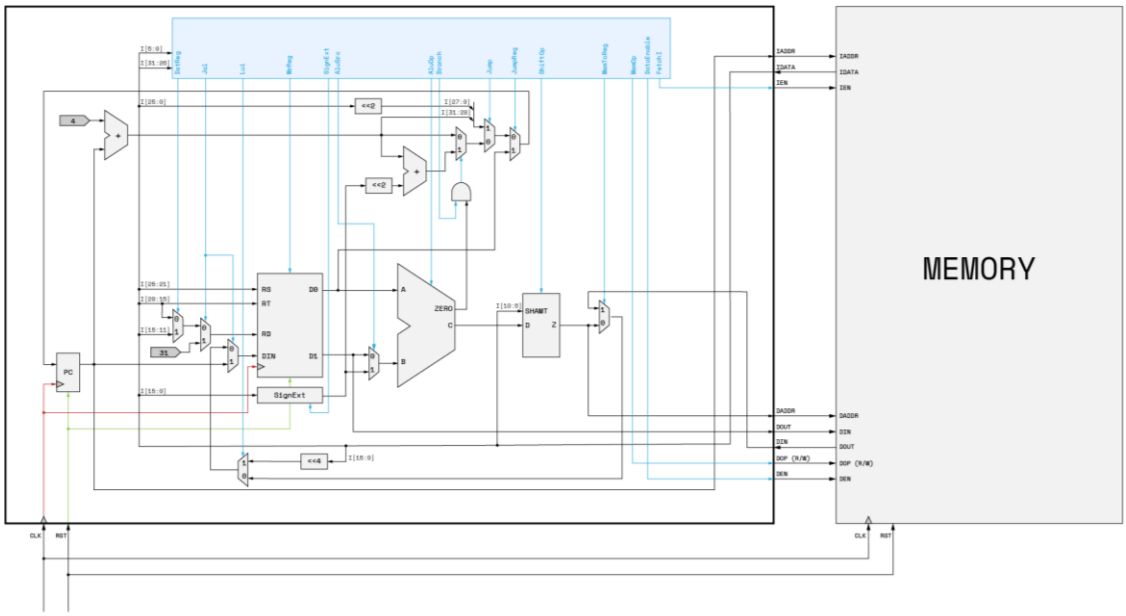
Progetto di Reti Logiche

MIPS - Microprocessor without Interlocked Pipeline Stages

Michele Bersani, Paolo Chiappini, Andrea Fraschini

Introduzione

Implementazione di un'architettura MIPS (Multiprocessor without Interlocked Pipeline Stages) a singolo ciclo, senza pipeline a 32 bit sulla base dell'immagine seguente:



L'architettura prevede l'implementazione di un instruction set ridotto che include le seguenti istruzioni:

- add Add
- sub Subtract
- addi Add Immediate
- addiu addi Unsigned
- and And
- or Or
- nor Nor
- xor Exclusive Or
- andi And Immediate
- ori Or Immediate
- xori Xor Immediate
- lw Load Word
- sw Store Word
- sll Shift Left Logical
- srl Shift Right Logical
- sra Shift Right Arithmetic
- slt Set if Less Than
- sltu Set if Less Than Unsigned
- slti Set if Less Than Immediate
- sltiu slti Unsigned
- j Jump
- jr Jump Register
- jal Jump And Link
- lui Load Upper Immediate
- beq Branch if Equal
- bne Branch if Not Equal

Le istruzioni vengono caricate dalla memoria, anch'essa semplificata per avere una dimensione complessiva ridotta di 1 Kbyte.

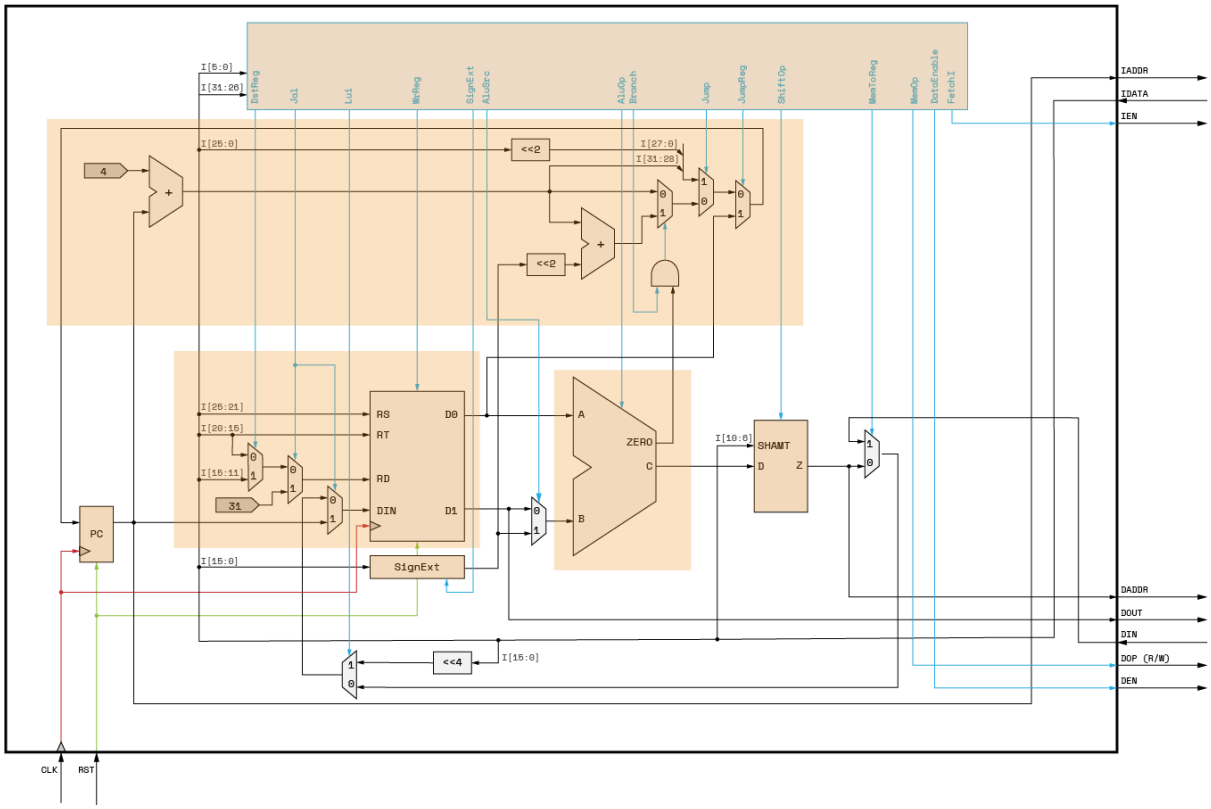
Trattandosi di un'architettura a singolo ciclo, è richiesto che tutte le operazioni in corrispondenza di un'istruzione vengano iniziate e concluse all'interno di un periodo di clock, permettendo quindi l'esecuzione di un'istruzione per ciclo di clock.

Specifica

L'architettura del sistema deve permettere l'esecuzione di tutte le istruzioni citate in precedenza. Ogni istruzione deve occupare al massimo un ciclo di clock e deve attraversare correttamente gli stadi caratteristici dell'architettura MIPS: Fetch, Decode, Execute, Memory e Writeback.

L'architettura deve interfacciarsi con un modulo Memory che permetta di simulare il funzionamento (semplificato) di una memoria RAM da 1 KB a parole indirizzabili da 8 bit, per un totale di 1024 indirizzi (`0x00000000` - `0x000003FC`). Attraverso l'interfaccia, Microprocessore e Memoria devono essere in grado di scambiare informazioni relative alle istruzioni e ai dati. In particolare, la memoria deve essere in grado di selezionare correttamente gli indirizzi comunicati dal processore e deve poter leggere le istruzioni ed eseguire operazioni di lettura o scrittura sui dati; dall'altra parte il processore deve essere in grado di interpretare correttamente le istruzioni sulla base del loro formato e contenuto al fine di produrre un risultato corretto. In caso di istruzioni non riconosciute, l'architettura non deve fare nulla.

A partire dall'immagine mostrata in precedenza, l'architettura viene divisa nei seguenti blocchi logici:



I moduli così individuati sono nominati: PC, SIGN_EXT, REG_UNIT, CONTROL_UNIT, MANAGEMENT_PC, ALU e SHIFTER.

I moduli hanno i seguenti compiti:

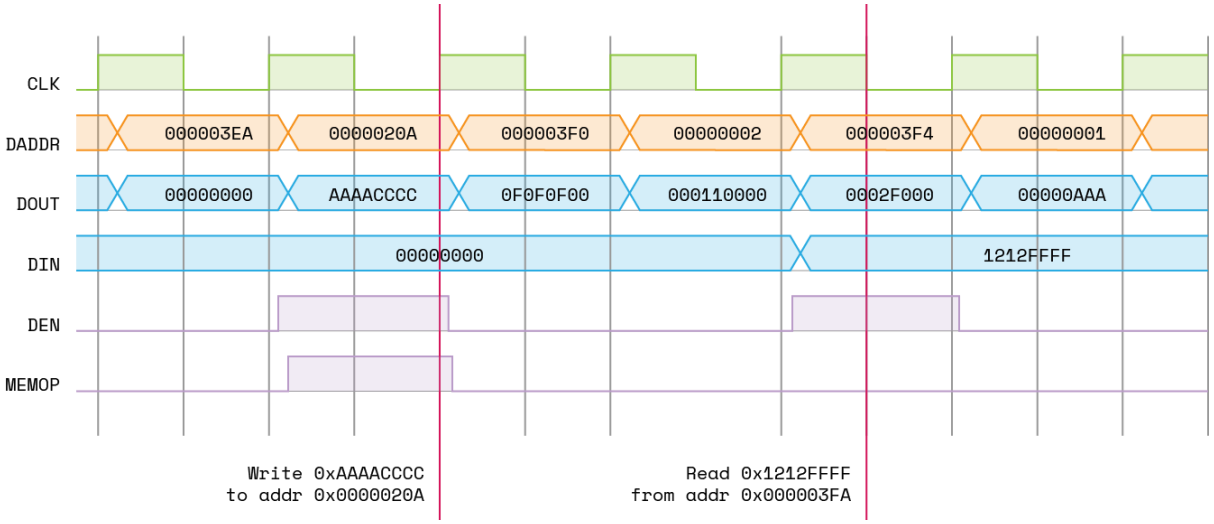
- PC : registro Program Counter, deve occuparsi della memorizzazione dell'indirizzo dell'istruzione corrente. Gli indirizzi devono avere dimensione 32 bit. Il registro deve essere attivo alto sul fronte di clock e deve presentare un reset sincrono. Nel momento del reset, il valore iniziale deve essere `0x00000080`.
- SIGN_EXT : il modulo si occupa dell'estensione del segno dell'immediato, pertanto deve trasformare valori a 16 bit in valori a 32 bit dove i primi 16 bit sono uguali al bit più significativo dell'immediato, mentre gli ultimi 16 bit corrispondono all'immediato stesso. Non tutte le operazioni richiedono di effettuare l'estensione del segno, in particolare le istruzioni `andi`, `ori`, `xori` e `sltiu` necessitano che l'estensione sia fatta ponendo i primi 16 bit a `0` andando ad effettuare l'equivalente di uno shift logico a destra di 16 posizioni.
- REG_UNIT : il modulo deve occuparsi della gestione dei 32 registri indirizzabili (register file) dalle istruzioni, il che include la gestione della selezione dei registri di lettura e di scrittura e la selezione e memorizzazione dei dati in ingresso. I registri devono essere attivi alti sul fronte di clock e devono presentare un reset sincrono. I registri `$gp` e `$sp` devono essere inizializzati rispettivamente ai valori `0x00000300` e `0x000003FC`.
- CONTROL_UNIT : il modulo deve occuparsi dell'interpretazione delle istruzioni e della conseguente generazione di segnali di controllo. I segnali di controllo devono tener conto del formato delle istruzioni (R, I, J) in modo da poter imporre comportamenti specifici sul resto dei moduli. Il valore dei segnali dipende dall'architettura dei singoli moduli.
- MANAGEMENT_PC : il modulo si occupa della gestione dell'aggiornamento del PC. L'aggiornamento è determinato dai segnali di controllo Jump, JumpRegister, Branch (beq, bne) e dal flag Zero della ALU.
    - Jump: `PC = PC & 0xF0000000 | (ADDR << 2);`
    - JumpRegister: `PC = $rs;`
    - BEQ: `if($rs = $rt) PC += 4 + (IMM << 2);`
    - BNE: `if($rs != $rt) PC += 4 + (IMM << 2).`
- ALU : il modulo deve permettere l'esecuzione di tutte le operazioni aritmetico-logiche, ovvero: `add`, `sub`, `addi`, `and`, `or`, `nor`, `xor`, `andi`, `ori`, `xori`, `slt`, `sltu`, `slti` e `sltiu`. Le operazioni sono fatte su due operandi da 32 bit e devono produrre in uscita un risultato anch'esso a 32 bit. Oltre al risultato dell'operazione, il modulo deve produrre in uscita un segnale a 1 bit chiamato Zero che valga 1 qualora il risultato dell'ultima operazione fosse `0`, il segnale vale `0` altrimenti. La tipologia di operazione deve poter essere scelta mediante la CU e deve poter essere possibile distinguere tra operazioni su operandi signed e unsigned.
- SHIFTER : il modulo deve permettere l'esecuzione di tutte le operazioni di shift: `sll`, `srl` e `sra`. Lo shift deve essere variabile sulla base del valore Shamt (shift amount) e deve essere eseguito in un ciclo di clock (*quindi no shift register o simili*).

Interfaccia del sistema

Il sistema, oltre ai segnali CLK (clock) e RST (reset), presenta un'interfaccia unicamente rivolta verso la memoria, la quale è composta dai seguenti segnali:

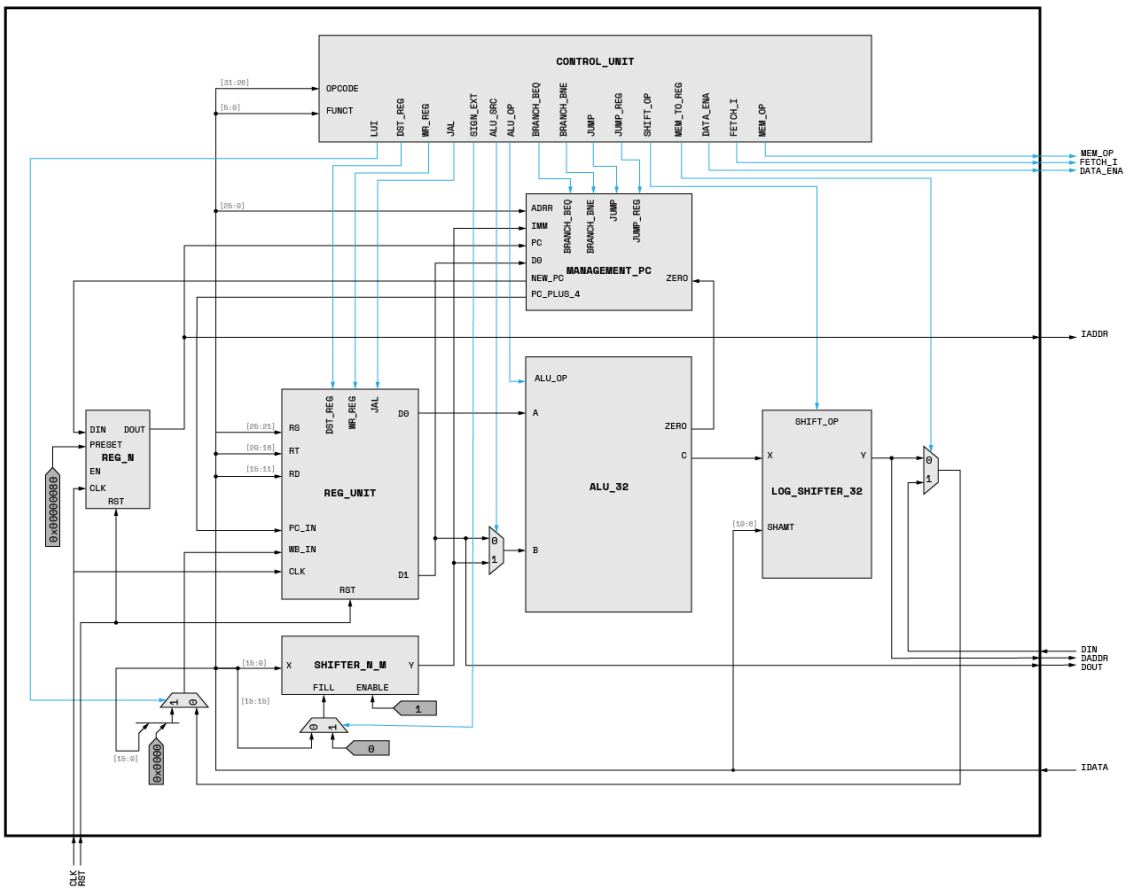
- IADDR : segnale in uscita in codifica binaria a 32 bit, rappresenta l'indirizzo dell'istruzione da leggere da memoria durante lo stadio di Fetch, corrisponde con il valore del registro PC;
- IDATA : segnale in ingresso in codifica binaria a 32 bit, rappresenta l'istruzione letta dalla memoria;
- IEN : segnale di controllo in uscita in codifica binaria a 1 bit, rappresenta il segnale di abilitazione alla lettura dell'istruzione da memoria;
- DADDR : segnale in uscita in codifica binaria a 32 bit, rappresenta l'indirizzo in memoria dei dati a cui si vuole accedere/scrivere;
- MEMOP : segnale di controllo in uscita in codifica binaria a 1 bit, rappresenta la tipologia di operazione che si vuole eseguire sui dati in memoria: se 0 corrisponde a una lettura, se 1 corrisponde a una scrittura;
- DEN : segnale di controllo in uscita in codifica binaria a 1 bit, rappresenta il segnale di abilitazione alle operazioni di lettura/scrittura sui dati (dipende da MEMOP);
- DIN : segnale in ingresso in codifica binaria a 32 bit, rappresenta il dato ricevuto dalla memoria a seguito di un'operazione di lettura
- DOUT : segnale in uscita in codifica binaria a 32 bit, rappresenta il dato in ingresso alla memoria.

Le operazioni di lettura e scrittura avvengono come segue:



Architettura del sistema

L'architettura complessiva del sistema è come riportata nell'immagine di seguito:



Essa è composta da 5 moduli principali:

- `CONTROL_UNIT` : modulo responsabile per l'interpretazione delle istruzioni ricevute dalla memoria e per la generazione dei segnali di controllo che determinano l'evoluzione degli altri moduli interni al sistema. I segnali di controllo sono evidenziati in azzurro nell'immagine e possono essere collegati direttamente ad un modulo per poi essere gestiti internamente, oppure ne possono influenzare gli ingressi.
- `MANAGEMENT_PC` : modulo che permette la gestione del PC (program counter) che è visibile come registro nella parte sinistra dell'immagine. La gestione del PC consiste essenzialmente nel decidere come aggiornare il registro sulla base dei segnali di controllo generati dalla CU (control unit).
- `REG_UNIT` : modulo responsabile per la gestione dei registri (ad eccezione del PC). L'aggiornamento dei registri è determinato dai segnali di controllo generati dalla CU.
- `ALU_32` : modulo responsabile dell'esecuzione di tutte le operazioni aritmetico-logiche dell'instruction set. Gli operandi della ALU, così come il risultato in uscita, dipendono sempre dai segnali di controllo.
- `LOG_SHIFTER_32` : modulo responsabile per le operazioni di shift. La tipologia di shift è determinata dai segnali di controllo.

Oltre ai moduli principali, nell'immagine nella pagina precedente, sono visibili altri due moduli:

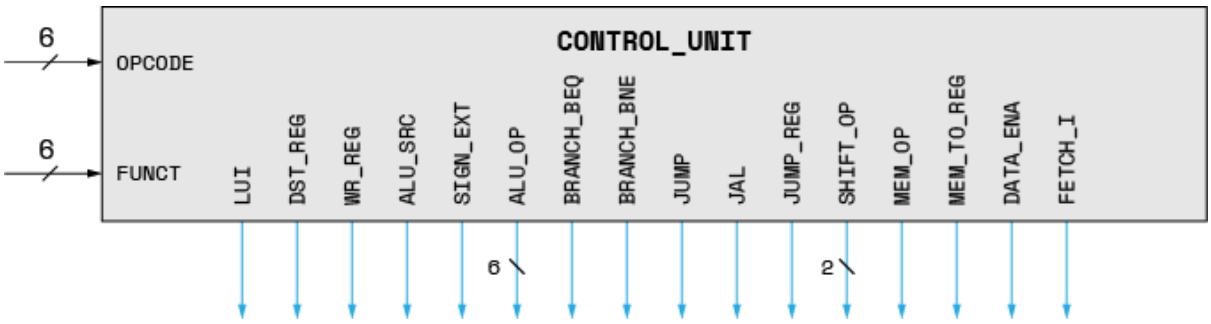
- `REG_N` : modulo generico che rappresenta un registro a N bit, nell'architettura sopra corrisponde con il registro PC.
- `SHIFTER_N_M` : modulo generico che rappresenta uno shifter verso destra di M posizioni con ingresso (e uscita) a N bit, nell'architettura sopra corrisponde con la funzione di sign extend.

(I moduli sono descritti più in dettaglio nelle rispettive sezioni)

I macro-moduli che compongono l'architettura sono pensati per svolgere funzioni molto specifiche, pertanto non si prestano bene al riuso, tuttavia, essi sono composti a loro volta da altri sottomoduli pensati per essere riutilizzati quanto più possibile, due dei quali sono già stati elencati appena sopra.

(Ulteriori sottomoduli generici pensati per essere riutilizzati saranno descritti in seguito quando si parlerà in dettaglio dell'architettura interna ai macro-moduli)

Modulo CONTROL_UNIT



Il modulo presenta i seguenti segnali:

- input:
  - `OPCODE` : segnale in codifica binaria a 6 bit, rappresenta codice operativo dell'istruzione;
  - `FUNCT` : segnale in codifica binaria a 6 bit, rappresenta codice di funzione dell'istruzione;
- output:
  - `LUI` : segnale in codifica binaria a 1 bit, posto a 1 quando l'istruzione è di tipo "`lui`" per fare lo shift di 16 bit, `0` altrimenti;
  - `DST_REG` : segnale in codifica binaria a 1 bit, ha lo scopo di selezionare in quale registro viene salvato il risultato dell'operazione; 1 per operazioni di tipo `R`, `0` per operazioni tipo `I`;
  - `WR_REG` : segnale in codifica binaria a 1 bit, posto a 1 se il dato deve essere scritto nel registro di destinazione, `0` altrimenti;
  - `ALU_SRC` : segnale in codifica binaria a 1 bit, posto a 1 se l'operando dell'ALU è un immediato, `0` se proviene da un registro;
  - `SIGN_EXT`: segnale in codifica binaria a 1 bit, posto a `0` se al numero immediato va esteso il segno, 1 altrimenti;
  - `ALU_OP` : segnale in codifica binaria a 6 bit, rappresenta il segnale per definire l'operazione specifica che deve eseguire l'ALU; (si veda tabella di seguito)
  - `BRANCH_BEQ` : segnale in codifica binaria a 1 bit, posto a 1 se l'operazione è di tipo "beq", `0` altrimenti;
  - `BRANCH_BNE` : segnale in codifica binaria a 1 bit, posto a 1 se l'operazione è di tipo "bne", `0` altrimenti;
  - `JUMP` : segnale in codifica binaria a 1 bit, posto a 1 se l'operazione è di tipo "j", `0` altrimenti;
  - `JAL` : segnale in codifica binaria a 1 bit, posto a 1 se l'operazione è di tipo "jal", `0` altrimenti;
  - `JUMP_REG` : segnale in codifica binaria a 1 bit, posto a 1 se il PC prende direttamente il valore da un registro;
  - `SHIFT_OP` : segnale in codifica binaria a 2 bit, serve per identificare il tipo di shift (si veda successivamente il modulo `SHIFTER_N_M`);

- MEM_OP : segnale in codifica binaria a 1 bit, specifica l'operazione che deve fare la memoria: posto a 1 se si scrive in memoria, 0 se si legge;
- MEM_TO_REG : segnale in codifica binaria a 1 bit, posto a 1 se il valore che viene mandato al Register File è preso in ingresso dalla memoria, 0 altrimenti;
- DATA_ENA : segnale in codifica binaria a 1 bit, posto a 1 l'istruzione prevede di legge o scrivere in memoria, 0 altrimenti;
- FETCH_I : segnale in codifica binaria a 1 bit, posto a 1 l'istruzione per abilitare la lettura dell'istruzione successiva;

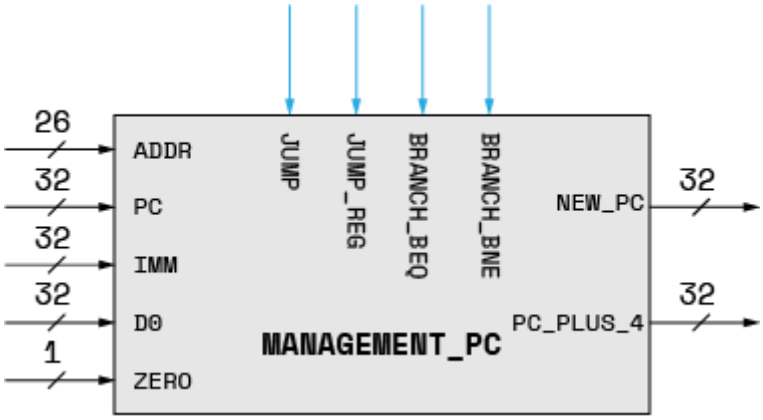
La CU è sintetizzata come una mappa, la relazione che si ha tra input ed output è descritta dalla seguente tabella:

Istruzione	Tipo	Opcode	Funct	Segnali															
				LUI	DstReg	WrReg	AluSrc	SignExt	AluOp	BranchBeq	BranchBne	Jump	Jal	JumpReg	ShiftOp	MemOp	MemToReg	DataEna	Fetchl
add	R	000000	100000	0	1	1	0	0	00-100	0	0	0	0	0	00	-	0	0	1
sub	R	000000	100010	0	1	1	0	0	01-100	0	0	0	0	0	00	-	0	0	1
addi	I	001000		0	0	1	1	0	00-100	0	0	0	0	0	00	-	0	0	1
and	R	000000	100100	0	1	1	0	0	00-001	0	0	0	0	0	00	-	0	0	1
or	R	000000	100101	0	1	1	0	0	00-011	0	0	0	0	0	00	-	0	0	1
nor	R	000000	100111	0	1	1	0	0	11-001	0	0	0	0	0	00	-	0	0	1
xor	R	000000	100110	0	1	1	0	0	00-011	0	0	0	0	0	00	-	0	0	1
andi	I	001100		0	0	1	1	1	00-001	0	0	0	0	0	00	-	0	0	1
ori	I	001101		0	0	1	1	1	00-010	0	0	0	0	0	00	-	0	0	1
xori	I	001110		0	0	1	1	1	00-011	0	0	0	0	0	00	-	0	0	1
sll	R	000000	000000	0	1	1	0	0	---000	0	0	0	0	0	01	-	0	0	1
srl	R	000000	000010	0	1	1	0	0	---000	0	0	0	0	0	10	-	0	0	1
sra	R	000000	000011	0	1	1	0	0	---000	0	0	0	0	0	11	-	0	0	1
slt	R	000000	101010	0	1	1	0	0	011101	0	0	0	0	0	00	-	0	0	1
sltu	R	000000	101011	0	1	1	0	0	010101	0	0	0	0	0	00	-	0	0	1
slti	I	001010		0	0	1	1	0	011101	0	0	0	0	0	00	-	0	0	1
sltiu	I	001011		0	0	1	1	1	010101	0	0	0	0	0	00	-	0	0	1
j	J	000010		0	-	0	-	0	------	0	0	1	-	0	--	-	0	0	1
jal	J	000011		0	-	1	-	0	------	0	0	1	1	0	00	-	0	0	1
jr	R	000000	001000	0	-	0	-	0	------	-	-	-	-	1	00	-	0	0	1
beq	I	000100		0	-	0	0	0	01-100	1	0	0	-	0	00	-	0	0	1
bne	I	000101		0	-	0	0	0	01-100	0	1	0	-	0	00	-	0	0	1
lui	I	001111		1	0	1	1	0	---000	0	0	0	0	0	00	-	0	0	1
lw	I	100011		0	0	1	1	0	00-100	0	0	0	0	0	00	0	1	1	1
sw	I	101011		0	-	0	1	0	00-100	0	0	0	0	0	00	1	0	1	1
"NOP"	-	-	-	-	-	0	-	-	------	0	0	0	-	0	--	-	-	0	1

Tabella per identificare l'operazione specifica della ALU:

OpID	Op	AluOP	A_INV	B_INV	SIGNED	ALU_OP[2:0]
0	B	---000	-	-	-	000
1	A + B	00-100	0	0	-	100
2	A - B	01-100	0	1	-	100
3	A < B (U)	010101	0	1	0	101
4	A < B (S)	011101	0	1	1	101
5	A and B	00-001	0	0	-	001
6	A or B	00-010	0	0	-	010
7	A nor B	11-001	1	1	-	001
8	A xor B	00-011	0	0	-	011
9	DC	------	-	-	-	---

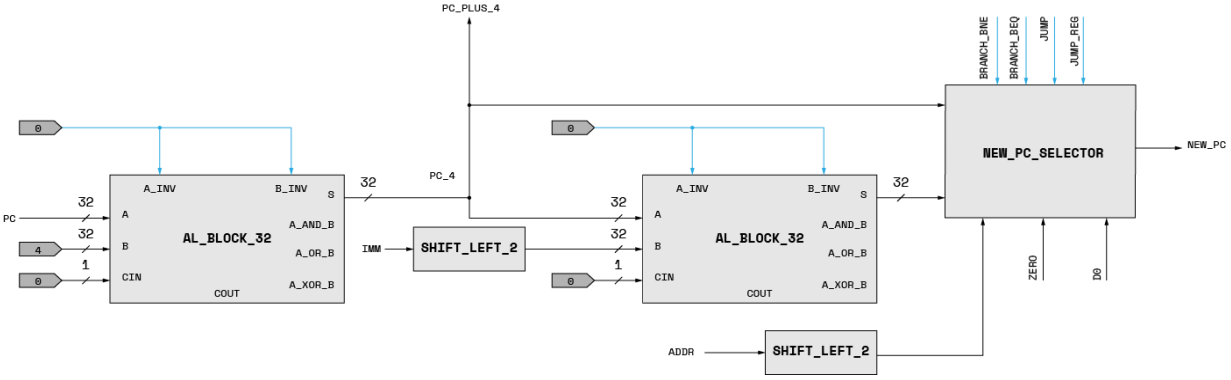
Modulo MANAGEMENT_PC



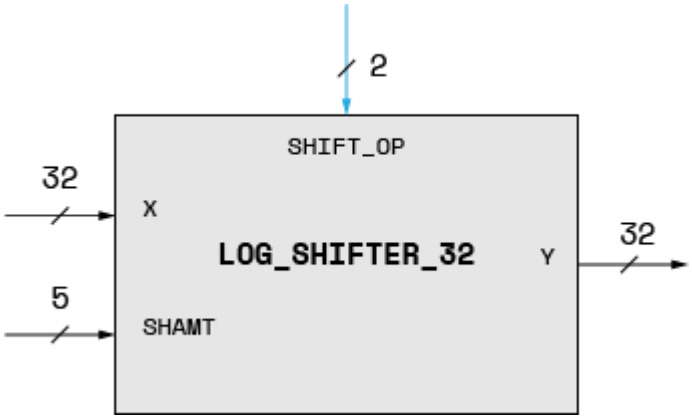
Il modulo presenta i seguenti segnali:

- Input:
  - BRANCH_BNE, BRANCH_BEQ, JUMP_REG, JUMP: segnali in codifica binaria a 1 bit, che sono direttamente collegati alla Control Unit;
  - ADDR: segnali in codifica binaria a 26 bit, rappresenta parte del nuovo indirizzo del PC, usato nelle operazioni di tipo salto incondizionato jump;
  - PC: segnali in codifica binaria a 32 bit, rappresenta indirizzo dell'istruzione corrente letta dal MIPS;
  - IMM: segnali in codifica binaria a 32 bit, rappresenta il valore immediato specificato nello operazioni di salto condizionato;
  - D0: segnali in codifica binaria a 32 bit, rappresenta il valore memorizzato in un registro, viene usato per i salti di tipo jal;
  - ZERO: segnali in codifica binaria a 1 bit, è il risultato della comparazione dei due operandi in ingresso nell'ALU, 1 se gli operandi sono uguali tra loro, 0 altrimenti;
- Output:
  - NEW_PC: segnali in codifica binaria a 32 bit, rappresenta il nuovo valore del PC che punta all'istruzione successiva anche in caso di salti;
  - PC_PLUS_4: segnali in codifica binaria a 32 bit, rappresenta il nuovo valore del PC che punta all'istruzione successiva senza i salti;

Il modulo nello specifico si presenta nel seguente modo:



Modulo LOG_SHIFTER_32



Il modulo presenta i seguenti segnali:

- X : segnale in ingresso in codifica binaria a 32 bit, rappresenta l'operando dell'operazione di shift;
- Y : segnale in uscita in codifica binaria a 32 bit, rappresenta il risultato dell'operazione di shift;
- SHAMT : segnale in ingresso in codifica binaria a 5 bit, rappresenta lo Shift Amount, ovvero la quantità di bit da spostare nell'operazione di shift;
- SHIFT_OP : segnale di controllo in ingresso in codifica binaria a 2 bit, rappresenta il tipo di shift da eseguire

Il modulo permette di eseguire 4 operazioni di shift:

- Shift nullo: il modulo riproduce sull'uscita Y il valore inalterato dell'ingresso X;

$$Y = X$$

- Shift logico a sinistra (SLL): il modulo produce sull'uscita Y il valore dell'ingresso X spostato verso sinistra di tante posizioni quante specificate dal segnale di SHAMT, i restanti spazi vengono riempiti con il valore logico 0;

$$Y = X << SHAMT$$

- Shift logico a destra (SRL): il modulo produce sull'uscita Y il valore dell'ingresso X spostato verso destra di tante posizioni quante specificate dal segnale di SHAMT, i restanti spazi vengono riempiti con il valore logico 0;

$$Y = X >> SHAMT$$

- Shift aritmetico a destra (SRA): il modulo produce sull'uscita Y il valore dell'ingresso X spostato verso destra di tante posizioni quante specificate dal segnale di SHAMT, i restanti spazi vengono riempiti con il valore logico del bit più significativo dell'ingresso X.

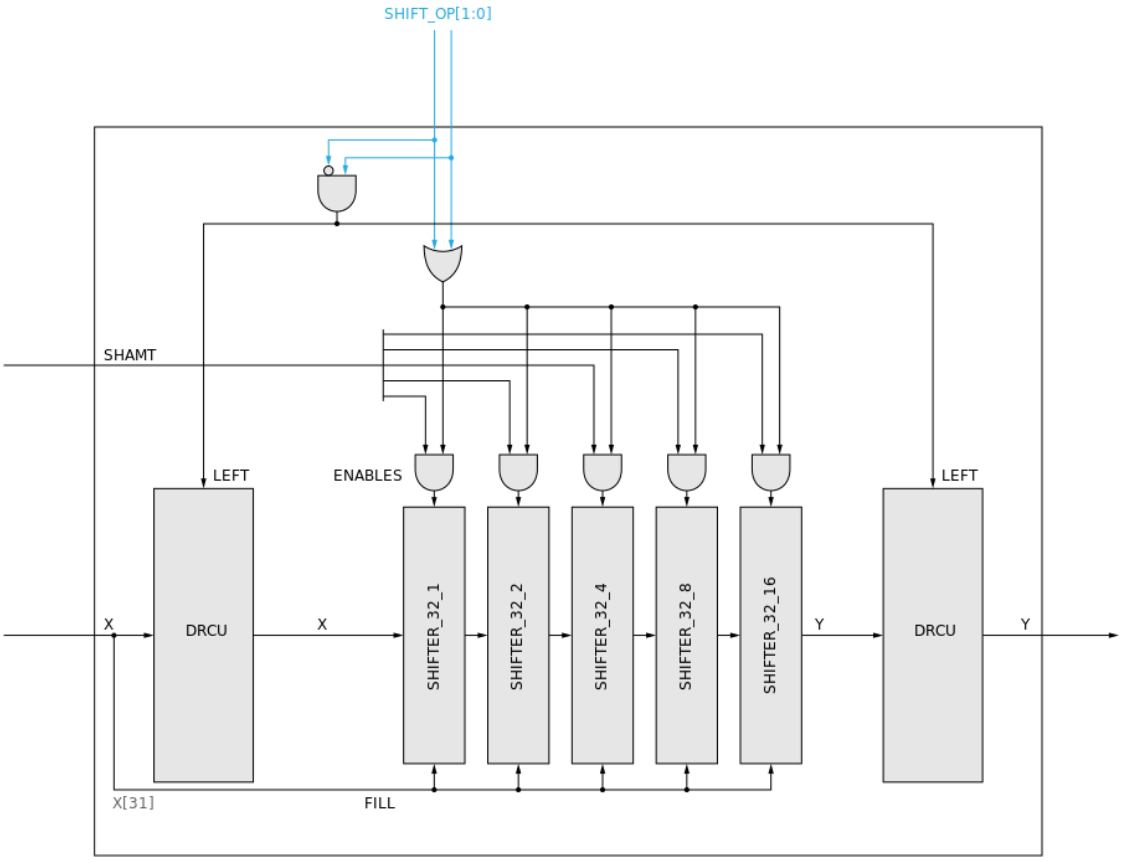
Abbiamo scelto di utilizzare un Logarithmic Barrel Shifter per l'architettura dello shifter. Questa scelta è stata motivata dalla sua bassa complessità in termini di spazio e tempo, che

dipende dalla dimensione degli ingressi. Inoltre, questa soluzione è semplice da implementare e non richiede un clock.

Al fine di implementare le operazioni sopra descritte, il modulo contiene a sua volta dei sottomoduli, i quali compongono gli stadi caratteristici di uno shifter logaritmico in aggiunta ai moduli responsabili per la trasformazione dell'operazione SLL in SRL.

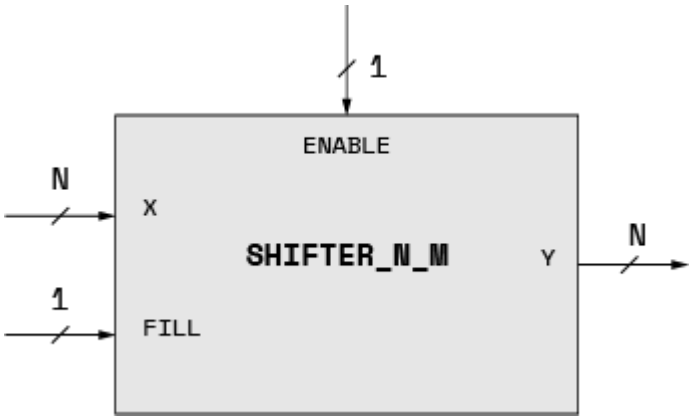
> L'operazione di SLL può essere vista come l'inversione del risultato di un'operazione di SRL su un ingresso a sua volta invertito:
>
> es:
>
> `(0001101 << 2) = 0110100` equivale a:
>
> `flip(flip(0001101) >> 2) = flip(1011000 >> 2) = flip(0010110)= = 0110100`

Per quanto riguarda il collegamento dei vari sottomoduli, si può fare riferimento allo schema a blocchi sottostante:



(Il segnale LEFT viene portato all'ingresso ENABLE dei moduli DRCU).

Modulo SHIFTER_N_M



Il modulo presenta i seguenti segnali:

- X : segnale in ingresso in codifica binaria a N bit, rappresenta l'operando su cui viene eseguita l'operazione di shift;
- Y : segnale in uscita in codifica binaria a N bit, rappresenta il risultato dell'operazione di shift;
- FILL : segnale in ingresso in codifica binaria a 1 bit, rappresenta il valore del (o dei) bit aggiunti durante l'operazione di shift;
- ENABLE : segnale in ingresso in codifica binaria a 1 bit, rappresenta il segnale di abilitazione dell'operazione di shift.

Il modulo esegue un'operazione di shift verso destra di M posizione su un operando a N bit andando a riempire gli M spazi creati con bit del valore dell'ingresso FILL:

$$Y = [FILL\_M\ ;\ (X >> M)]$$

(Si consideri FILL_M come un vettore di M bit di valore FILL)

L'operazione, al fine di essere eseguita, richiede l'abilitazione del modulo ponendo alto il segnale di ENABLE (valore logico 1). In caso di segnale di ENABLE basso, il modulo produrrà sull'uscita Y il valore inalterato dell'ingresso X:

$$Y = X$$

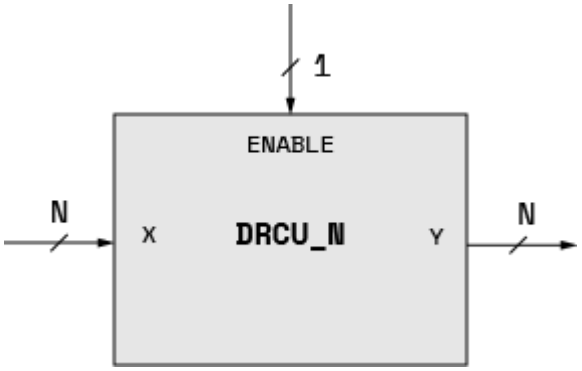
Il comportamento del modulo può essere riassunto come:

```
if ENABLE = 1 then
      Y = [FILL_M ; (X >> M)]
else
      Y = X
end
```

Modulo DRCU_N

Il modulo presenta i seguenti segnali:

- X : segnale in ingresso in codifica binaria a N bit, rappresenta l'operando su cui eseguire l'operazione di inversione;
- Y : segnale in uscita in codifica binaria a N bit, rappresenta il risultato dell'operazione di inversione;
- ENABLE : segnale in ingresso in codifica binaria a 1 bit, rappresenta il segnale di abilitazione dell'operazione di inversione.

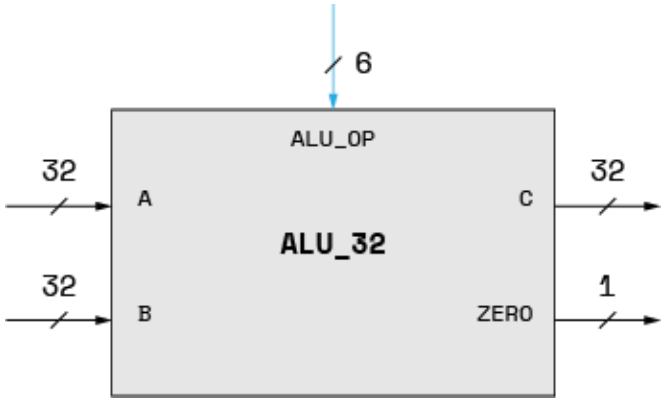


Il modulo DRCU (Data Reversal Control Unit) esegue l'operazione di inversione dell'ordine dei bit su un vettore di dimensione N bit. Se il modulo riceve un segnale di ENABLE basso, produrrà sull'uscita Y il valore inalterato dell'ingresso X.

Il comportamento del modulo può essere riassunto come:

```
if ENABLE = 1 then
      Y[31:0] = X[0:31]
else
      Y = X
end
```

Modulo ALU_32



Il modulo presenta i seguenti segnali:

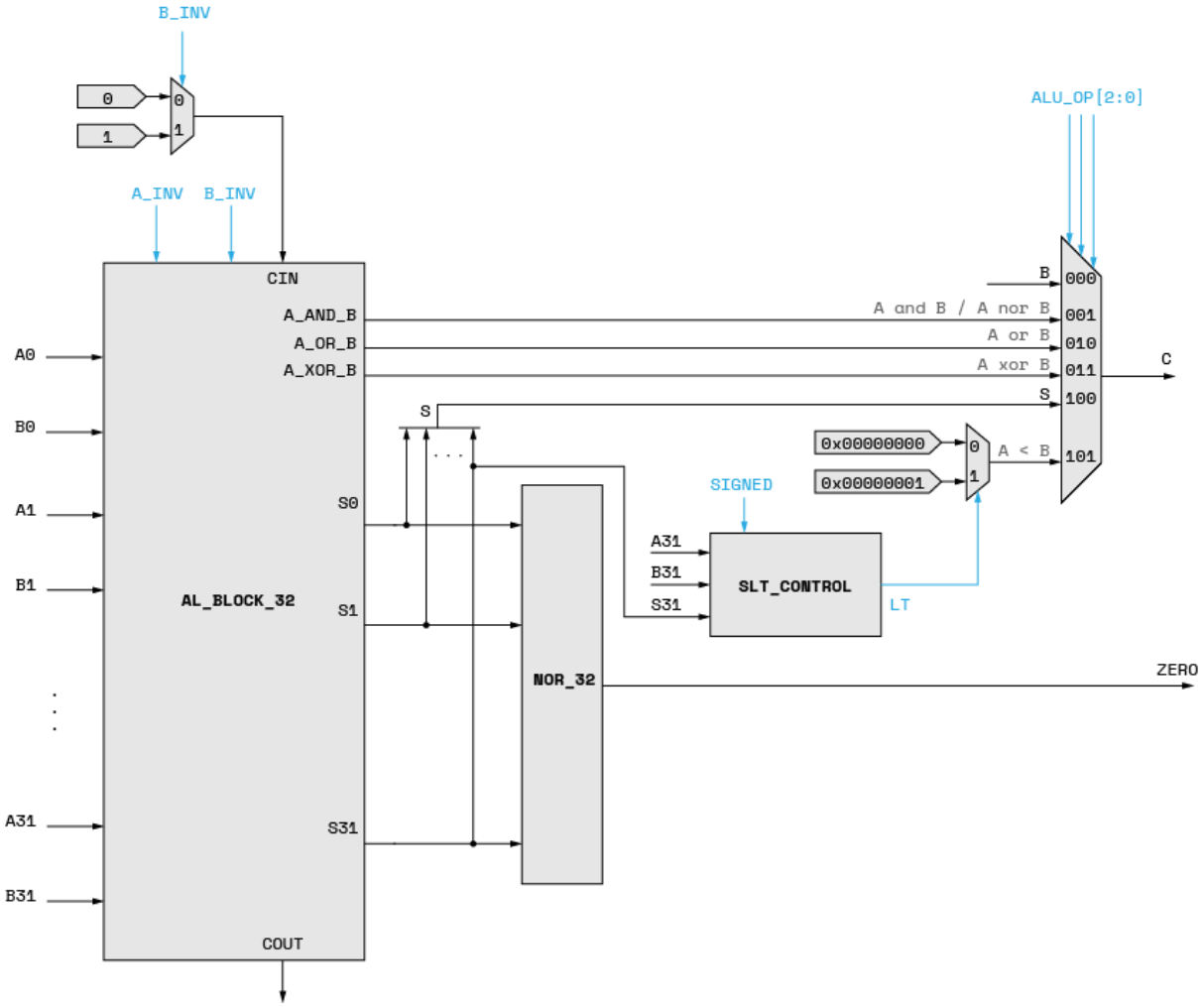
- A : segnale in ingresso in codifica binaria a 32 bit, rappresenta il primo operando della ALU;
- B : segnale in ingresso in codifica binaria a 32 bit, rappresenta il secondo operando della ALU;
- `ALU_OP` : segnale di controllo in ingresso in codifica binaria a 6 bit, rappresenta l'insieme di altri 4 segnali per la selezione delle operazioni interne alla ALU:
  - `ALU_OP[5]` = `A_INV` : segnale di complemento a 1 sull'operando A;
  - `ALU_OP[4]` = `B_INV` : segnale di complemento a 1 sull'operando B (abilita anche il complemento a 2 nella somma);
  - `ALU_OP[3]` = `SIGNED` : segnale usato per differenziare tra operazioni signed e unsigned;
  - `ALU_OP[2:0]` : segnali utilizzati per selezionare il risultato in uscita alla ALU.
- `C` : segnale in uscita in codifica binaria a 32 bit, rappresenta il risultato dell'operazione scelta;
- `ZERO` : segnale in uscita in codifica binaria a 1 bit, segnale che l'ultima operazione ha prodotto come risultato `0` (codificato a 32 bit).

Le operazioni richieste dall'instruction set sono le seguenti:

- Operazioni bitwise tra gli operandi:
  - `A and B`;
  - `A or B`;
  - `A xor B`;
  - `A nor B`;
- Operazioni aritmetiche:
  - `A + B`;
  - `A - B`.
- Altre operazioni:
  - `C = B`;
  - `C = 1 if A < B else C = 0`.

Le operazioni devono poter trattare sia su operandi in codifica binaria `signed`, che `unsigned`.

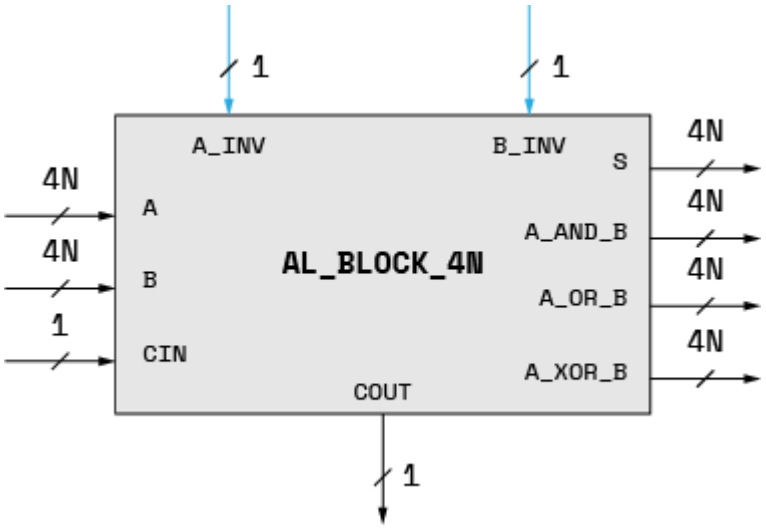
La ALU è un modulo complesso che contiene diversi blocchi logici come mostrato nella figura riportata di seguito:



Il sottomodulo principale è `AL_BLOCK_4N` (in questo caso particolare `AL_BLOCK_32`), mentre gli altri blocchi rappresentano delle suddivisioni logiche del resto del circuito.

Sebbene non rappresenti un modulo a sé stante, il blocco `SLT_CONTROL` svolge una funzione importante all'interno della ALU, pertanto merita di essere accennato: si tratta di un piccolo circuito combinatorio (sintetizzato direttamente in RTL) responsabile del calcolo del segnale di controllo `LT` (less than) interno alla ALU. Tale segnale permette di svolgere i confronti tra gli operandi A e B sia in versione signed che unsigned sulla base del valore dei bit più significativi degli operandi e del bit più significativo del risultato dell'operazione $S = A - B$.

Modulo AL_BLOCK_4N



Il modulo presenta i seguenti segnali:

- A : segnale in ingresso in codifica binaria di dimensione multipla di 4 bit (4 x N), rappresenta il primo operando;
- B : segnale in ingresso in codifica binaria di dimensione multipla di 4 bit (4 x N), rappresenta il secondo operando;
- CIN : segnale in ingresso in codifica binaria a 1 bit, rappresenta il riporto in ingresso all'operazione di somma;
- A_INV : segnale di controllo in ingresso in codifica binaria a 1 bit, segnala al modulo di trasformare A nel suo complemento a 1;
- B_INV : segnale di controllo in ingresso in codifica binaria a 1 bit, segnala al modulo di trasformare B nel suo complemento a 1;
- COUT : segnale in uscita in codifica binaria a 1 bit, rappresenta il riporto in overflow prodotto dall'operazione di somma;
- S : segnale in uscita in codifica binaria a 4 x N bit, rappresenta il risultato dell'operazione di somma;
- A_AND_B : segnale in uscita in codifica binaria a 4 x N bit, rappresenta il risultato dell'and bitwise tra gli operandi;
- A_OR_B : segnale in uscita in codifica binaria a 4 x N bit, rappresenta il risultato dell'or bitwise tra gli operandi;
- A_XOR_B : segnale in uscita in codifica binaria a 4 x N bit, rappresenta il risultato dello xor bitwise tra gli operandi.

Il modulo compie contemporaneamente le seguenti operazioni di base:

- `A and B`;
- `A or B`;
- `A xor B`;
- `A + B`.

Le operazioni riportate sopra possono essere alterate tramite i segnali A_INV e B_INV per produrre le seguenti operazioni:

Operazione	**A_INV**	**B_INV**	**Uscita**
A nor B	1	1	A_AND_B
A nand B	1	1	A_OR_B
A xnor B	1 (0)	0 (1)	A_XOR_B
A - B (con CIN = 1)	0	1	S

(Si noti che ai fini dell'implementazione dell'instruction set, le operazioni di NAND e XNOR bitwise non sono richieste, bensì sono una conseguenza diretta di come è stata progettata l'architettura).

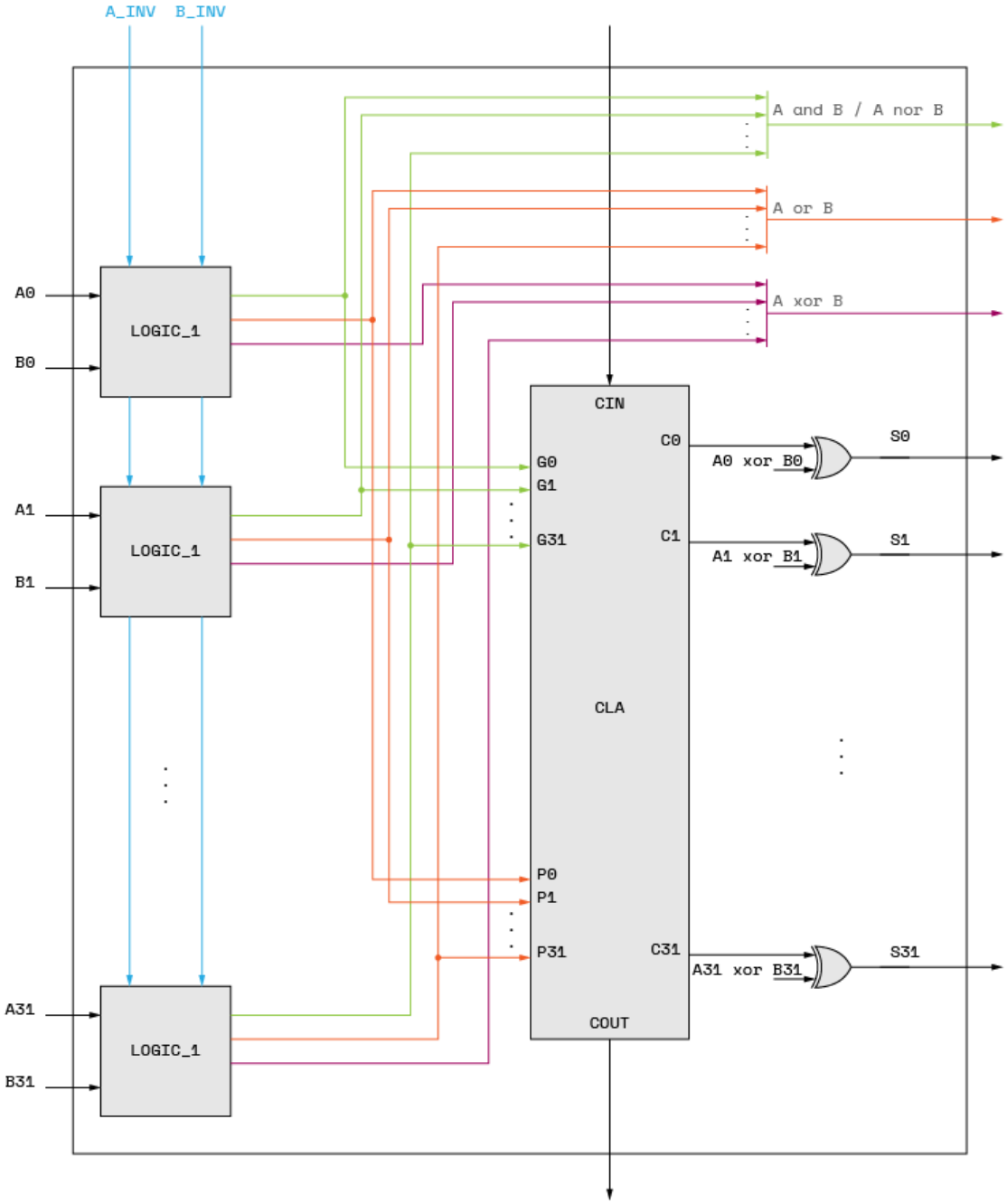
Per l'architettura del sommatore sono state fatte alcune considerazioni in termini di velocità e modularità:

- Un RCA (Ripple Carry Adder), sebbene semplice da implementare e facilmente modularizzabile, avrebbe introdotto un ritardo eccessivo.
- Un CLA (Carry Look-Ahead) a 32 bit, sebbene veloce, sarebbe stato troppo complicato da implementare per intero per via del numero e della dimensione delle equazioni per descrivere i riporti.

Per ottenere un equilibrio tra due soluzioni, è stata adottata un'architettura ibrida composta da moduli CLA in cascata. Questo approccio garantisce una velocità inferiore rispetto a un semplice CLA a 32 bit, ma comunque più rapida di un RCA. Inoltre, consente di sintetizzare moduli CLA più gestibili in termini di complessità delle equazioni.

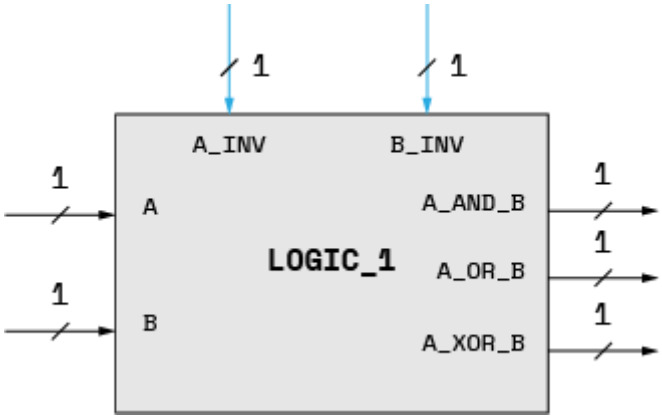
Come granularità dei moduli CLA sono stati scelti 4 bit, pertanto il sommatore a 32 bit è composto da un totale di 8 moduli CLA in cascata.

Come accennato in precedenza, il modulo è composto a sua volta da altri sottomoduli, in particolare: `CLA_4` e `LOGIC_1`.



(Lo schema riporta, in particolare, lo schema logico di un blocco a 32 bit. Si noti che il blocco `CLA` sarà quindi composto da N = 8 moduli `CLA_4` in cascata)

Modulo LOGIC_1



Il modulo presenta i seguenti segnali:

- A : segnale in ingresso in codifica binaria a 1 bit, rappresenta il primo operando;
- B : segnale in ingresso in codifica binaria a 1 bit, rappresenta il secondo operando;
- `A_AND_B` : segnale in uscita in codifica binaria a 1 bit, rappresenta il risultato dell'and bitwise tra gli operandi;
- `A_OR_B` : segnale in uscita in codifica binaria a 1 bit, rappresenta il risultato dell'or bitwise tra gli operandi;
- `A_XOR_B` : segnale in uscita in codifica binaria a 1 bit, rappresenta il risultato dello xor bitwise tra gli operandi;
- `A_INV` : segnale di controllo in ingresso in codifica binaria a 1 bit, segnala al modulo di trasformare A nel suo complemento a 1;
- `B_INV` : segnale di controllo in ingresso in codifica binaria a 1 bit, segnala al modulo di trasformare B nel suo complemento a 1.

Questo semplice modulo permette di compiere tutte le operazioni logiche (bitwise) come descritte nel modulo `AL_BLOCK_4N`.

Le operazioni possono essere riassunte come:

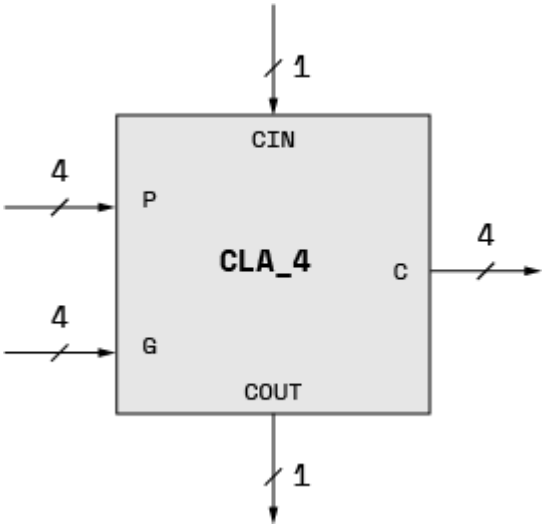
Siano $A' = A \ if \ A\_INV \ = \ 0 \ else \ \neg A$ e $B' = B \ if \ B\_INV \ = \ 0 \ else \ \neg B$,

- $A\_AND\_B = A' \ \& \ B'$
- $A\_OR\_B = A' \mid B'$
- $A\_XOR\_B = A' \ \hat{} \ B'$.

Modulo CLA_4

Il modulo presenta i seguenti segnali:

- `CIN` : segnale in ingresso in codifica binaria a 1 bit, rappresenta il riporto in ingresso all'operazione di somma;
- `P` : segnale in ingresso in codifica binaria a 4 bit, rappresenta i termini di propagazione per il calcolo dei riporti;
- `G` : segnale in ingresso in codifica binaria a 4 bit, rappresenta i termini di generazione per il calcolo dei riporti;
- `C` : segnale in uscita in codifica binaria a 4 bit, rappresenta i riporti calcolati per la somma;
- `COUT` : segnale in uscita in codifica binaria a 1 bit, rappresenta l'ultimo riporto.



Il modulo calcola i riporti per la somma a 4 bit secondo la metodologia Carry Look-Ahead. In generale il riporto n-esimo viene calcolato come:

$$C_n = \sum_{i=0}^{n-1} \quad G_i \prod_{j=i+1}^{n-1} \quad P_j + C_0 \prod_{i=0}^{n-1} \quad P_i$$

Per cui i riporti a 4 bit sono:

$$C_0 = CIN$$

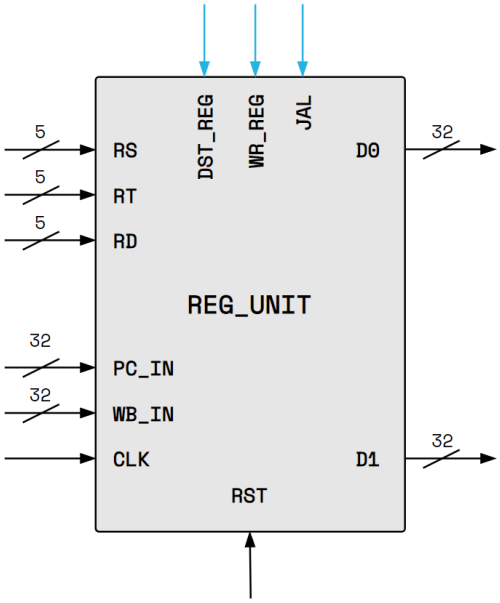
$$C_1 = G_0 + CIN\ P_0$$

$$C_2 = G_1 + G_0P_1 + CIN\ P_0P_1$$

$$C_3 = G_2 + G_1P_2 + G_0P_1P_2 + CIN\ P_0P_1P_2$$

$$COUT = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + CIN\ P_0P_1P_2P_3$$

Modulo REG_UNIT

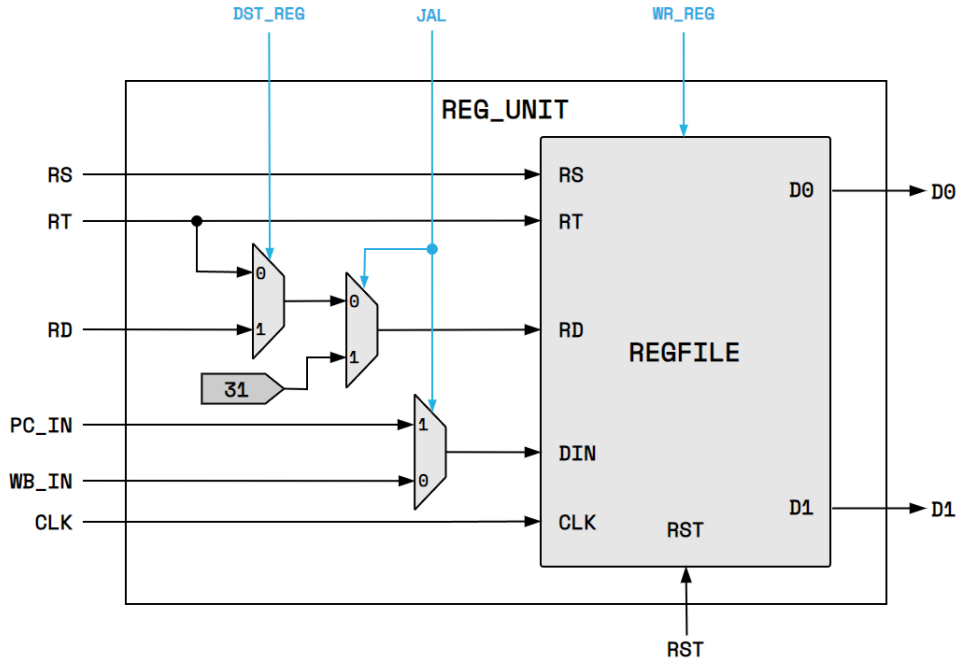


Il modulo presenta i seguenti segnali:

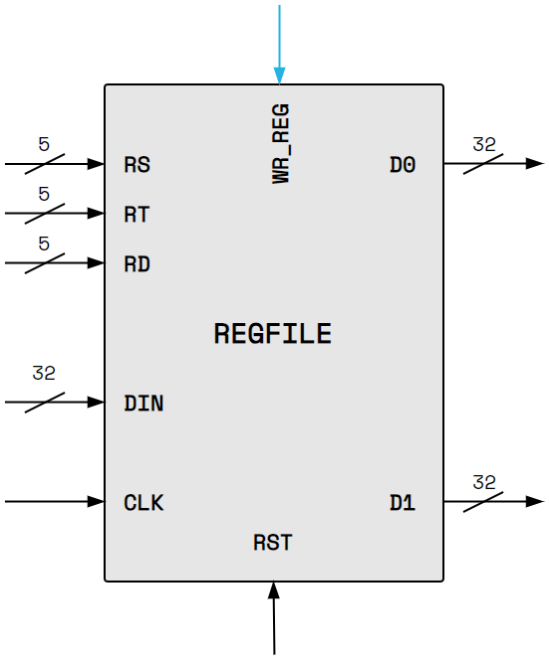
- RS: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il primo registro sorgente;
- RT: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il secondo registro sorgente/destinazione;
- RD: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il terzo registro destinazione;
- PC_IN: Segnale in ingresso in codifica binaria a 32 bit. Rappresenta il valore in ingresso dal registro PC (Program Counter) per le istruzioni che lo richiedono;
- WB_IN: Segnale in ingresso in codifica binaria a 32 bit. Rappresenta il valore in ingresso dalla fase di Write-Back;
- D0: Segnale in uscita in codifica binaria a 32 bit. Rappresenta il valore contenuto nel primo registro sorgente;
- D1: Segnale in uscita in codifica binaria a 32 bit. Rappresenta il valore contenuto nel secondo registro sorgente;
- DST_REG: Segnale di controllo in ingresso in codifica binaria ad 1 bit. Segnala al modulo quale registro tra RT e RD selezionare come registro destinazione;
- WR_REG: Segnale di controllo in ingresso in codifica binaria ad 1 bit. Segnala al modulo di scrivere nel registro destinazione;
- JAL: Segnale di controllo in ingresso in codifica binaria ad 1 bit. Segnala al modulo che un'istruzione JAL è in esecuzione;
- CLK: Segnale di clock in ingresso;
- RST: Segnale di reset sincrono in ingresso.

Questo modulo serve a incapsulare il vero e proprio Register File del processore e la logica che seleziona i registri e i dati in scrittura dai segnali in ingresso.

Internamente il modulo si presenta nel seguente modo:



Modulo REGFILE



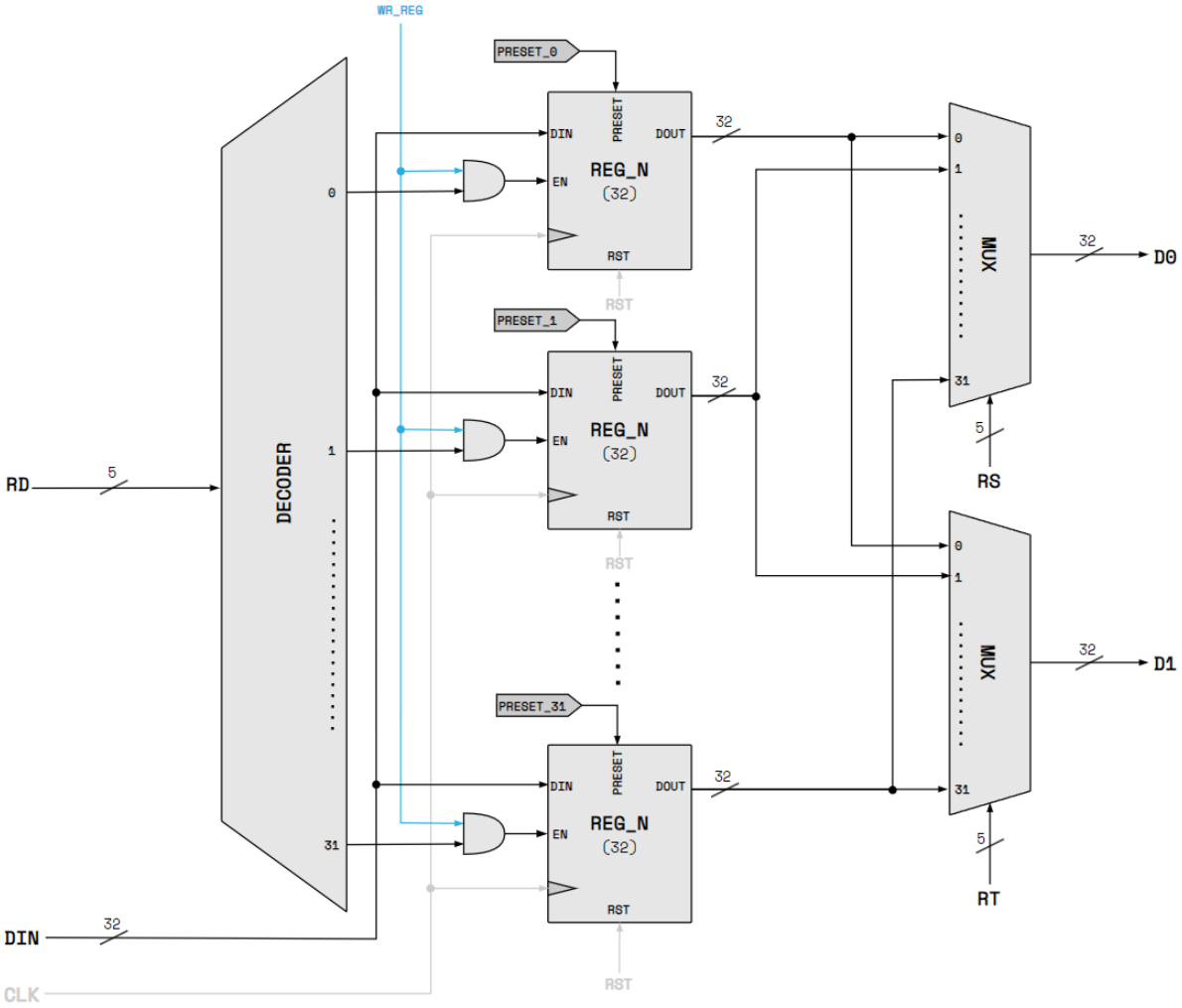
Il modulo presenta i seguenti segnali:

- RS: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il primo registro sorgente;
- RT: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il secondo registro sorgente;
- RD: Segnale in ingresso in codifica binaria a 5 bit. Rappresenta il registro destinazione;
- DIN: Segnale in ingresso in codifica binaria a 32 bit. Rappresenta il valore in ingresso da salvare;
- D0: Segnale in uscita in codifica binaria a 32 bit. Rappresenta il valore contenuto nel primo registro sorgente;
- D1: Segnale in uscita in codifica binaria a 32 bit. Rappresenta il valore contenuto nel secondo registro sorgente;
- WR_REG: Segnale di controllo in ingresso in codifica binaria ad 1 bit. Segnala al modulo di scrivere nel registro destinazione;
- CLK: Segnale di clock in ingresso;
- RST: Segnale di reset sincrono in ingresso.

Questo modulo contiene 32 registri da 32 bit. Si tratta di un register file orientato all'utilizzo con una ALU a 2 operandi; infatti, permette di leggere contemporaneamente il contenuto dei registri indirizzati dai segnali RS e RT rispettivamente sulle uscite D0 e D1. La scrittura avviene sul fronte di salita del segnale di clock quando il segnale di controllo WR_REG è alto. Il registro di destinazione per la scrittura è indirizzato dal segnale RD e il valore da scrivere è rappresentato dal segnale DIN.

Il segnale di reset (RST) è sincrono e attivo alto sul fronte di salita del clock.

Internamente il modulo si presenta in questo modo:

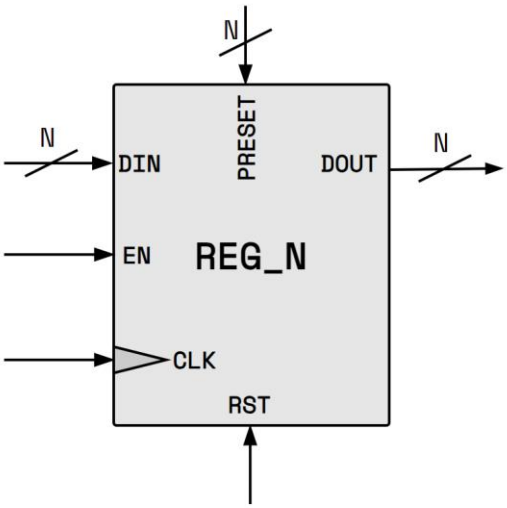


Il decoder e i due multiplexer sono realizzati in logica standard RTL. I valori di preset per ogni registro sono i seguenti:

Registro	Valore di PRESET
0 - 27	0X00000000
28	0X00000300
29	0X000003FC
30	0x00000000
31	0x00000080

Modulo REG_N

Il modulo presenta i seguenti segnali:



- DIN: Segnale in ingresso in codifica binaria a N bit. Rappresenta il valore in ingresso da salvare;
- EN: Segnale di controllo in ingresso in codifica binaria a 1 bit. Abilita la scrittura nel registro;
- DOUT: Segnale in uscita in codifica binaria a N bit. Rappresenta il valore salvato nel registro;
- PRESET: Segnale in ingresso in codifica binaria a N bit. Rappresenta il valore di default salvato nel registro;
- CLK: Segnale di clock in ingresso;
- RST: Segnale di reset sincrono in ingresso.

Si tratta di un registro a N bit attivo sul fronte di salita e con reset sincrono e attivo alto sul fronte di salita del clock. Quando il reset è attivo, al fronte di salita successivo del clock, il valore salvato nel registro e di conseguenza DOUT assumono il valore del segnale PRESET. Durante il normale utilizzo, DOUT assume il valore di DIN ogniqualvolta che si verifica un fronte di salita nel clock e il segnale EN è alto. Nel caso EN fosse basso, DOUT mantiene il suo valore costante.

Riuso dei moduli

Durante la progettazione dell'architettura è stata posta particolare enfasi sul riuso dei sottomoduli del sistema, cercando di generalizzare quanti più componenti possibili.

I moduli soggetti a riuso sono, in particolare:

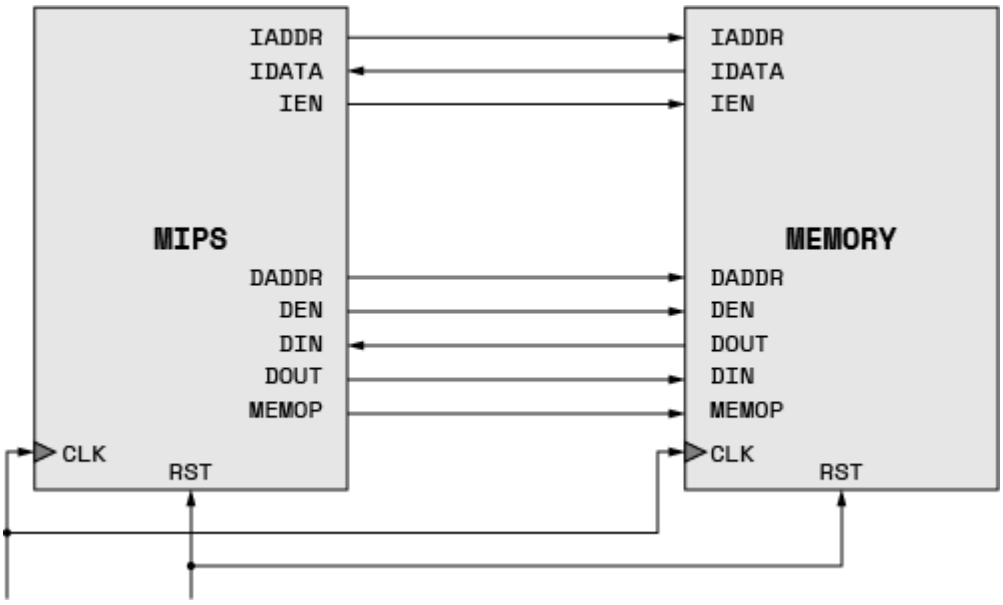
- AL_BLOCK_4N: il modulo viene utilizzato ovunque sia necessaria un'operazione di somma, ed è presente sia nel modulo ALU che nel modulo MANAGEMENT_PC, sostituendo i sommatori visibili nell'architettura iniziale per il calcolo dei salti e degli incrementi del PC. Questo modulo, come precedentemente discusso in dettaglio nelle sezioni dedicate, è costituito dai moduli CLA_4 e LOGIC_1, che sono anch'essi progettati per consentire il riutilizzo e semplificare l'implementazione dell'architettura.
- CLA_4: il modulo si presta per essere riutilizzato ogni volta sia necessario eseguire un'operazione di somma. La dimensione (a 4 bit) permette di adattarsi facilmente a ingressi di dimensioni differenti. Anche in caso di ingressi di dimensione non multipla di 4, il modulo può essere tranquillamente usato ignorando alcune delle uscite.
- LOGIC_1: il modulo è stato pensato per contenere al suo interno tutte le operazioni logiche bitwise ed è particolarmente utile nei casi in cui sia necessario utilizzare più risultati contemporaneamente (come nel caso dei termini di propagazione e di generazione del CLA).
- REG_N: il modulo viene impiegato per l'implementazione di tutti i registri dell'architettura, compresi quelli interni al register file e il PC.
- SHIFTER_N_M: Il modulo è utilizzato per costituire gli stadi interni dello shifter logaritmico, ma è anche impiegato per implementare la funzione di sign extend. In questo caso, il sign extend viene trattato come uno shift fisso di 16 bit verso destra. L'implementazione del sign extend attraverso questo modulo beneficia del fatto che non tutte le operazioni richiedono l'estensione del segno, ma solo un riempimento di '0'. Dal punto di vista dello shifter non comporta differenze, ciò che cambia sarà semplicemente il valore di FILL determinato mediante gli opportuni segnali della CU in ingresso al modulo.
- DRCU_N: il modulo viene utilizzato dallo shifter per compiere l'operazione di "flip" dei bit, la quale permette allo shifter stesso di eseguire sia operazioni di shift a destra che a sinistra secondo le considerazioni fatte nella relativa sezione.
- MUXN: il modulo nell'architettura trova uso solo nel register file, ma si presta per essere riutilizzato per un qualsiasi multiplexer a 32 ingressi.

Verifica

Il test complessivo per testare il corretto funzionamento del MIPS si è svolto nel seguente modo:

- Prima sono stati testati i singoli componenti con test-bench indipendenti;
- Successivamente sono stati realizzati più test-bench in behavioral in cui sono state testate le singole istruzioni inserendole in input all'architettura top-level del MIPS;
- Infine abbiamo realizzato un programma completo con ricorsione e testato in Post Place and Route caricando il programma stesso nella memoria RAM così che il MIPS caricasse le istruzioni tramite il PC dalla memoria;

Struttura top-level su cui abbiamo fatto i test-bench finali e completi:



Per verificare la correttezza dei test abbiamo confrontato lo stato del MIPS con l'output della simulazione con il software Mars per ogni istruzione testata. Abbiamo identificato come stato del MIPS i seguenti elementi: valore dell'indirizzo (ADDR) ad ogni clock, lo stato della memoria (valori memorizzati al suo interno) e lo stato dei registri.

In generale il processo per testare il programma completo è stato quello di scrivere il codice in assembly nel simulatore Mars, successivamente abbiamo esportato le codifiche delle istruzioni in binario per poi inserirle all'interno della RAM nella sezione TEXT partendo all'indirizzo esadecimale `0x00000080`.

Test-bench

Il test più completo che abbiamo fatto è nella seguente pagina. In questo test abbiamo voluto verificare il funzionamento dei salti condizionati (e non) ed in particolare l'esatta esecuzione della ricorsione con lettura e scrittura in memoria RAM.

Il test bench non fa altro che simulare l'evoluzione temporale dei segnali di clock e di reset, gli altri segnali di ingresso e d'uscita sono determinati dall'evoluzione del sistema; in particolare, il programma è stato testato con RST alto per `150ns` e CLK con periodo `200ns`.

Il programma testato è il seguente:

```
main:   addi    $a0, $a0, 2
        jal     rec
        addi    $s0, $v0, 0
        nop
        nop
        nop
        nop
        nop
rec:    addi    $sp, $sp, -20
        sw      $fp, 8($sp)
        addi    $fp, $sp, 8
        sw      $a0, 8($fp)
        sw      $ra, 4($fp)
if:     bne     $a0, $zero, else
then:   addi    $t0, $t0, 1
        sw      $t0, -8($fp)
        addi    $t1, $t1, 0
        beq     $t1, 0, ret
else:   addi    $a0, $a0, -1
        sw      $a0, -4($fp)
        lw      $a0, -4($fp)
        jal     rec
        lw      $t0, 8($fp)
        add     $t0, $t0, $v0
        sw      $t0, -8($fp)
ret:    lw      $v0, -8($fp)
        lw      $ra, 4($fp)
        lw      $fp, 0($fp)
        addi    $sp, $sp, 20
        jr      $ra
```

Casi d'uso

Il test bench, come descritto in precedenza, riproduce il comportamento del microprocessore e della memoria dal momento del loro avvio fino alla terminazione di un programma complesso che interagisce attivamente con la memoria.

Siccome il test bench simula unicamente l'evoluzione del clock (e inizialmente del reset), nella realtà il processore proseguirà a leggere i contenuti di memoria puntati dal PC, il che può portare a diverse situazioni:

- lettura completa del contenuto in memoria (fino all'ultimo indirizzo disponibile);
- loop di sezioni di codice;
- comportamenti inaspettati dovuti a quanto detto sopra, come il salto in sezioni di memoria al di fuori di TEXT a seguito di istruzioni come "`jr $ra`" con "`$ra = 0`".