Continuiamo l'analisi della descrizione Verilog con la fase di esecuzione. La cosa sarà semplice:

- parte della complessità è stata già gestita in fase di fetch;
- la maggior parte delle istruzioni complesse sono implementate tramite reti combinatorie, quindi si riducono a un semplice assegnamento a registro.

```
//------------- istruzione NOP ----------------
```
// Istruzione che non fa niente. Ho un passo in più, ritorno subito al primo step di fetch.
```
nop: begin STAR<=fetch0; end
```

```
//------------- istruzione HLT ----------------
```
// Loop infinito da cui si esce solo premendo reset
```
hlt: begin STAR<=hlt; end
```

```
//------------- istruzione MOV AL,AH -----------
```
// Si assegna al registro AH il contenuto del registro AL
```
ALtoAH: begin AH<=AL; STAR<=fetch0; end
```

```
//------------- istruzione MOV AH,AL ----------------
```
// Si assegna al registro AL il contenuto del registro AH
```
AHtoAL: begin AL<=AH; STAR<=fetch0; end
```

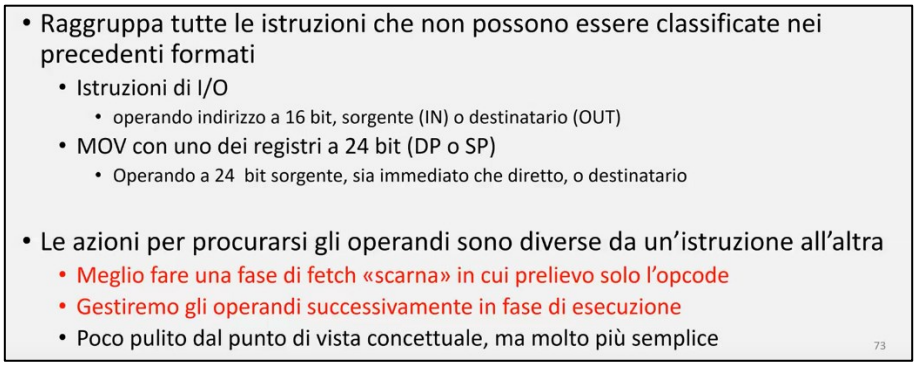
```
//------------- istruzione INC DP ----------------
```
// Incremento il contenuto del registro DP
```
incDP: begin DP<=DP+1; STAR<=fetch0; end
```

```
//------------- istruzioni MOV (DP),AL       -----------
                           MOV $operando,AL
                           MOV indirizzo,AL
```
// Assegno al registro AL il contenuto del registro SOURCE (ricordiamo quanto detto nella scorsa lezione)
```
ldAL: begin AL<=SOURCE; STAR<=fetch0; end
```

```
//------------- istruzioni MOV (DP),AH       ------------
                           MOV $operando,AH
                           MOV indirizzo,AH
```
// Assegno al registro AH il contenuto del registro SOURCE (uguale a prima, registro destinatario diverso)
```
ldAH: begin AH<=SOURCE; STAR<=fetch0; end
```

```
//------------- istruzioni MOV AL,(DP)       ---------
                           MOV AL,indirizzo
```
// Indirizzo dell'operando destinatario nel registro DEST_ADDR. Contrariamente a prima dobbiamo svolgere un'operazione di scrittura al di fuori del processore.
```
storeAL: begin A23_A0<=DEST_ADDR; APP0<=AL; MJR<=fetch0; STAR<=writeB; end
```

```
//------------- istruzioni MOV AH,(DP)       ----------
                           MOV AH,indirizzo
```
// Stessa cosa di prima, cambia solo il valore assegnato al registro APP0
```
storeAH: begin A23_A0<=DEST_ADDR; APP0<=AH; MJR<=fetch0; STAR<=writeB; end
```



```
//---------- istruzione MOV $operando,SP ---------
```
// L'operando SP è a 24bit. IP è stato incrementato di 1 quando arriviamo qua. Incremento di tre andando al primo bit che non mi interessa. Svolgo un'operazione di lettura readM (3 byte). Come indirizzo da leggere ho l'Instruction Pointer, come MJR lo stato successivo in cui utilizzo i valori letti. L'unione dei tre registri modificati con l'operazione di lettura consiste nel valore dell'SP (In ordine decrescente, ricordiamo dove stanno le cifre più significative e quelle meno significative).

```
ldSP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldSP1; STAR<=readM; end
ldSP1: begin SP<={APP2,APP1,APP0}; STAR<=fetch0; end

//---------- istruzione MOV $operando,DP --------
```
// Anche in questo caso abbiamo un operando (quello sorgente) a 24 bit. Facciamo le stesse cose di prima, cambia solo l'assegnamento a DP invece che ad SP.

```
ldimmDP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldimmDP1; STAR<=readM; end
ldimmDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------ istruzione MOV indirizzo,DP -------
```
// Devo leggere l'indirizzo, quindi svolgo una lettura di 3 byte. Dopo aver estratto l'indirizzo svolgo un'ulteriore operazione di lettura per leggere il contenuto dell'indirizzo. Pongo il contenuto dell'indirizzo come nuovo valore del registro DP.

```
lddirDP: begin A23_A0<=IP; IP<=IP+3; MJR<=lddirDP1; STAR<=readM; end
lddirDP1: begin A23_A0<={APP2,APP1,APP0}; MJR<=lddirDP2; STAR<=readM; end
lddirDP2: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------ istruzione MOV DP,indirizzo -----
```
// Mi devo procurare l'indirizzo (l'operando destinatario) con un'operazione di lettura su 3 byte. Successivamente svolgo un'operazione di scrittura sull'indirizzo trovato. Questa istruzione, suggeriva qualcuno, potrebbe essere accorpata ad F6: mi sbarazzo del primo step e nel secondo assegno DEST_ADDR invece di {APP2, APP1, APP0}

```
storeDP: begin A23_A0<=IP; IP<=IP+3; MJR<=storeDP1; STAR<=readM; end
storeDP1: begin A23_A0<={APP2,APP1,APP0}; {APP2,APP1,APP0}<=DP; MJR<=fetch0;
STAR<=writeM; end

//------------ istruzione IN offset,AL ---------
```
// Per prima cosa devo procurarmi l'offset: svolgiamo un'operazione di lettura su 2 byte a partire dall'IP. Dopo la lettura prendo i primi 16 bit (gli unici che mi interessano). A questo punto devo fare una lettura nello spazio di I/O

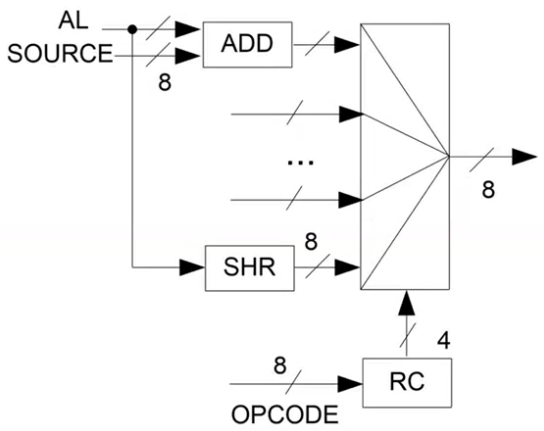
```
in: begin A23_A0<=IP; IP<=IP+2; MJR<=in1; STAR<=readW; end
```

// Preparo l'indirizzo, allo step successivo abbasso IOR_ (DOPO), nello step finale prendo i dati letti e li metto nel registro. Alzo IOR_ visto che ho finito l'operazione.

```
in1: begin A23_A0<={'H00,APP1,APP0}; STAR<=in2; end
in2: begin IOR_<=0; STAR<=in3; end
in3: begin AL<=d7_d0; IOR_<=1; STAR<=fetch0; end

//------------ istruzione OUT AL,offset ---------
```
// Leggo l'offset e svolgo operazione di lettura. Ricordarsi che non possiamo abbassare DIR e alzare IOW_ in contemporanea (altrimenti abbiamo problemi con la porta tristate). Il comando di scrittura nello spazio di I/O è il fronte di discesa.

```
out: begin A23_A0<=IP; IP<=IP+2; MJR<=out1; STAR<=readW; end
out1: begin A23_A0<={'H00,APP1,APP0}; D7_D0<=AL; DIR<=1; STAR<=out2; end
out2: begin IOW_<=0; STAR<=out3; end
out3: begin IOW_<=1; STAR<=out4; end
out4: begin DIR<=0; STAR<=fetch0; end

//------ istruzioni ADD (DP),AL -------
                    ADD $operando,AL
                    ADD indirizzo,AL
                    SUB (DP),AL
                    SUB $operando,AL
```

```
                    SUB indirizzo,AL
                    AND (DP),AL
                    AND $operando,AL
                    AND indirizzo,AL
                    OR (DP),AL
                    OR $operando,AL
                    OR indirizzo,AL
                    CMP (DP),AL
                    CMP $operando,AL
                    CMP indirizzo,AL
                    NOT AL
                    SHR AL
                    SHL AL
```



// Tutte le istruzioni elencate possono essere collassate nel seguente codice. Si assegna ad AL il risultato di un'operazione combinatoria dove indico OPCODE, SOURCE, AL. Dall'OPCODE capisco quale operazione dovrà essere svolta, gli altri due consistono negli operandi. Le operazioni possibili sono ADD, SUB, AND, OR, CMP, NOT, SHL, SHR: segue un multiplexer a otto vie con variabili di comando generate dall'OPCODE (Rete combinatoria dipendente dall'implementazione scelta dell'OPCODE).
Dobbiamo tener conto anche dell'aggiornamento dei flag: ricordiamo che il registro dei flag consiste in un insieme di 8 bit di cui i primi 4 importanti. Attraverso l'assegnamento non bloccante aggiorniamo i flag nelle posizioni 0,1,2,3 (OF,SF,ZF,CF): facciamo in questo modo visto che gli altri flag, in queste operazioni, sono insignificanti.

```
aluAL: begin
AL <= alu_result(OPCODE,SOURCE,AL);
F<={F[7:4], alu_flag(OPCODE,SOURCE,AL)};
STAR<=fetch0;
end

//----- istruzioni ADD (DP),AH --
                    ADD $operando,AH
                    ADD indirizzo,AH
                    SUB (DP),AH
                    SUB $operando,AH
                    SUB indirizzo,AH
                    AND (DP),AH
                    AND $operando,AH
                    AND indirizzo,AH
                    OR (DP),AH
                    OR $operando,AH
                    OR indirizzo,AH
                    CMP (DP),AH
                    CMP $operando,AH
                    CMP indirizzo,AH
                    NOT AH
                    SHL AH
                    SHR AH
```

// In modo del tutto speculare facciamo le stesse cose di prima con AH operando destinatario.

```
aluAH: begin
AH<=alu_result(OPCODE,SOURCE,AH);
F<={F[7:4],alu_flag(OPCODE,SOURCE,AH)};
STAR<=fetch0;
end

//----- istruzioni JMP indirizzo -----------
                    JA indirizzo
                    JAE indirizzo
                    JB indirizzo
                    JBE indirizzo
```

```
                    JC indirizzo
                    JE indirizzo
                    JG indirizzo
                    JGE indirizzo
                    JL indirizzo
                    JLE indirizzo
                    JNC indirizzo
                    JNE indirizzo
                    JNO indirizzo
                    JNS indirizzo
                    JNZ indirizzo
                    JS indirizzo
                    JO indirizzo
                    JZ indirizzo
```

// Tutte le istruzioni di salto consistono nell'impostare un nuovo valore dell'Instruction pointer. La fase di fetch mi ha portando l'indirizzo nel DEST_ADDR.
Attraverso l'OPCODE capisco se la condizione di JUMP è vera (ovviamente se ho la JMP la condizione sarà sempre vera), quindi se devo fare il salto.

```
jmp: begin
IP<=(jmp_condition(OPCODE,F)==1)?DEST_ADDR:IP;
STAR<=fetch0;
end

//------------- istruzione PUSH AL ----------------
```

// La push è una scrittura in memoria all'indirizzo del nuovo top della pila. Se SP punta al top della pila allora decremento per andare alla posizione successiva.

```
pushAL: begin
A23_A0<=SP-1;
SP<=SP-1;
APP0<=AL;
MJR<=fetch0;
STAR<=writeB;
end
```

Il decremento dipende dalla dimensione degli elementi di una pila (quanti byte?)

```
//------------- istruzione PUSH AH ----------------
```

// Ibidem

```
pushAH: begin
A23_A0<=SP-1;
SP<=SP-1;
APP0<=AH;
MJR<=fetch0;
STAR<=writeB;
end

//------------- istruzione PUSH DP ----------------
```

// Ibidem

```
pushDP: begin
A23_A0<=SP-3;
SP<=SP-3;
{APP2,APP1,APP0}<=DP;
MJR<=fetch0;
STAR<=writeM;
end

//------------- istruzione POP AL ----------------
```

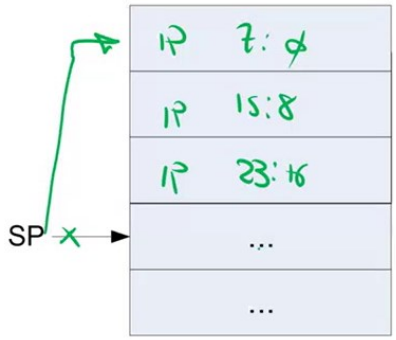
// Letture in memoria all'indirizzo puntato da SP. Ricordiamoci che dobbiamo incrementare in modo da rendere consistenti le letture successive: questo incremento dipende dal numero di byte letti.

```
popAL: begin
A23_A0<=SP;
SP<=SP+1;
```

```
MJR<=popAL1;
STAR<=readB;
end
popAL1: begin AL<=APP0; STAR<=fetch0; end

//------------ istruzione POP AH ---------------
// Ibidem
popAH: begin
A23_A0<=SP;
SP<=SP+1;
MJR<=popAH1;
STAR<=readB;
end
popAH1: begin AH<=APP0; STAR<=fetch0; end

//------------ istruzione POP DP ---------------
// Ibidem
popDP: begin
A23_A0<=SP;
SP<=SP+3;
MJR<=popDP1;
STAR<=readM;
end
popDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------ istruzione CALL indirizzo ---------
```

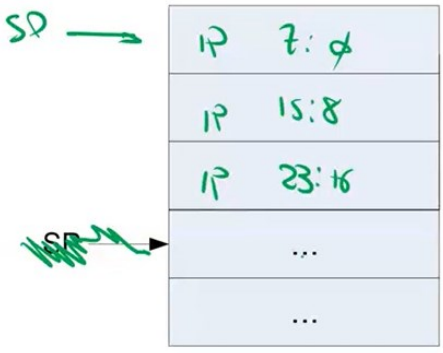
// La CALL è una JMP preceduta da una scrittura in memoria, se volete. Nella pila devo salvare 3 byte di IP. Quindi scrivo 3 byte in memoria all'indirizzo SP-3. Assegno ad IP il DESTINATION ADDRESS.

```
call: begin
A23_A0<=SP-3;
SP<=SP-3;
{APP2,APP1,APP0}<=IP;
MJR<=call1;
STAR<=writeM;
end
call1: begin IP<=DEST_ADDR; STAR<=fetch0; end

//------------ istruzione RET ---------------
```



// Si ha una lettura in memoria della pila e la sostituzione di IP con quanto letto.

```
ret: begin
A23_A0<=SP;
SP<=SP+3;
MJR<=ret1;
STAR<=readM;
end
ret1: begin IP<={APP2,APP1,APP0}; STAR<=fetch0;
end
```

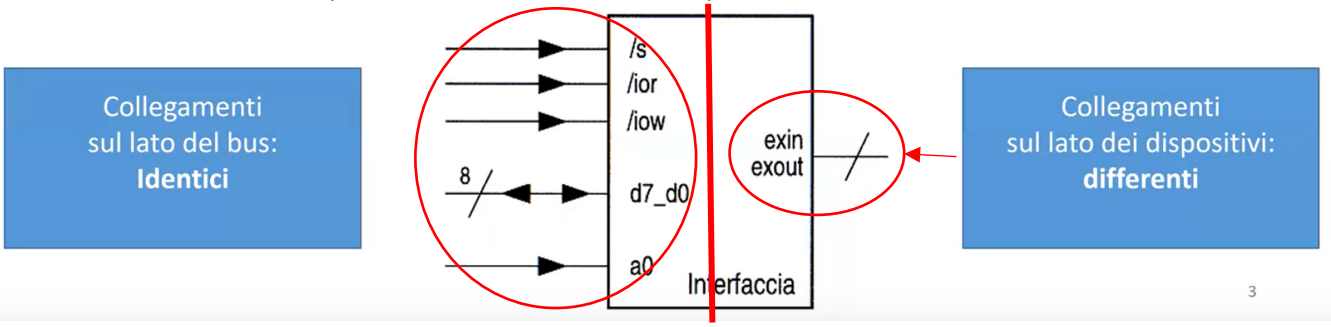


- **Conclusioni**:
  - In ogni stato della descrizione appena visto sono presenti al più micro-salti a due vie.
  - Se vogliamo essere precisi la maggior parte dei micro-salti sono a una solo via (micro-salti incondizionati). I micro-salti a due sono presenti quasi esclusivamente nelle sottoliste di lettura/scrittura in memoria.
  - Il processore è sintetizzabile con il metodo di scomposizione *Parte operativa – Parte controllo* visto a lezione. Le reti combinatorie contengono solo cose già viste a lezione. Non sono difficili da sintetizzare, il problema è solo la noia.
  - **Riflessione**: in soli due mesi di corso abbiamo acquisito la capacità di descrivere e sintetizzare dell'hardware capace di eseguire programmi software arbitrariamente complessi.

Interfacce

- A questo punto ci mancano le varie interfacce che completano il calcolatore.
- Vedremo interfacce di tre tipi:
  - interfacce parallele, quelle che colloquiano con dispositivi ai quali si invia un byte alla volta;
  - interfacce seriali, quelle che colloquiano con dispositivi con i quali si scambia un bit alla volta;
  - quelle per la conversione A-D e D-A, che trasformano gruppi di bit in tensioni e viceversa. Utilizzeremo la tensione come grandezza analogica.

- Le interfacce sono dispositivi collocati tra il bus e i dispositivi.



  - Per capire di quale interfaccia si parla dobbiamo guardare il lato dei dispositivi, diverso da interfaccia a interfaccia.
  - Dal lato del processore le interfacce sono tutte uguali e si presentano come piccole memorie.
  - Chiaramente se ho un solo filo di indirizzo avrò solo due porte: una di ingresso e una di uscita.
  - I fili di indirizzo rimanenti passano in una maschera che genera il /s.

Descrizione dell'interfaccia a livello funzionale:

- Chi usa l'interfaccia (un sistemista per il montaggio, o un programmatore) come la deve usare?

  - Nell'interfaccia sono presenti due registri:
    - RBR (*Receive Buffer Register*), dove salvo i dati scritti dal dispositivo esterno;
    - TBR (*Transmit Buffer Register*)., dove salvo i dati da mandare al dispositivo esterno.
  - L'indirizzo interno a_0 mi permette di indicare quale registro mi interessa (0 per *RBR* e 1 per *TBR*).

  - La lettura del registro RBR consiste nell'istruzione Assembler IN
    `IN offset_RBR, %AL`
  - La scrittura nel registro TBR consiste nell'istruzione Assembler OUT
    `OUT %AL, offset_TBR`
  - **Problema**: sincronizzazione tra processore e dispositivo. Il processore non ha modo di sincronizzarsi con i dispositivi (anche perché li vede solo come memorie)
    - Supponiamo che un programma contenga le seguenti istruzioni
      `IN offset_RBR, %AL`
      …
      `IN offset_RBR, %AL`
      Nessuno può garantirmi che tra le due IN il dispositivo sia stato in grado di produrre un dato nuovo. Il secondo dato potrebbe non essere significativo.
    - Dualmente…
      `OUT %AL, offset_TBR`
      …
      `OUT %AL, offset_TBR`

Nessuno può garantire che tra le due OUT il dispositivo abbia processato il dato.

Per risolvere questo problema dobbiamo dotarci di registri di stato per implementare un handshake tra processore e dispositivi. I due registri sono i seguenti:

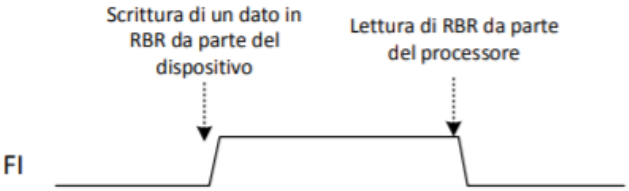
- RSR (*Receive Status Register*),
- TSR (*Transmit Status Register*).

Molto spesso collassati in un unico registro *RTSR*. Di ciascun registro è significativo un solo bit, rispettivamente i seguenti:

- *Flag di buffer ingresso pieno* (FI)
- *Flag di buffer di uscita vuoto* (FO)

I flag sono gestiti dall'interfaccia che li setta e resetta quando capta certi eventi.

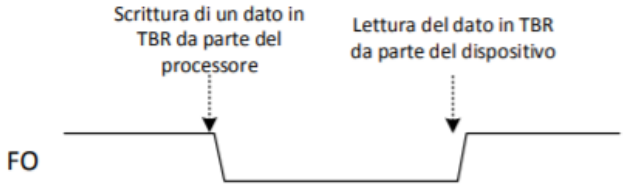
- **Registro FI**:
  - Il flag è inizialmente a 0.
  - L'interfaccia lo mette a 1 quando il dispositivo scrive un dato in RBR, segnalando la presenza di un nuovo dato
  - Quando il processore accede in lettura al registro RBR, l'interfaccia porta a 0 il flag FI.



  - Un sottoprogramma Assembler che legge dati nuovi da un'interfaccia con handshake, mettendoli dentro AL, è il seguente

```
testFI: IN RSR_offset,%AL      # Copia in AL il contenuto di RSR
        AND $0x01,%AL          # Evidenzia in AL il contenuto di FI
        JZ testFI              # cicla finché FI vale 0
        IN RBR_offset,%AL      # Copia in AL il contenuto di RBR
        RET                    # Ritorna al chiamante
```

- **Registro FO**:
  - Il flag inizialmente è a 1.
  - L'interfaccia lo mette a 0 quando il processore scrive un dato in TBR (istruzione OUT), segnalando che il dispositivo non ha ancora processato.
  - Quando il dispositivo (con i suoi tempi) accede al registro TBR e legge il dato, l'interfaccia porta nuovamente a 1 il flag FO.



  - Un sottoprogramma Assembler che scrive il contenuto di AL dentro TBR in un'interfaccia con handshake è il seguente

```
        PUSH %AL               # Salva in pila il contenuto di AL
testFO: IN TSR_offset, %AL     # Copia in AL il contenuto di TSR
        AND $0x20,%AL          # Evidenzia in AL il contenuto FO
        JZ testFO              # Salta indietro se FO era a 0
        POP %AL                # Ripristina il contenuto di AL
        OUT %AL,TBR_offset     # Immette in TBR il contenuto di AL
        RET                    # Ritorna al chiamante
```

Ma tutto questo è efficiente? Per nulla: l'idea è che il processore rimanga in attesa, cioè che cicli finchè il dispositivo esterno non sarà pronto. Il processore perde tempo, considerando la lentezza delle reti che comunicano con lui.

Molto meglio se il processore può andare avanti per conto proprio, con l'interfaccia che segnala al processore quando è pronto. A quel punto il processore interrompe il lavoro che sta facendo.

Direct memory access (DMA): il processore demanda ad un'altra unita (il DMA controller) il compito di trasferire dati tra la memoria e le interfacce, mentre va avanti con le sue elaborazioni. Il processore riferisce al DMA l'area di memoria e l'interfaccia con cui lavorare.