di JMP è condizionata e la condizione di salto risulta verificata (lettura dei flag per capire questo).*   **input** [7:0] opcode;  **input** [7:0] flag;  ... ...  **endfunction**  // ALTRI MNEMONICI  **parameter** [2:0] F0='B000,F1='B001,F2='B010,F3='B011, F4='B100,F5='B101,F6='B110,F7='B111;  *Formati possibili*  //----------------------------------------------------------------  // AL RESET\_ INIZIALE  **always** @(reset\_==0) #1 **begin** IP<='HFF0000; F<='H00; DIR<=0;  MR\_<=1; MW\_<=1; IOR\_<=1; IOW\_<=1; STAR<=fetch0; **end**  *Inizializzazione di registri importanti: primo indirizzo di istruzione, reset dei flag, porta tristate in alta impedenza, attivi bassi alzati, prima istruzione di fetch come stato successivo.*  //----------------------------------------------------------------  // ALL’ARRIVO DEI SEGNALI DI SINCRONIZZAZIONE  **always** @(posedge clock) **if** (reset\_==1) #3  **casex**(STAR)  //----------------------------------------------------------------  // FASE DI FETCH (CHIAMATA)  Ricordarsi le fasi:  *Operazione di lettura di un byte.*   * *LEGGI GLI OPCODE* * *PROCURATI GLI OPERANDI* * *ESECUZIONE DELL’OPERAZIONE CON OPCODE VALIDO*   // Pongo come indirizzo quello presente nell’IP e lo incremento (ricordare, esecuzione in parallelo, non in sequenza)  fetch0: **begin** A23\_A0<=IP; IP<=IP+1; MJR<=fetch1; STAR<=readB; **end**  fetch1: **begin** OPCODE<=APP0; STAR<=fetch2; **end**  fetch2: **begin**  *Istruzione di ritorno. Passiamo a fetch1 dopo la lettura.*  MJR<=(OPCODE[7:5]==F0)? fetchEnd:  (OPCODE[7:5]==F1)? fetchEnd:  (OPCODE[7:5]==F2)? fetchF2\_0:  *Porto il byte letto in OPCODE e passo allo stato successivo*  (OPCODE[7:5]==F3)? fetchF3\_0:  (OPCODE[7:5]==F4)? fetchF4\_0:  *Prendo le tre cifre più significative dell’OPCODE e verifico a quale formato corrispondono.*  (OPCODE[7:5]==F5)? fetchF5\_0:  (OPCODE[7:5]==F6)? fetchF6\_0:  /\* default \*/ fetchF7\_0;  *Se l’OPCODE è valido passo a fetch3, altrimenti ad nvi*  STAR<=(valid\_fetch(OPCODE)==1)? fetch3 : nvi;  *Nessun problema a eseguirle nello stesso stato, valid\_fetch non dipende da MJR.*  **end**  *// SALTO AL PRIMO PASSO SPECIFICO DELLA FASE DI FETCH (Lo abbiamo determinato prima)*  fetch3: **begin** STAR<=MJR; **end**      fetchF2\_0: **begin** A23\_A0<=DP; MJR<=fetchF2\_1; STAR<=readB; **end**  fetchF2\_1: **begin** SOURCE<=APP0; STAR<=fetchEnd; **end**    fetchF3\_0: **begin** DEST\_ADDR<=DP; STAR<=fetchEnd; **end**    fetchF4\_0: **begin** A23\_A0<=IP; IP<=IP+1; MJR<=fetchF4\_1; STAR<=readB; **end**  fetchF4\_1: **begin** SOURCE<=APP0; STAR<=fetchEnd; **end**    fetchF5\_0: **begin** A23\_A0<=IP; IP<=IP+3; MJR<=fetchF5\_1; STAR<=readM; **end**  fetchF5\_1: **begin** A23\_A0<={APP2,APP1,APP0}; MJR<=fetchF5\_2; STAR<=readB; **end**  fetchF5\_2: **begin** SOURCE<=APP0; STAR<=fetchEnd; **end**    fetchF6\_0: **begin** A23\_A0<=IP; IP<=IP+3; MJR<=fetchF6\_1; STAR<=readM; **end**  fetchF6\_1: **begin** DEST\_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; **end**    fetchF7\_0: **begin** A23\_A0<=IP; IP<=IP+3; MJR<=fetchF7\_1; STAR<=readM; **end**  fetchF7\_1: **begin** DEST\_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; **end**  //----------------------------------------------------------------  // TERMINAZIONE DELLA FASE DI CHIAMATA  // TERMINAZIONE CON BLOCCO PER ISTRUZIONE NON VALIDA  *// Eseguo se l’OPCODE non è valido (rivedere fetch2)*  nvi: **begin** STAR<=nvi; **end**  // TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE  // Individuo il primo passo da eseguire per la fase di esecuzione.  // Osservazione: perché il salto lo faccio un passo dopo? Ricordiamo che le istruzioni sono eseguite in parallelo, non in sequenza!  fetchEnd: **begin** MJR<=first\_execution\_state(OPCODE); STAR<=fetchEnd1; **end**  fetchEnd1: **begin** STAR<=MJR; **end**  **All’uscita dalla fase di fetch avrò**:   * L’OPCODE che contiene il codice operativo dell’istruzione * Se l’istruzione ha un operando sorgente immediato o in memoria questo è in SOURCE. * Se l’istruzione ha un operando destinatario in memoria il suo indirizzo sta in DEST\_ADDR. * IP è stato incrementato e punta alla prossima istruzione da prelevare (ESECUZIONE IN PARALLELO, NON IN SEQUENZA). |