UNIVERSITÀ DEGLI STUDI DI BOLOGNA

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica
Calcolatori Elettronici M – Prof. Giovanni Neri, Prof. Stefano Mattoccia

Progetto di Calcolatori Elettronici M
"Progettazione di un branch target buffer per processore DLX Pipelined nel linguaggio VHDL"

Realizzato da:
Enrico Baioni
Raffaele Luca Iannario
Simone Tallevi Diotallevi

Anno Accademico 2009 - 2010

Indice

Introduzione

Il progetto realizzato consiste nell'implementazione in linguaggio VHDL di un Branch Target Buffer (BTB) da utilizzare con un processore DLX operante con una pipeline costituita da cinque stadi: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), Write Back (WB).

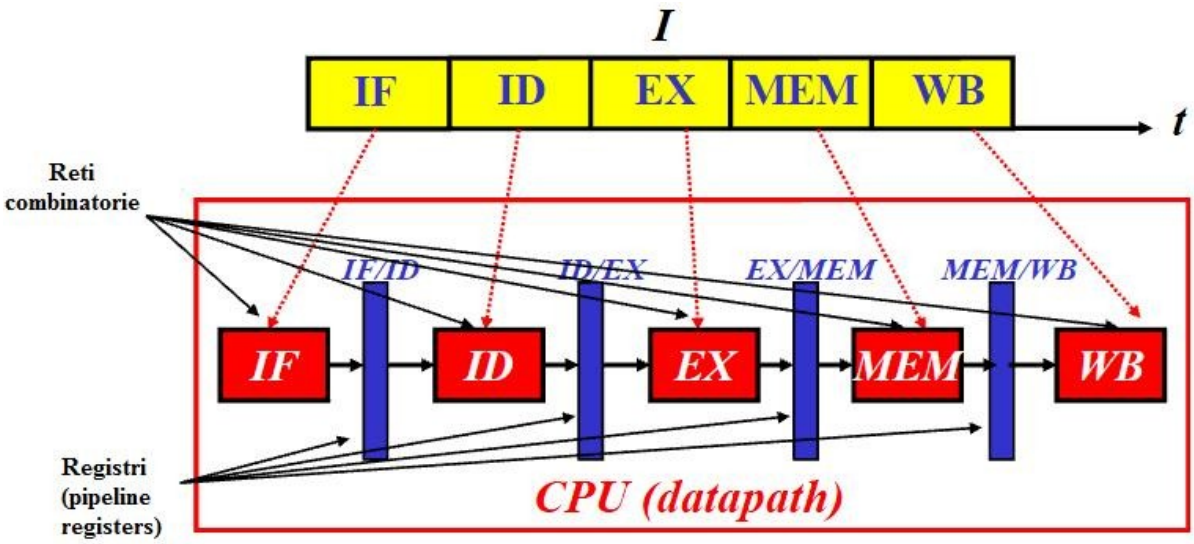


Figura 1. Pipeline a cinque stadi

Il componente realizzato dovrebbe fornire un incremento delle prestazioni della pipeline, in particolare dello throughput, in quanto permette di evitare stalli dovuti alle istruzioni di branch nel sistema DLX predicendo le stesse, grazie ad un algoritmo di predizione, nello stadio di IF. Il sistema verifica sempre la predizione nello stadio di EX in cui è valutata la condizione del branch. Nel caso di valutazione contrastante con la predizione è necessario inserire degli stalli all'interno della pipeline e riprendere l'esecuzione dall'istruzione corretta. Nel caso di valutazione coerente non è necessario apportare alcuna modifica al flusso di esecuzione poiché è stato già eseguito il fetch dell'istruzione corretta.

Architettura

Il BTB è realizzato come una cache i cui TAG sono costituiti dai Program Counter (PC) corrispondenti ad istruzioni che in precedenza sono state individuate, dallo stadio di EX, come branch. La prima volta che s'incontra un'istruzione di branch, non è presente una linea relativa all'interno del BTB; pertanto l'esecuzione procede normalmente fino allo stadio di EX, il quale valuta il branch e procede all'aggiornamento del BTB.

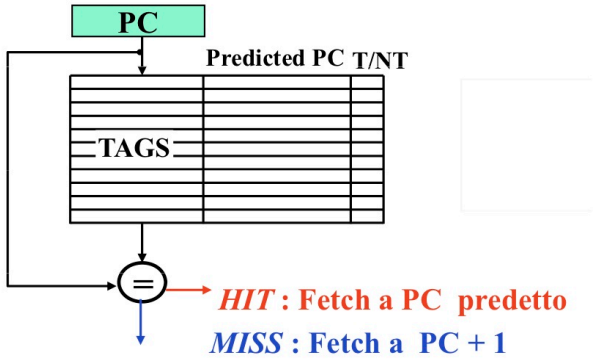


Figura 2. Funzionamento logico BTB

Struttura interna

Nel caso specifico, l'unità è stata realizzata mediante una cache set-associative a due vie da 64 slot.

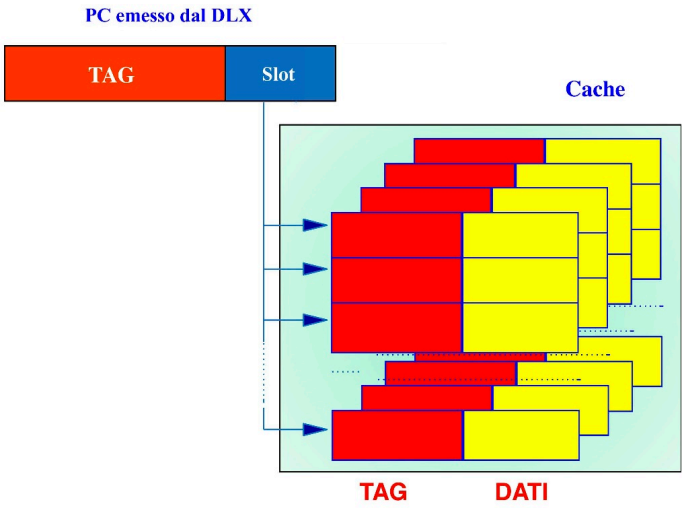


Figura 3. Cache set-associative a n vie

Ogni via segue la seguente modellazione:

TAG	DESTINAZIONE	PREDIZIONE	RIMPIAZZAMENTO	STATO

- TAG: 24 bit (30 relativi al PC a cui vengono sottratti i 6 che formano l'index). Siccome l'architettura RISC del DLX prevede indirizzi a 32 bit e poiché le istruzioni sono di lunghezza fissa (32 bit), di conseguenza allineate, gli ultimi due bit assumono sempre il valore zero ed è possibile non considerarli. È necessario un index a 6 bit per l'identificazione univoca dei 64 slot ($\log_2 64 = 6$)
- DESTINAZIONE: 30 bit che individuano l'indirizzo di destinazione
- PREDIZIONE: 2 bit (vedi Algoritmo di predizione)
- RIMPIAZZAMENTO: 1 bit (vedi Politica di rimpiazzamento)
- STATO: 1 bit che indica la validità della linea

Quindi ogni via è composta da 58 bit e di conseguenza ogni slot da 116 bit. La dimensione totale della cache è di 116*64/8 = 928 Byte.

Algoritmo di predizione

Gli algoritmi di predizione determinano il successo o meno di un BTB in quanto sono fondamentali per l'incremento delle prestazioni. Esistono diversi algoritmi di cui alcuni protetti da segreto industriale.

Nel progetto è stato implementato un algoritmo semplice che prevede l'utilizzo di due bit per la codifica di quattro stati al fine di memorizzare la storia relativa ad un'istruzione.

Basandosi sulla correttezza della predizione, comunicata dal mondo esterno al BTB, è possibile eseguire una transizione di stato seguendo la logica rappresentata in figura 3. In caso di MISS in scrittura lo stato iniziale è determinato staticamente e quindi portato in uno stato forte ("11" o "00") sulla base della correttezza della predizione iniziale, la quale, in caso di MISS in lettura, è sempre UNTAKEN poiché non c'è

modo di determinare la destinazione del branch (in mancanza di una logica aggiuntiva nello stadio di IF).

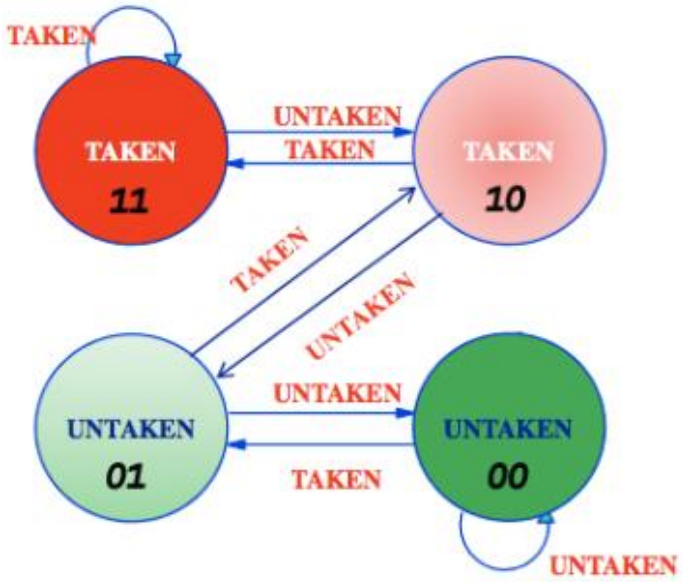


Figura 4. Grafo degli stati dell'algoritmo di predizione

I quattro stati garantiscono una maggiore robustezza nei casi di predizione errata. Infatti sono necessarie due "misprediction" consecutive per cambiare la predizione. Inoltre è possibile invertire repentinamente la predizione qualora si sbagliasse una terza volta, sempre consecutiva.

Con questo algoritmo di predizione è attesa un'accuratezza superiore all'80%.

Politica di rimpiazzamento

Quando il sistema si trova a regime, a causa della limitata dimensione della cache, a seguito della richiesta d'inserimento di una linea non presente, è possibile che occorra rimpiazzare una delle linee appartenenti ad un determinato slot. Tra le varie politiche possibili ne è stata scelta una di tipo Least Recently Used (LRU), la quale prevede la sostituzione della linea utilizzata meno recentemente.

Nel caso di una cache set-associative a due vie è necessario un bit per il rimpiazzamento per ogni linea di ogni slot. In realtà sarebbe sufficiente un solo bit per ogni slot che indichi la linea da rimpiazzare, ma è stato scelto comunque di utilizzare due campi dedicati in modo da fornire una maggiore scalabilità al sistema.

Le situazioni che portano all'applicazione della politica LRU sono essenzialmente due:

- HIT in lettura: occorre marcare la linea trovata come più giovane, portando il bit di rimpiazzamento al valore logico zero e settando il bit relativo all'altra via
- MISS in scrittura: sono possibili due situazioni. La prima è che una delle due vie (o entrambe) dello slot corrente sia invalida. In tal caso si scrivono tutte le informazioni in quella via (o nella prima linea invalida trovata) settatandola come valida e più giovane, mentre l'altra, se valida, diventa la più vecchia. La seconda situazione è che entrambe le linee siano valide e quindi occorra rimpiazzare la via più vecchia (bit di rimpiazzamento a "1"). Dopo la scrittura si agisce su tale linea come se ci fosse stato un HIT in lettura

Casi d'uso

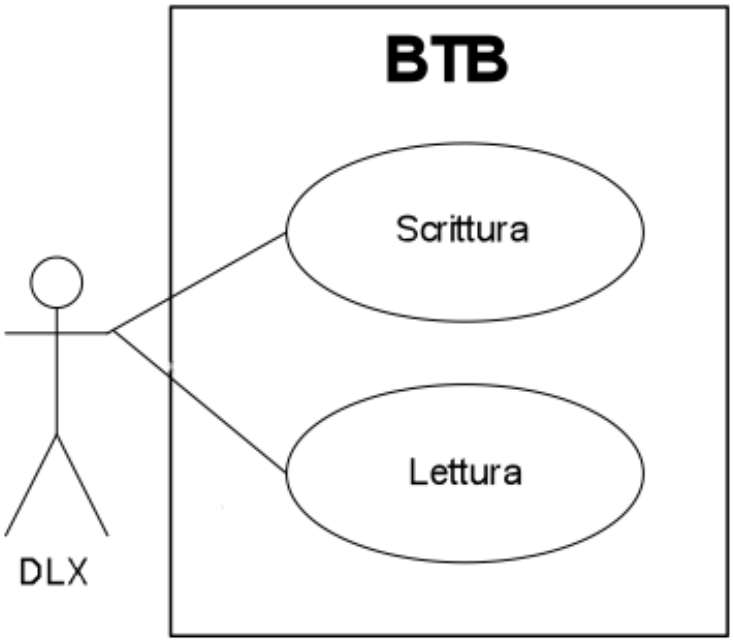


Figura 5. Casi d'uso

Scenario – Lettura

Descrizione

- Lo scenario descrive l'interrogazione del BTB da parte dello stadio di IF

Attore

- Stadio di IF

Flusso principale

1. Lo stato IF invia al BTB il PC corrente attivando il segnale RD
2. Il BTB ricerca il PC al suo interno
3. Il PC è presente e la linea è valida
4. Il BTB aggiorna i bit di rimpiazzamento
5. Il BTB emette i 30 bit relativi all'indirizzo di destinazione del branch e contemporaneamente emette il valore determinato dall'algoritmo di predizione sulla relativa uscita.
6. Lo stadio di IF, al clock successivo, esegue il fetch dell'istruzione successiva in base alla predizione

Flusso alternativo

3a. Il PC non è presente (o la corrispondente linea è invalida)

4a. Il BTB emette sempre come predizione UNTAKEN

5a. Lo stadio di IF considera l'indirizzo di destinazione ricevuto dal BTB come non significativo ed esegue il fetch all'indirizzo PC+1 (idealmente +4)

Scenario – Scrittura

Descrizione

- Lo scenario descrive il dialogo tra il BTB e lo stadio di EX

Attore

- Stadio di EX

Flusso principale

1. Lo stadio di EX invia al BTB il proprio PC (se esso è associato ad un'istruzione di branch), l'indirizzo di destinazione calcolato e la correttezza della predizione
2. Il BTB ricerca il PC al suo interno
3. Il PC è presente
4. Il BTB aggiorna i bit di predizione della via identificata dal PC
5. Il BTB sovrascrive l'indirizzo di destinazione

Flusso alternativo

3a. Il PC non è presente

4a. Il BTB applica la politica di rimpiazzamento per lo slot identificato dal PC

5a. Il BTB sovrascrive l'indirizzo di destinazione

6a. Il BTB inizializza lo stato della predizione

Nota

La riscrittura dell'indirizzo di destinazione avviene sempre, in previsione dell'utilizzo del BTB da parte di un processore in grado di eseguire istruzioni di branch non solamente con operando immediato (offset).

Realizzazione VHDL

In questa sezione verrà presentata l'implementazione del componente BTB nel linguaggio VHDL.

Pin in/out logico BTB

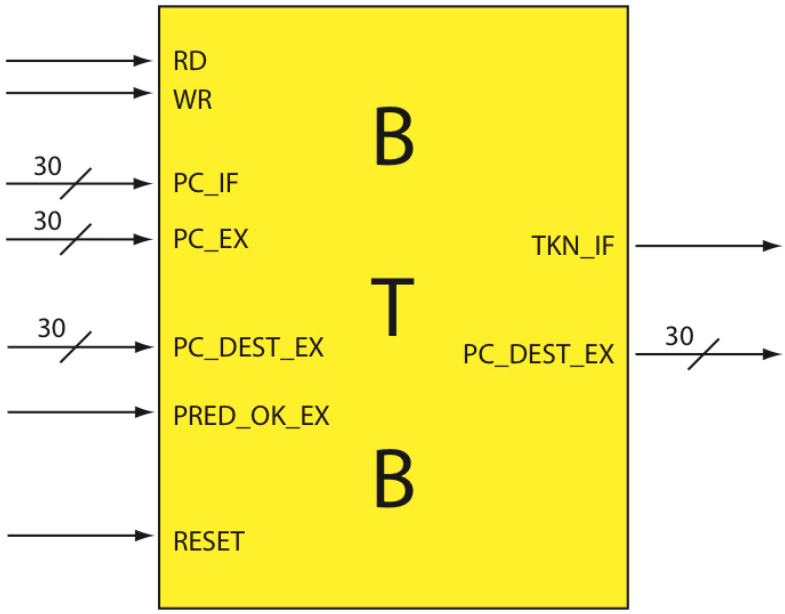


Figura 6. Pin in/out logico del componente BTB

- RD: settato dallo stadio di IF per la lettura del BTB
- WR: settato dallo stadio di EX per la scrittura sul BTB
- PC_IF: PC, inviato dallo stadio di IF, dell'istruzione di cui si vuole sapere la predizione
- PC_EX: PC, inviato dallo stadio di EX, dell'istruzione di cui si vuole aggiornare o aggiungere il record sul BTB
- PC_DEST_EX: PC di destinazione relativo all'istruzione di branch che si trova all'indirizzo PC_EX
- PRED_OK_EX: se alto significa che la predizione effettuata per l'istruzione all'indirizzo PC_EX è corretta. Se basso significa che la predizione è errata
- RESET: quando alto resetta il BTB invalidando tutte le linee
- TKN_IF: predizione relativa all'istruzione che si trova all'indirizzo PC_IF

- PC_DEST_IF: PC di destinazione dell'istruzione all'indirizzo PC_IF

Definizione del componente BTB in VHDL:

```
entity Btb_component is
    Port ( -- segnale write
           wr : in  STD_LOGIC;
           -- segnale read
           rd : in  STD_LOGIC;
           -- pc proveniente da if
           pc_if : in  std_logic_vector(PC_BITS-1 downto 0);
           -- pc proveniente da ex
           pc_ex : in  std_logic_vector(PC_BITS-1 downto 0);
           -- destinazione salto proveniente da ex
           pc_dest_ex : in  std_logic_vector(PC_BITS-1 downto 0);
           -- predizione corretta o meno (proveniente da ex)
           pred_ok_ex : in  STD_LOGIC;
           -- reset
           reset : in  STD_LOGIC;
           -- predizione
           tkn_if : out  STD_LOGIC;
           -- destinazione del salto se tkn_if = 1
           pc_dest_if : out  std_logic_vector(PC_BITS-1 downto 0));
end Btb_component;


type way_type is record
   tag_pc : std_logic_vector(TAG_BITS-1 downto 0);
   dest_pc : std_logic_vector(PC_BITS-1 downto 0);
   pred : std_logic_vector(PRED_BITS-1 downto 0);
   status: std_logic; -- 0 invalido 1 valido
   repl : std_logic; -- 1 se linea da sostituire
end record;

type btb_cache is array (0 to SLOTS_NUM-1, 0 to WAYS_NUM-1) of way_type;
```

Figura 7. Realizzazione interfaccia e struttura interna BTB

BTB Component Logic

Implementazione dello scenario di lettura:

```
   if(pc_if'event and rd = '1') then
      -- estrazione tag e index da pc_if
      tag_rd := pc_if(PC_BITS-1 downto SLOT_BITS);
      index_rd := conv_integer(pc_if(SLOT_BITS-1 downto 0));
      found_rd := '0';
      -- ricerca della linea di cache
      for i in 0 to WAYS_NUM-1 loop
      -- linea trovata e valida
         if(Btb_inst(index_rd, i).tag_pc = tag_rd and Btb_inst(index_rd, i).status = VALID) then
            found_rd := '1';
            found_index_rd := i;
             -- aggiornamento pc_dest_if
            pc_dest_if <= Btb_inst(index_rd, i).dest_pc;
            -- emissione bit di predizione
            case Btb_inst(index_rd, i).pred is
               when TAKEN_STRONG => tkn_if <= TAKEN;
               when TAKEN_WEAK => tkn_if <= TAKEN;
               when others => tkn_if <= UNTAKEN;
            end case;
            -- aggiornamento bit di rimpiazzamento
            Btb_inst(index_rd, i).repl <= '0';
         end if;
      end loop;
```

```
   -- gestione della politica di rimpiazzamento
   -- la linea è stata trovata
   if(found_rd = '1') then
      for i in 0 to WAYS_NUM-1 loop
         --se linea valida diversa da quella che ha appena subito un hit
         if(i /= found_index_rd and Btb_inst(index_rd, i).status = VALID) then
            --aggiornamento predizione
            Btb_inst(index_rd, i).repl <= '1';
         end if;
      end loop;
   -- la linea non è stata trovata
   else
      tkn_if <= UNTAKEN;
   end if;
end if;
```

Figura 8. Logica di lettura

Implementazione dello scenario di scrittura:

```
-- Scrittura
if(pc_ex'event and wr = '1') then
   -- estrazione tag e index da pc_ex
   tag_wr := pc_ex(PC_BITS-1 downto SLOT_BITS);
   index_wr := conv_integer(pc_ex(SLOT_BITS-1 downto 0));
   --inizializzazione variabili
   found_wr := '0';
   found_invalid_wr := '0';
   -- linea trovata
   for i in 0 to WAYS_NUM-1 loop
      if(Btb_inst(index_wr, i).tag_pc = tag_wr and Btb_inst(index_wr, i).status = VALID) then
         report "Scrittura: Linea trovata";
         found_wr := '1';
         -- aggiornamento indirizzo di destinazione
         Btb_inst(index_wr, i).dest_pc <= pc_dest_ex;


      --aggiornamento bit di predizione
      -- predizione corretta
      if(pred_ok_ex = PRED_OK) then
         case Btb_inst(index_wr, i).pred is
            when TAKEN_STRONG => Btb_inst(index_wr, i).pred <= TAKEN_STRONG;
            when TAKEN_WEAK => Btb_inst(index_wr, i).pred <= TAKEN_STRONG;
            when UNTAKEN_WEAK => Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
            when UNTAKEN_STRONG => Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
            when others => Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
         end case;
      -- predizione sbagliata
      else
         case Btb_inst(index_wr, i).pred is
            when TAKEN_STRONG => Btb_inst(index_wr, i).pred <= TAKEN_WEAK;
            when TAKEN_WEAK => Btb_inst(index_wr, i).pred <= UNTAKEN_WEAK;
            when UNTAKEN_WEAK => Btb_inst(index_wr, i).pred <= TAKEN_WEAK;
            when UNTAKEN_STRONG => Btb_inst(index_wr, i).pred <= UNTAKEN_WEAK;
            when others => Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
         end case;
      end if;
      exit;
   end if;
end loop;
```

```
--linea non trovata
-- cerco linea valida o invalida (da rimpiazzare per la prima scrittura)
if(found_wr = '0') then
   for i in 0 to WAYS_NUM-1 loop
      -- trovata linea invalida
      if(Btb_inst(index_wr, i).status = INVALID) then
         found_invalid_wr := '1';
         found_invalid_index_wr := i;
         -- aggiornamento linea
         Btb_inst(index_wr, i).tag_pc <= tag_wr;
         Btb_inst(index_wr, i).dest_pc <= pc_dest_ex;
         Btb_inst(index_wr, i).status <= VALID;
         Btb_inst(index_wr, i).repl <= '0';
         -- la linea non era nella cache
         -- predizione in lettura untaken
         if(pred_ok_ex = PRED_OK) then
            Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
         else
            Btb_inst(index_wr, i).pred <= TAKEN_STRONG;
         end if;
         exit;
      end if;
   end loop;


-- trovata linea invalida
if(found_invalid_wr = '1') then
   for i in 0 to WAYS_NUM-1 loop
      -- linea diversa da quella trovata e valida
      if(i /= found_invalid_index_wr and Btb_inst(index_wr, i).status = VALID) then
         --aggiornamento bit di rimpiazzamento
         Btb_inst(index_wr, i).repl <= '1';
      end if;
   end loop;


-- non ci sono linee invalide
else
   for i in 0 to WAYS_NUM-1 loop
      -- trovata la linea valida da rimpiazzare
      if(Btb_inst(index_wr, i).repl = '1') then
         -- aggiornamento linea
         Btb_inst(index_wr, i).tag_pc <= tag_wr;
         Btb_inst(index_wr, i).dest_pc <= pc_dest_ex;
         Btb_inst(index_wr, i).repl <= '0';
         -- la linea non era nella cache
         -- predizione in lettura untaken
         if(pred_ok_ex = PRED_OK) then
            Btb_inst(index_wr, i).pred <= UNTAKEN_STRONG;
         else
            Btb_inst(index_wr, i).pred <= TAKEN_STRONG;
         end if;
      else
         Btb_inst(index_wr, i).repl <= '1';
      end if;
   end loop;
end if;
```

Figura 9. Logica di scrittura

Integrazione con il sistema DLX Pipelined

```
--segnali per il btb
btb_fetch_pc_dest: inout std_logic_vector(PC_BITS-1 downto 0);
btb_fetch_tkn: inout std_logic;
btb_fetch_rd : inout std_logic;
btb_pred_ok: inout std_logic;
btb_exe_wr: inout std_logic;
btb_exe_pc_dest: inout std_logic_vector(PC_BITS-1 downto 0);
btb_exe_tkn: inout std_logic;
--segnali per le statistiche
btb_exe_num_branch_pred_ok: out std_logic_vector(PC_BITS-1 downto 0);
btb_exe_num_branch_pred_not_ok: out std_logic_vector(PC_BITS-1 downto 0);


Btb_component_inst: Btb_component PORT MAP (
    wr => btb_exe_wr,
    rd => btb_fetch_rd,
    pc_if => pc_fetch,
    pc_ex => pc_execute,
    pc_dest_ex => btb_exe_pc_dest,
    pred_ok_ex => btb_pred_ok,
    reset => reset,
    tkn_if => btb_fetch_tkn,
    pc_dest_if => btb_fetch_pc_dest
  );
```

Figura 10. Port map

```
when I_BNEZ =>
   pc_dest_btb <= pc_buffer + to_stdlogicvector(to_bitvector(sxt(a_immediate_16, PC_BITS)) sra 2) + 1;
   if conv_integer(var_register_a) /= 0 then -- branch da prendere
      --segnali per il btb
      wr_btb <= '1';
      pc_for_jump <= pc_buffer + to_stdlogicvector(to_bitvector(sxt(a_immediate_16, PC_BITS)) sra 2) + 1;
      if(tkn_buffer = TAKEN) then -- e preso
         pred_ok_btb <= PRED_OK;
         if(pc_buffer'event) then num_branch_pred_ok_buffer := num_branch_pred_ok_buffer + 1; end if;
      else -- e non preso
         force_jump <= '1';
         pred_ok_btb <= PRED_NOT_OK;
         if(pc_buffer'event) then num_branch_pred_not_ok_buffer := num_branch_pred_not_ok_buffer + 1; end if;
      end if;
   else -- branch da non prendere
      wr_btb <= '1';
      if(tkn_buffer = UNTAKEN) then -- e non preso
         pred_ok_btb <= PRED_OK;
         pc_for_jump <= pc_buffer + to_stdlogicvector(to_bitvector(sxt(a_immediate_16, PC_BITS)) sra 2) + 1;
         if(pc_buffer'event) then num_branch_pred_ok_buffer := num_branch_pred_ok_buffer + 1; end if;
      else -- e preso
         force_jump <= '1';
         pred_ok_btb <= PRED_NOT_OK;
         pc_for_jump <= pc_buffer + 1;
         if(pc_buffer'event) then num_branch_pred_not_ok_buffer := num_branch_pred_not_ok_buffer + 1; end if;
      end if;
   end if;
   alu_exit <= (others => '0');
```

Figura 11. Modifiche allo stadio di EX

Affinchè il processore DLX possa utilizzare il BTB sono necessarie modifiche alla sua pipeline per quanto concerne gli stadi di IF, ID ed EX. Lo stadio di IF deve essere in grado di pilotare il segnale RD e PC_IF verso il BTB e campionare i segnali di risposta TKN_IF e PC_DEST_IF. È fondamentale portare la predizione lungo la pipeline per dare la possibilità allo stadio di EX di verificare la correttezza della stessa. A tal fine, la modifica riguardante lo stadio di ID è semplicemente l'aggiunta di un segnale, appunto la predizione, da campionare dall'uscita del BTB e da trasmettere allo stadio di EX. Le modifiche relative allo stadio di EX sono più articolate in quanto, oltre all'aggiunta dei segnali di WR, PC_DEST_EX e PRED_OK (PC_EX già presente) che saranno connessi al BTB, è necessario modificare la logica in caso di predizione errata: in tale situazione occorre riportare la pipeline nello stato corretto attivando il segnale già presente di force_jump e settando pc_for_jump. L'unità J&B, presente nel sistema, eseguirà il fetch asincrono dell'istruzione corretta e provvederà a trasformare in NOP l'istruzione presente nello stadio ID.

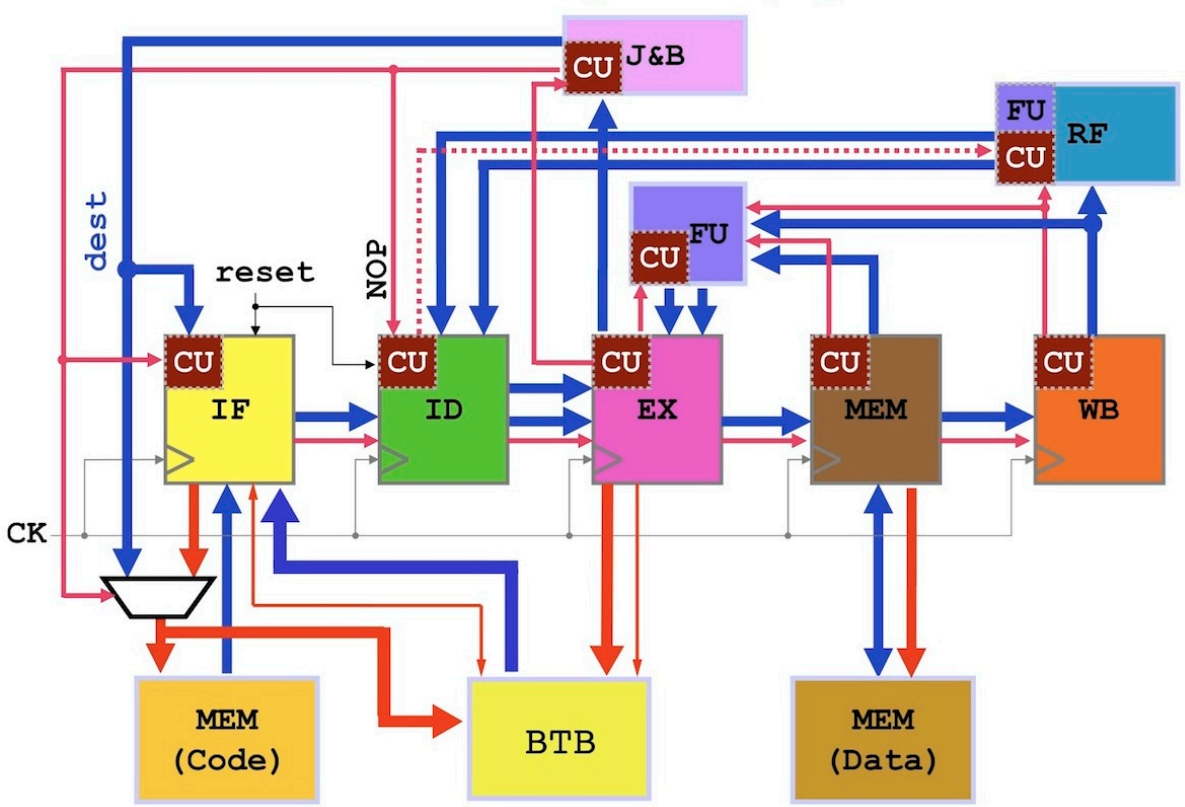


Figura 12. Schema a blocchi aggiornato della pipeline

Test bench

Il test bench è composto da più fasi:

1. Reset: si tiene alto il segnale di reset e ciò permette di verificare che l'invalidazione del BTB avvenga in maniera corretta
2. Riempimento del BTB con dati fittizi in modo da permettere il test del funzionamento a regime
3. Scrittura di un PC la cui linea è già presente nel BTB ma con la comunicazione di una predizione non corretta
4. Verifica che un cambiamento dei segnali di PC_EX e PC_DEST_EX non influenza il BTB se il segnale di WR è basso
5. Scrittura dello stesso PC del punto 3, nel BTB, con la comunicazione di una predizione non corretta
6. Lettura dei primi slot del BTB per verificare l'esattezza dei dati inseriti
7. Verifica che un cambiamento dei segnali di PC_IF non influenza il BTB se il segnale RD è basso

Nota: le letture e le scritture avvengono simultaneamente dopo la fase di reset per simulare il comportamento del BTB in presenza di accessi concorrenti.

Esaminando l'output, si può vedere l'istanza del BTB al termine della fase di reset

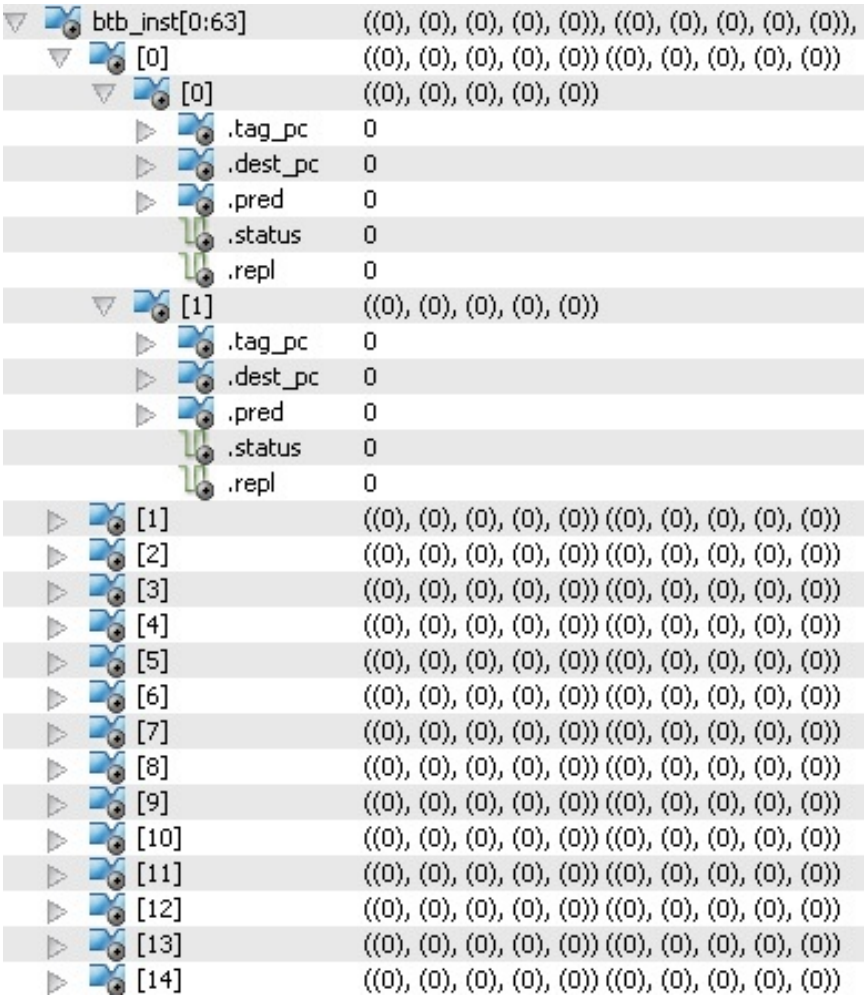


Figura 13. BTB alla fine del reset (dati in base dieci)

Dal grafico della simulazione si possono osservare le risposte in caso di lettura: il BTB risponde correttamente alle interrogazioni in quanto pilota alternativamente il segnale di TKN_IF. Questo è il comportamento voluto poiché in fase d'inizializzazione si è deciso di impostare gli slot pari con predizione corretta, quindi il BTB risponde UNTAKEN, mentre agli slot dispari è stata segnalata una predizione sbagliata e quindi il BTB risponde con TAKEN. Si ricorda che quando si presenta un PC a cui non è associata alcuna linea, il BTB risponde UNTAKEN in quanto non vi è modo di predire la destinazione del branch.

Per quanto riguarda le scritture, si nota che la predizione del PC 64 ha subito due transizioni di stato. La prima volta è stata segnalata giusta la predizione e di conseguenza si è partiti da uno stato di UNTAKEN forte. Le successive due scritture hanno

comunicato, sempre per il PC 64, due predizioni errate e quindi si è passati da UN-TAKEN forte a TAKEN debole. A verifica di ciò, l'ultima lettura con PC 64, ha ottenuto come previsione TAKEN (TKN_IF alto).

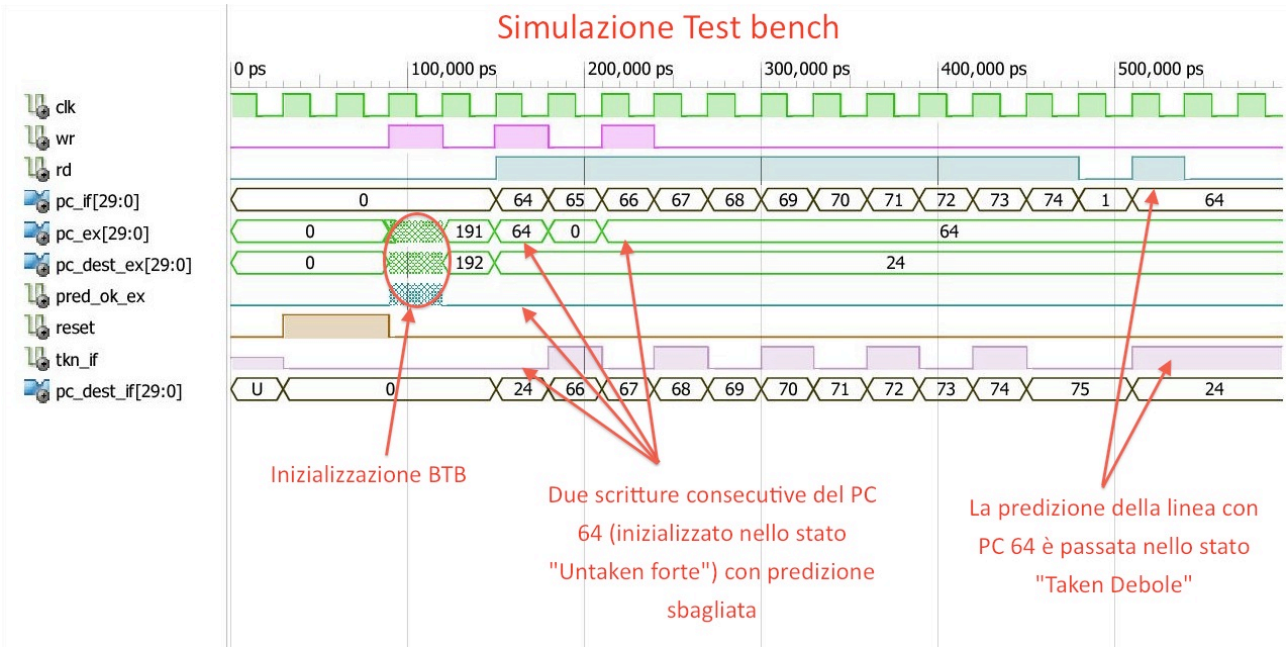


Figura 14. Grafico della simulazione test bench

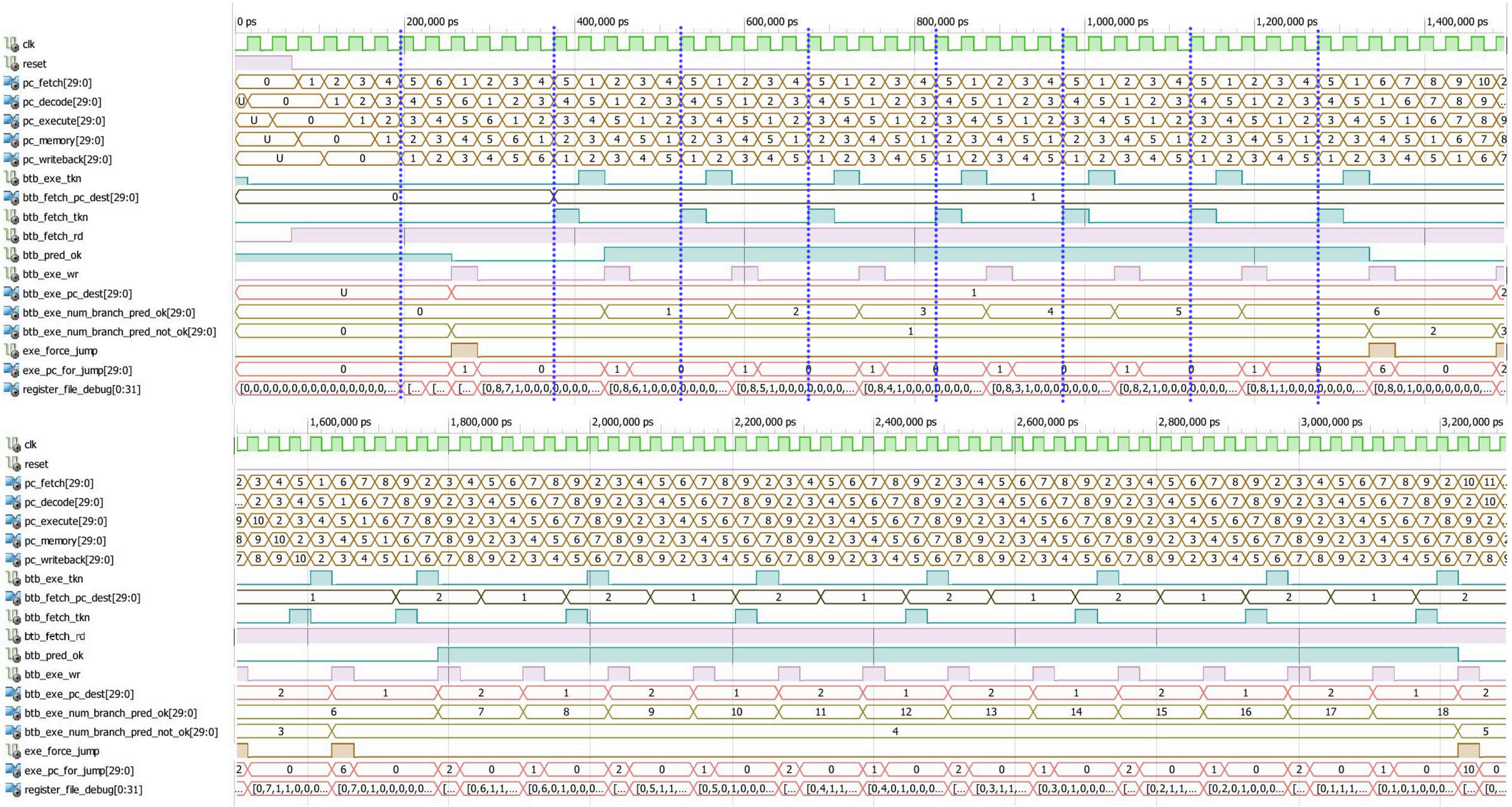
Risultati sperimentali

Conclusa la parte d'integrazione con il sistema DLX Pipelined, sono state eseguite simulazioni per esaminare il funzionamento del BTB e le migliorie che esso apporta all'esecuzione delle istruzioni lungo la pipeline. Si può notare che, al ripresentarsi di un'istruzione di branch taken e di predizione giusta, non si esegue più il fetch all'indirizzo corretto con un clock di ritardo. Dalla simulazione si evince anche che, quando il segnale btb_pred_ok è pilotato verso il basso si attiva il segnale exe_force_jump della J&B Unit per ripristinare il corretto flusso delle istruzioni.

Le simulazioni sono state effettuate eseguendo programmi che presentano loop innestati in modo da testare l'effettiva efficacia del componente aggiunto. Al fine di raccogliere dati statistici sono stati inseriti nella simulazione due contatori che indicano il numero di predizioni corrette ed errate (btb_exe_num_branch_pred_ok e btb_exe_num_branch_pred_not_ok). Questi due contatori hanno evidenziato un comportamento previsto: maggiore è il numero d'iterazioni del loop e maggiore è il guadagno in periodi di clock.

Per completezza si allega il grafico di una simulazione completa relativa al seguente programma:

```
l1: addi r2, r1, 8
l2: addi r1, r0, 8
l3: subi r2, r2, 1
l4: addi r3, r0, 1
l5: addi r1, r1, 0
l6: bnez r2, l2
l7: addi r2, r2, 1
l8: subi r1, r1, 1
l9: addi r3, r0, 1
l10: bnez r1, l3
```


0 ps
200,000 ps
400,000 ps
600,000 ps
800,000 ps
1,000,000 ps
1,200,000 ps
1,400,000 ps
1,600,000 ps
1,800,000 ps
2,000,000 ps
2,200,000 ps
2,400,000 ps
2,600,000 ps
2,800,000 ps
3,000,000 ps
3,200,000 ps
clk
reset
pc_fetch[29:0]
pc_decode[29:0]
pc_execute[29:0]
pc_memory[29:0]
pc_writeback[29:0]
btb_exe_tkn
btb_fetch_pc_dest[29:0]
btb_fetch_tkn
btb_fetch_rd
btb_pred_ok
btb_exe_wr
btb_exe_pc_dest[29:0]
btb_exe_num_branch_pred_ok[29:0]
btb_exe_num_branch_pred_not_ok[29:0]
exe_force_jump
exe_pc_for_jump[29:0]
register_file_debug[0:31]

Conclusioni

Analizzando i dati delle simulazioni è emerso un incremento delle prestazioni, in termini di tempo medio di esecuzione, del 12% circa.

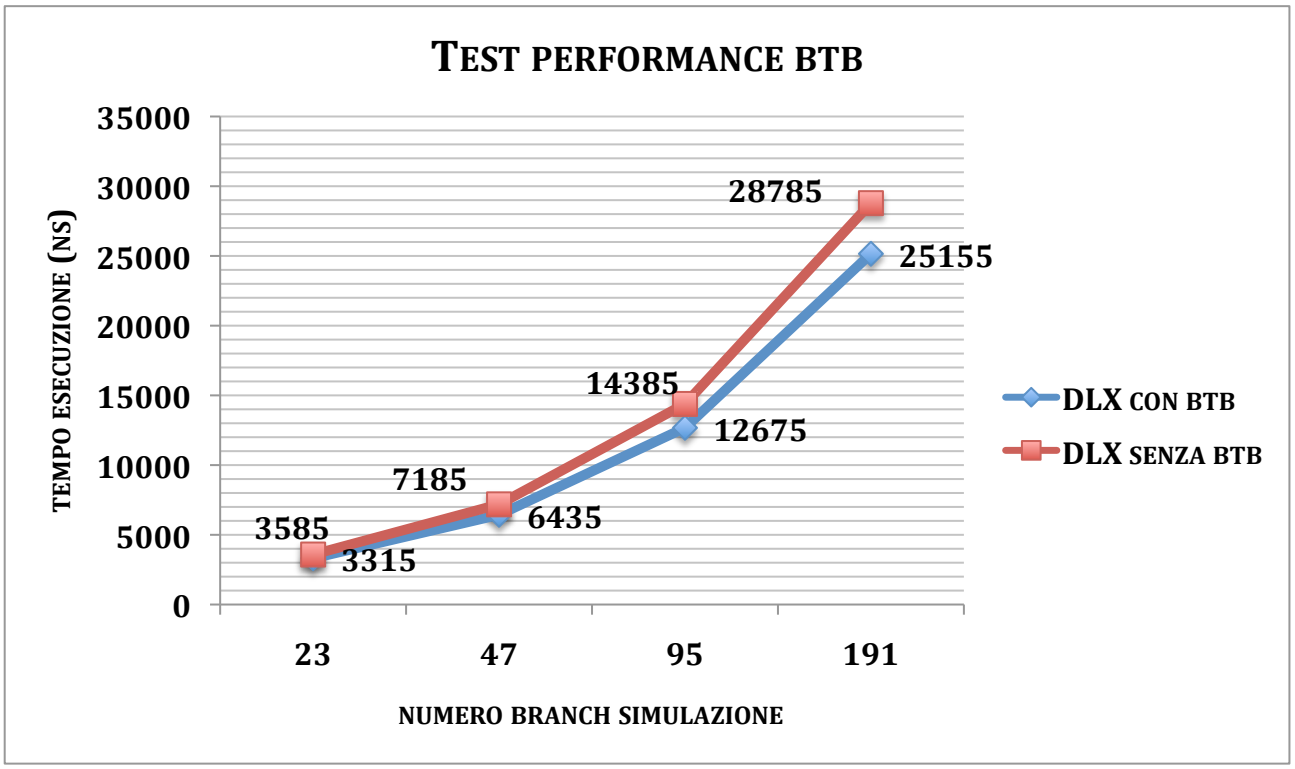


Figura 15. Grafico test performance sul tempo di esecuzione

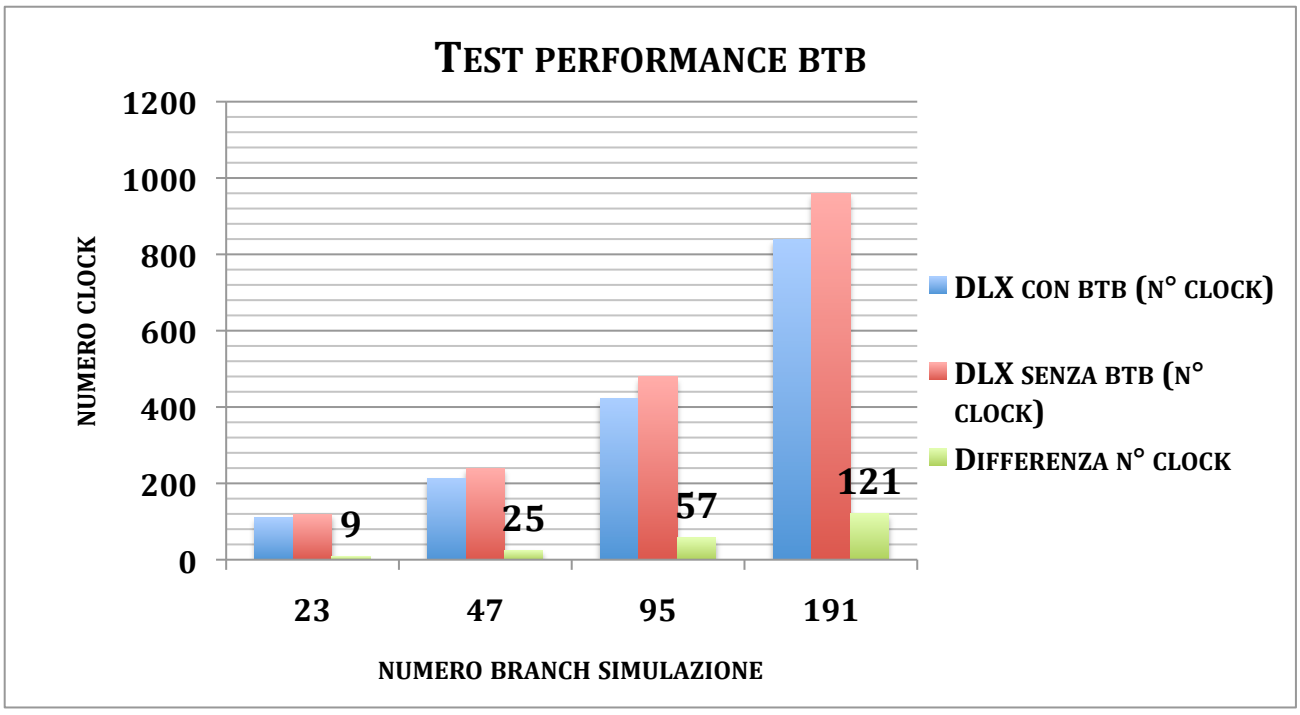


Figura 16. Grafico test performance sul numero di clock

Il miglioramento è da considerarsi comunque valido in quanto, nel sistema DLX utilizzato per questo progetto, è presente la logica della J&B Unit che permette di avere un solo stallo lungo la pipeline in caso d'istruzioni di jump o branch taken. Addirittura, nel caso in cui l'istruzione di branch è predetta TAKEN in modo errato, l'uso del BTB è svantaggioso in quanto obbliga lo stadio di IF ad eseguire il fetch all'indirizzo di destinazione del salto, con successivo intervento della J&B, mentre l'esecuzione senza BTB prevederebbe il fetch all'istruzione successiva (che è quella corretta). È evidente che questo caso non mette in discussione il vantaggio apportato del BTB poiché i dati statistici, estratti dalle simulazioni, confermano che l'algoritmo di predizione garantisce predizioni corrette in più dell'80% dei casi.

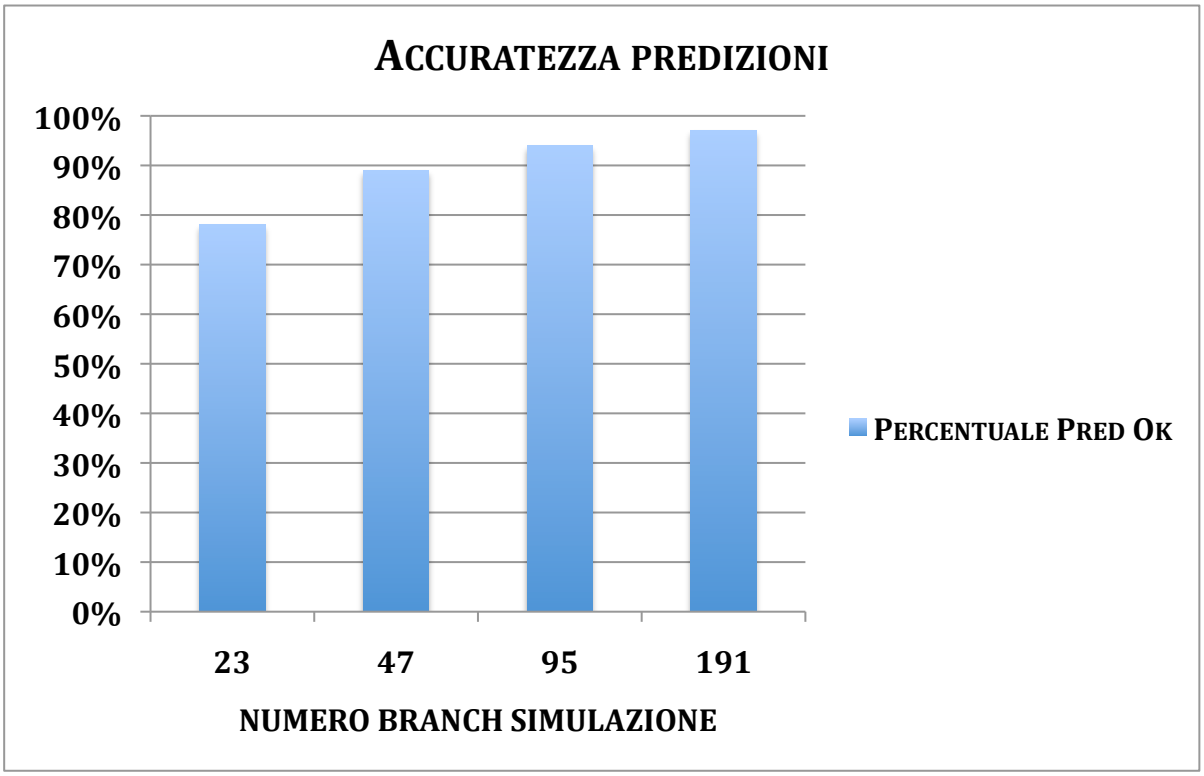


Figura 17. Grafico accuratezza predizioni