Esercitazione 06

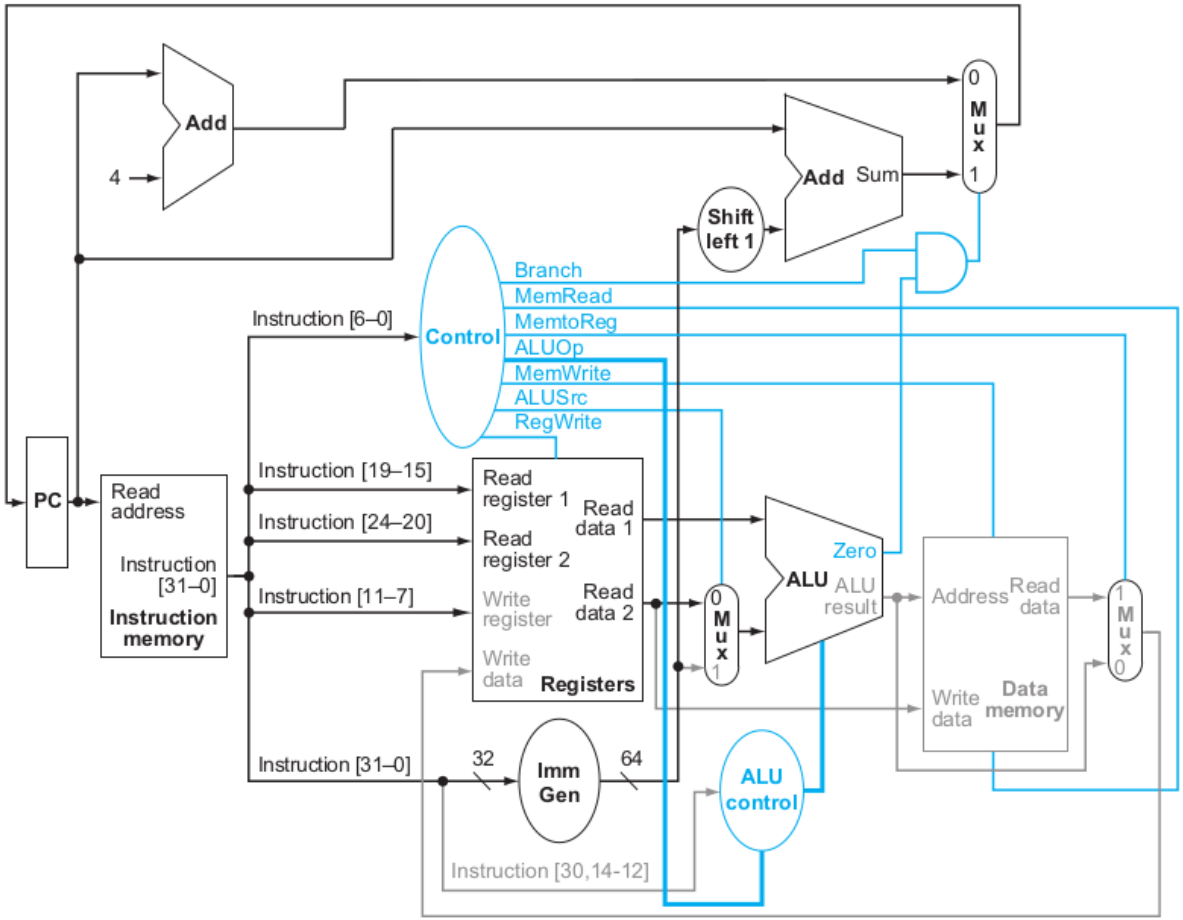
Progettazione di una CPU RISC-V

Gianluca Brilli
gianluca.brilli@unimore.it

Overview

- In questa esercitazione andremo a progettare una semplice architettura di esempio, il cui **instruction set** (ISA) è composto dalle seguenti istruzioni:
- **Istruzioni di Memoria**: operanti su byte, in particolare load/store byte (*lb* e *sb*);
- **Istruzioni ALU**: and, or, addizione e sottrazione (*and*, *or*, *add*, *sub*);
- **Istruzioni di Salto**: per il controllo del flusso del programma, nello specifico, branch if equal (*beq*).

Overview


Add
4
Add
Sum
Shift left 1
0
M
u
x
1
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [6–0]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [19–15]
Instruction [24–20]
Instruction [11–7]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
Instruction [31–0]
32
Imm Gen
64
ALU control
Instruction [30,14-12]

Codifica delle Istruzioni (1)

- Seguiamo le specifiche di codifica del RISC-V:

Formato	Istruzione	Opcode	funct3	funct7	ALU Action	ALU Control
R-Type	add	10	00	00	add	10
	sub	10	00	01	sub	11
	and	10	11	00	and	00
	or	10	10	00	or	01
SB-Type	beq	01	xx	xx	sub	11
I-Type	lb	00	xx	xx	add	10
S-Type	sb	11	xx	xx	add	10

Codifica delle Istruzioni (2)

- Manteniamo uno standard simile alla codifica delle istruzioni vista in classe per l'architettura RISC-V a 64 bit. Nel nostro caso riduciamo la codifica a soli 16 bit di istruzioni.

- Istruzioni I-Type:

15		…		*0*
Imm[7:0]	**rs1**	**funct3**	**rd**	**op**
8bit	*2bit*	*2bit*	*2bit*	*2bit*

Codifica delle Istruzioni (3)

- Istruzioni S-Type:

15		…			0
Imm[7:2]	**rs2**	**rs1**	**funct3**	**Imm[1:0]**	**op**
6bit	*2bit*	*2bit*	*2bit*	*2bit*	*2bit*

- Istruzioni SB-Type:

15		…			0
Imm[0, 7, 5:2]	**rs2**	**rs1**	**funct3**	**Imm[1, 6]**	**op**
6bit	*2bit*	*2bit*	*2bit*	*2bit*	*2bit*

- Istruzioni R-Type:

15		…				0
xxxx	**funct7**	**rs2**	**rs1**	**funct3**	**rd**	**op**
4bit	*2bit*	*2bit*	*2bit*	*2bit*	*2bit*	*2bit*

Parte 01 - Program Counter

- In questa implementazione prevediamo l'utilizzo di due differenti zone di memoria: la **instruction memory** e la **data memory**.
- Realizzare secondo il precedente schema, la circuiteria di gestione del program counter (PC), in modo tale che venga incrementato ad ogni ciclo di clock;
- Inserire inoltre la possibilità di incrementi maggiori di 1 (branch).

Parte 02 – connessione Regfile / ALU

- **Requisiti da soddisfare:**
- **1)** Prevedere la possibilità di sommare il valore di un registro con un immediato e di andare in memoria all'indirizzo calcolato dall'ALU.
- Esempio:
  - sb X1, **4(X3)** → addr = X3 + 4
    → mem[addr]

Secondo operando, necessario sommare un offset e andare in memoria. Connettere il risultato della somma al BUS Indirizzi.

Parte 02 – connessione Regfile / ALU

- **Requisiti da soddisfare:**
- **2)** Prevedere la possibilità di scrivere in memoria il valore contenuto all'interno di un registro.
- Esempio:
  - sb **X1**, 4(X3) → mem[addr] = X1

Dopo aver calcolato l'indirizzo su cui andare a scrivere, è necessario scrivere il valore contenuto nel registro. Connettere il primo registro al BUS Dati.

Parte 02 – connessione Regfile / ALU

- **Requisiti da soddisfare:**
- **3)** Prevedere la possibilità di scrivere in un registro il risultato di un'operazione ALU.
- Esempio:
  - add **X1**, X2, X3

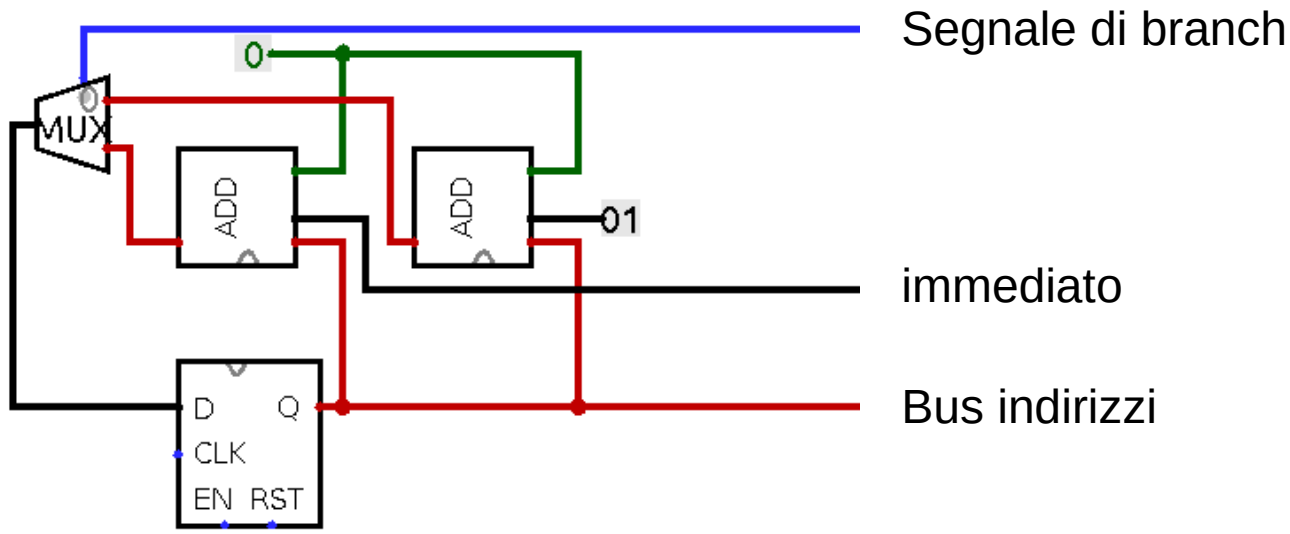
Dopo l'operazione ALU, viene scritto il valore della somma all'interno del Regfile.

Parte 02 – connessione Regfile / ALU

- **Requisiti da soddisfare:**
- **4)** Prevedere la possibilità di leggere un byte dalla memoria e caricarlo in un registro.
- Esempio:
  - lb **X1**, 0(X3)

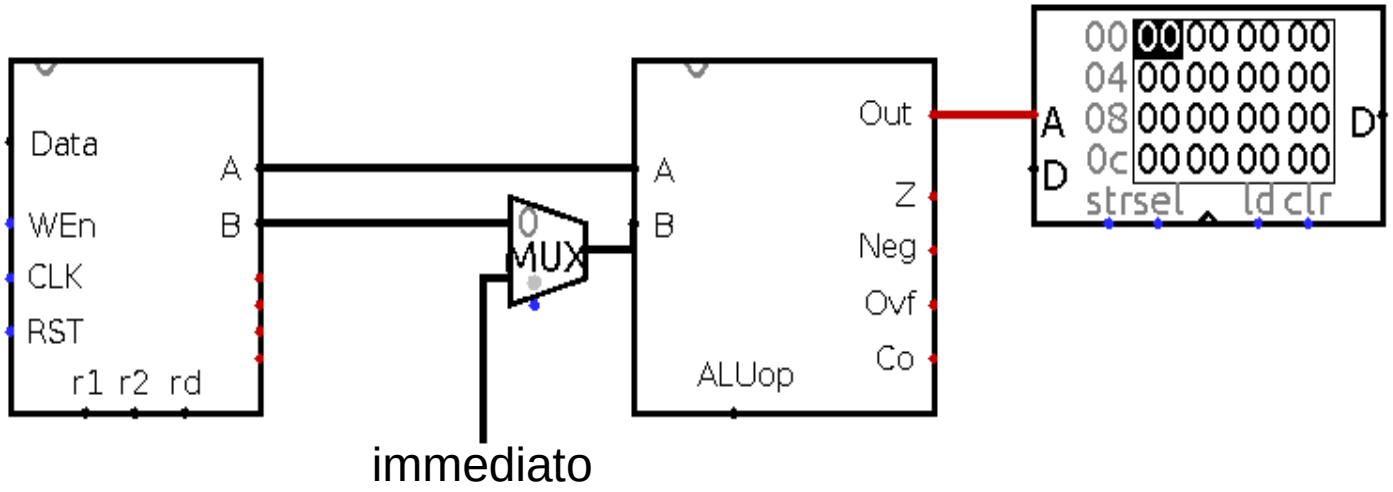
Dopo il calcolo dell'indirizzo (analogo alla store, vista prima), viene letto un dato dalla memoria e scritto all'interno del Regfile.

Parte 01 - circuito



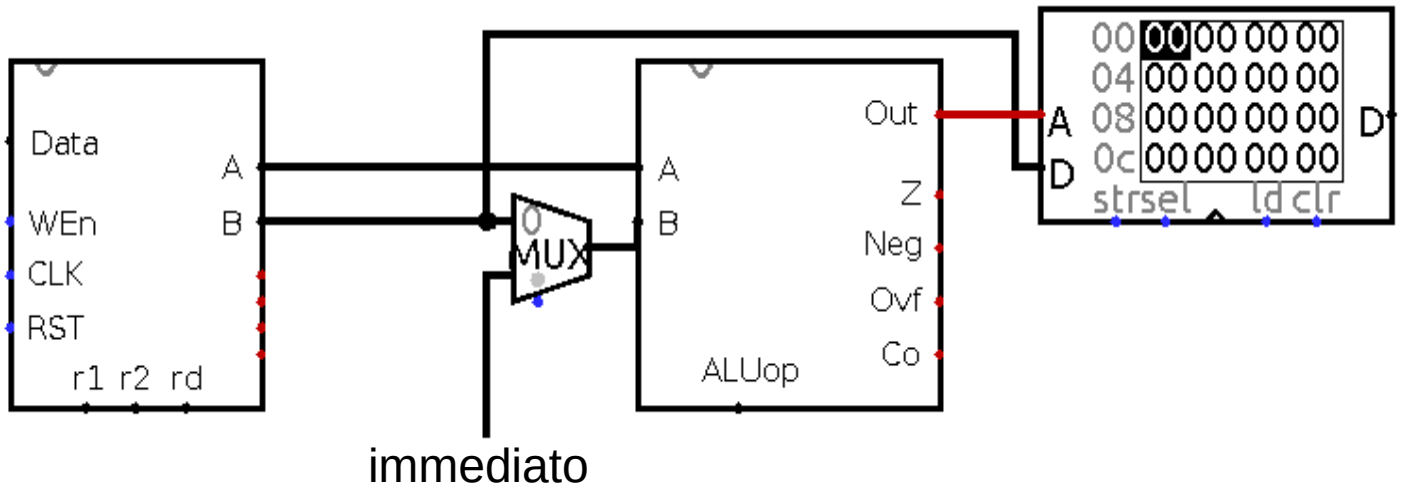
- l'adder a sinistra somma l'uscita del PC con gli immediati (*beq x1, x2, imm*).
- l'adder a destra gestisce il normale incremento del PC ad ogni ciclo di clock.

Parte 02 - circuito



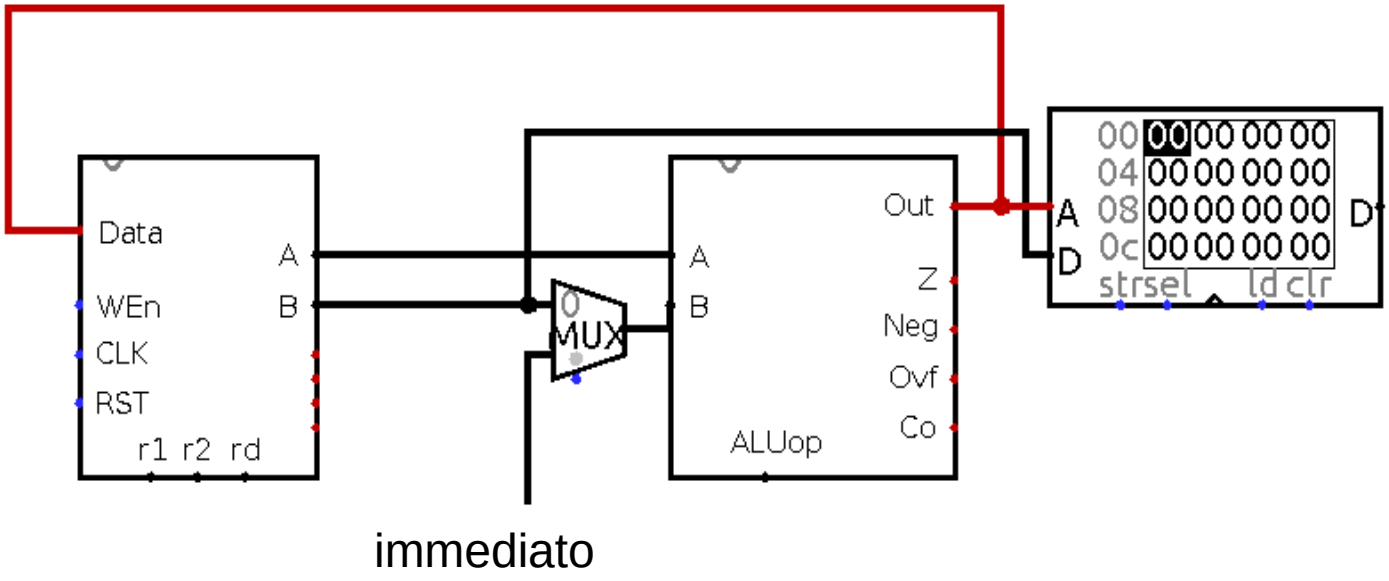
- **1)** Somma tra immediato (operando B) e valore contenuto in un registro (operando A) e accesso in memoria all'indirizzo calcolato dall'ALU.

Parte 02 - circuito



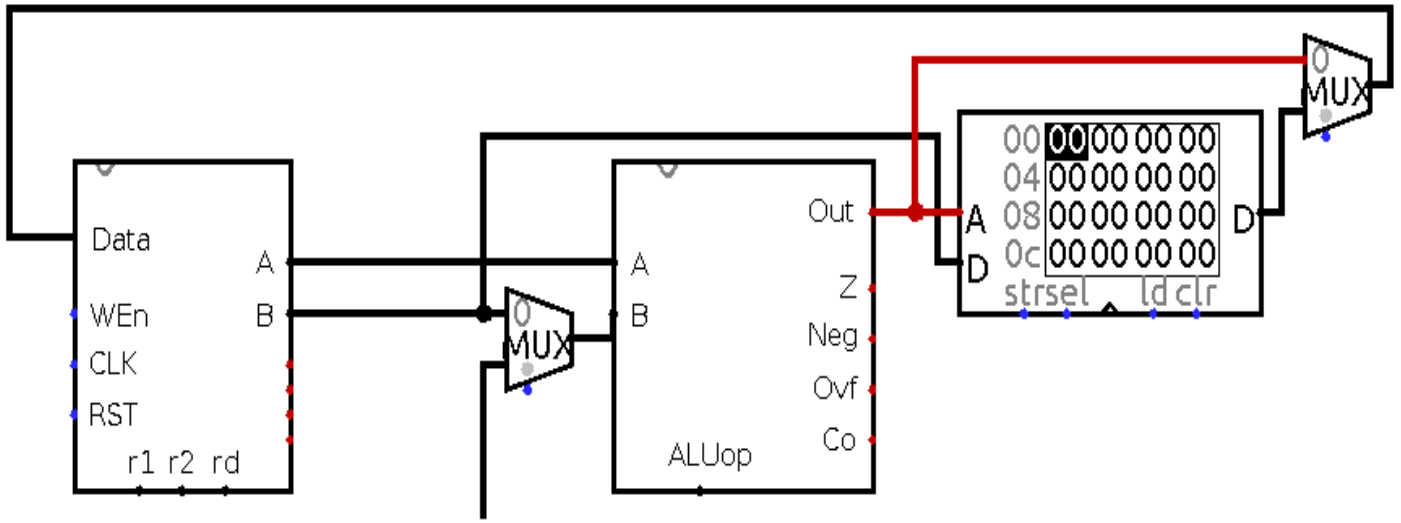
› **2)** Scrittura in memoria del contenuto di un registro.

Parte 02 - circuito



› **3)** Scrittura all'interno del Regfile il risultato di un'operazione ALU.

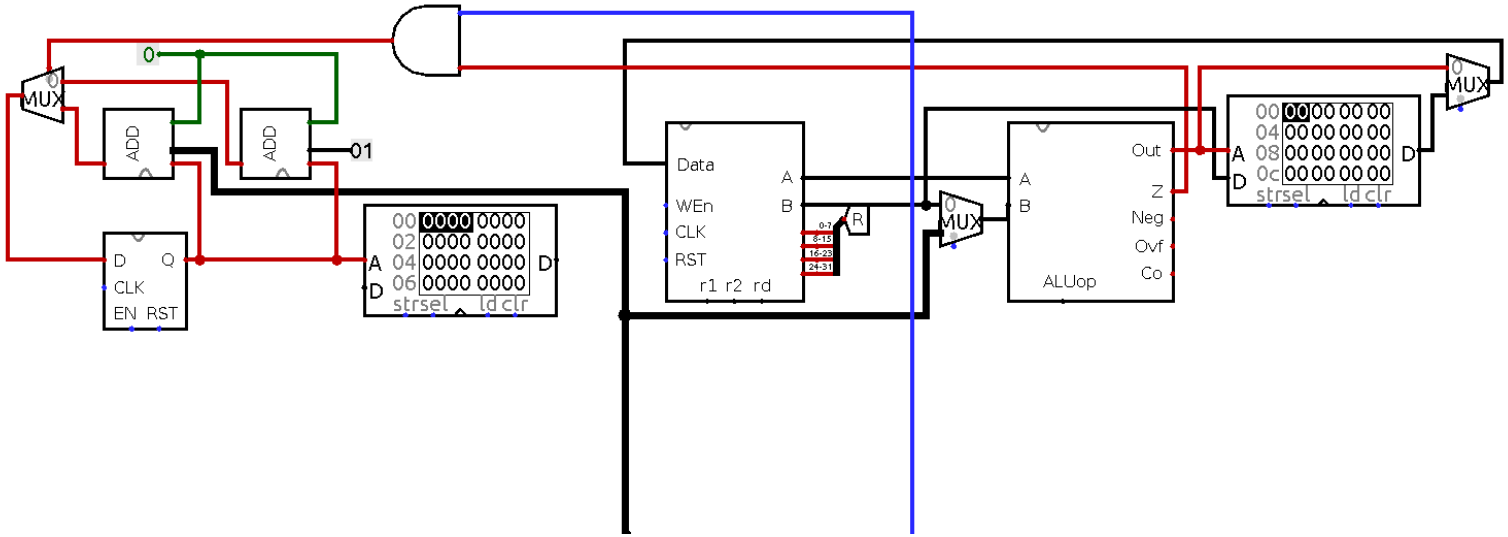
Parte 02 - circuito



› **4)** Lettura di un byte dalla memoria e scrittura all'interno del Regfile.

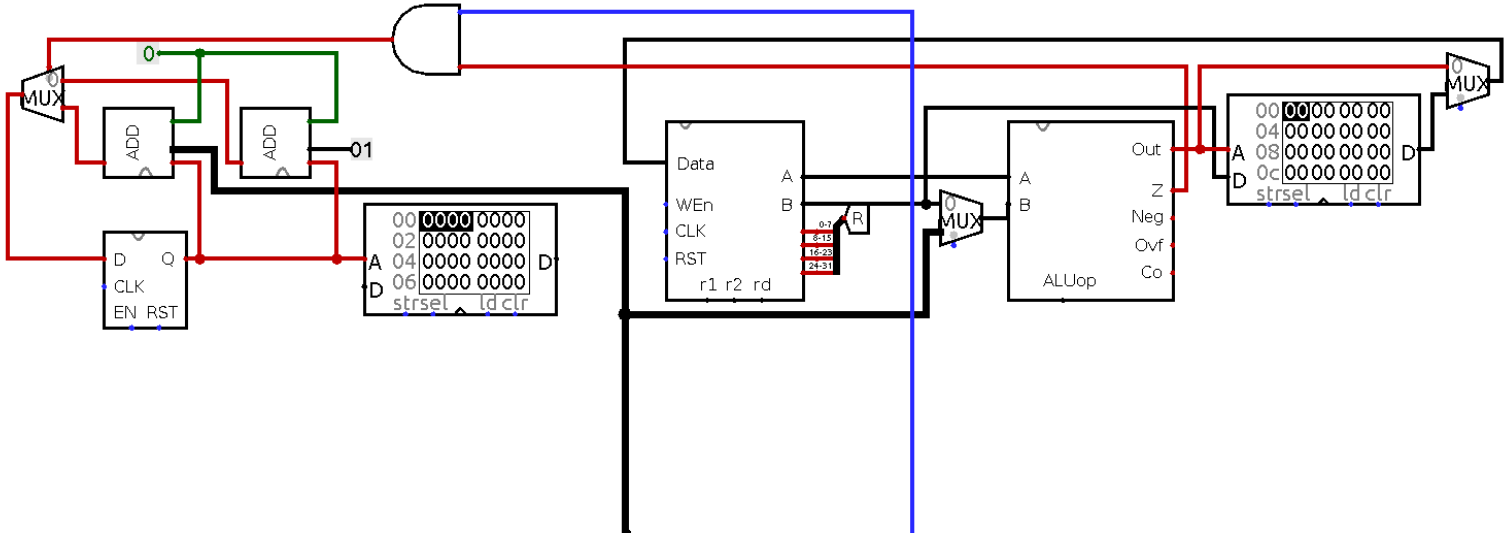
Parte 03 - connessioni

- Andiamo a connettere tra di loro i due circuiti realizzati.



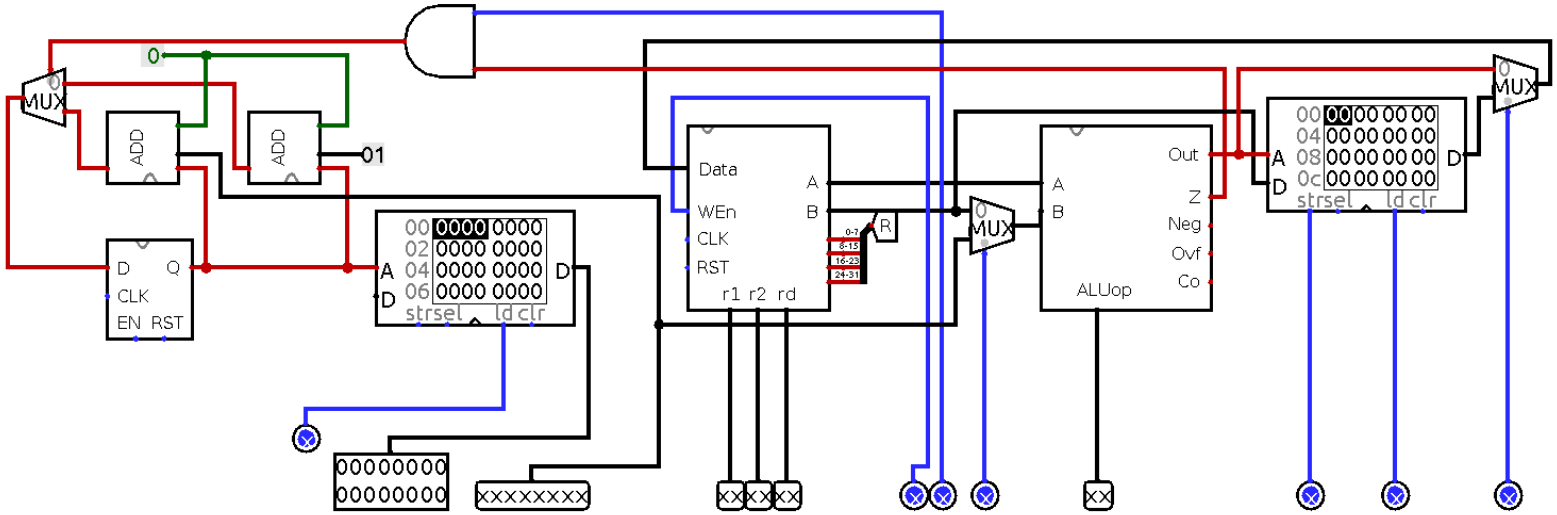
- Il filo blu e nero (scuro) sono due segnali provenienti dalla control unit (ancora da gestire).

Parte 03 - connessioni



- Nello specifico abbiamo interconnesso il **bus degli immediati** ed il **flag zero** dell'ALU al segnale di controllo dei jump.
- L'and abilita il jump solo se è impostato dalla control unit.

Parte 04 – Control Unit



- Progettiamo un'unità di controllo che in base all'istruzione da effettuare vada a settare in maniera automatica tutti i segnali di controllo di tutti i moduli della CPU (i pin di ingresso / uscita riportati in figura).

Parte 04 – Control Unit

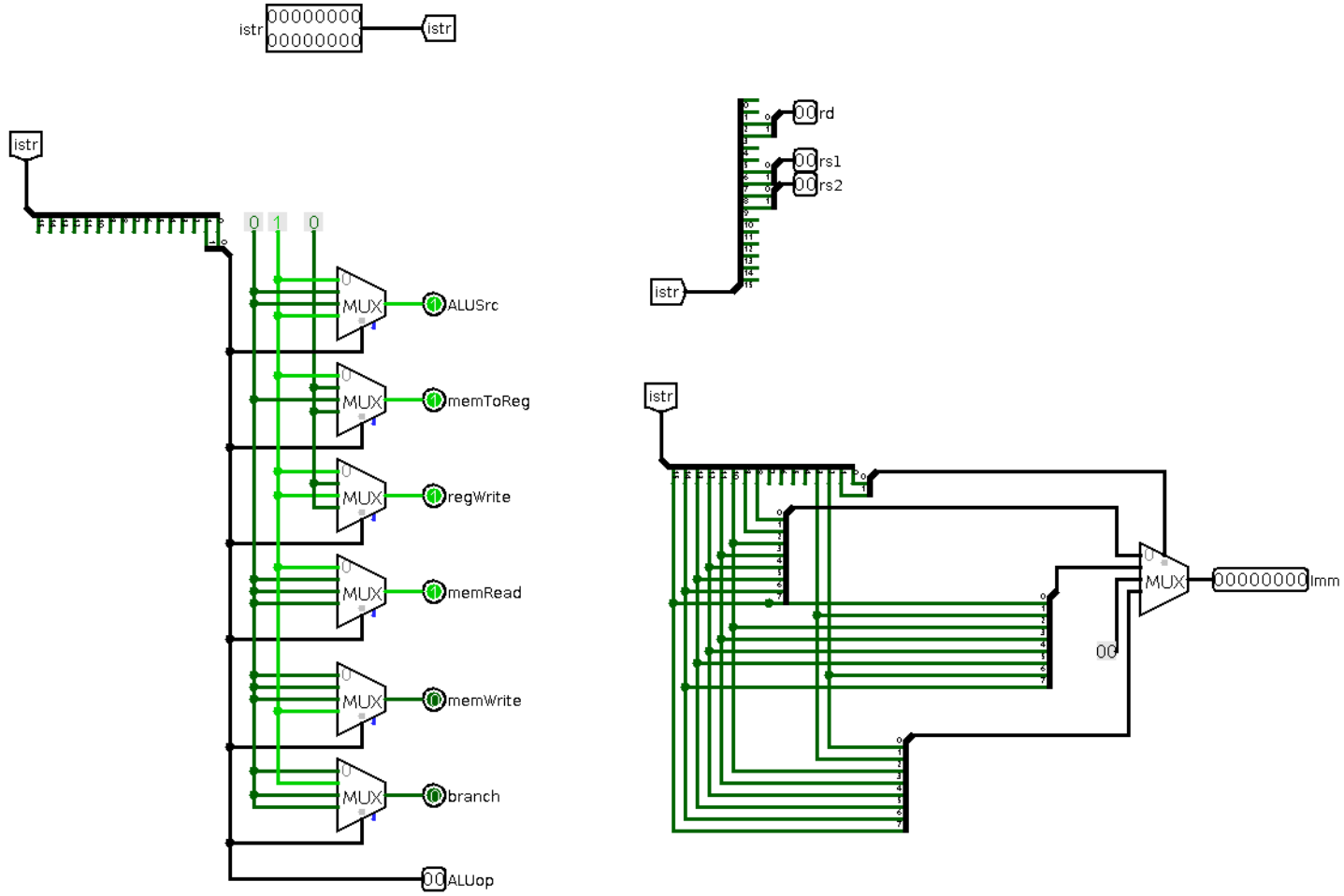
- Realizzare un circuito digitale che vada ad implementare le seguenti tabelle di verità:

Istruzione	ALUsrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUop1	ALUop0
R-Type	0	0	1	0	0	0	1	0
lb	1	1	1	1	0	0	0	0
sb	1	x	0	0	1	0	1	1
beq	0	x	0	0	0	1	0	1

- Per le istruzioni R-Type:

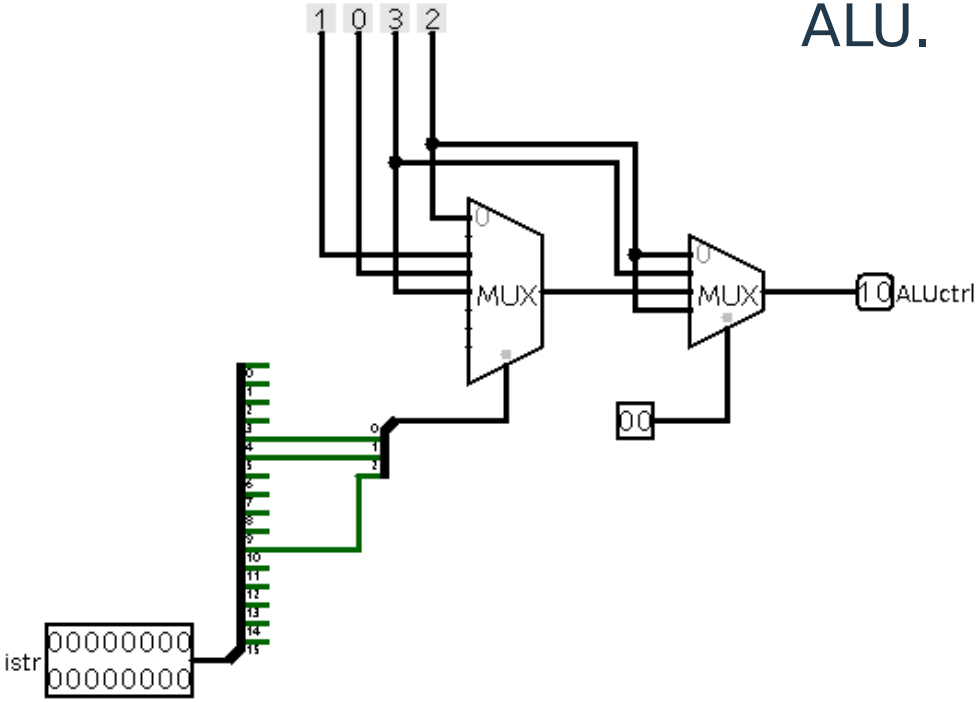
opcode	funct7	funct3	ALU Control	Istruzione
10	00	00	10	add
10	01	00	11	sub
10	00	11	00	and
10	00	10	01	or

Parte 04 – circuito (1)



Parte 04 – circuito (2)

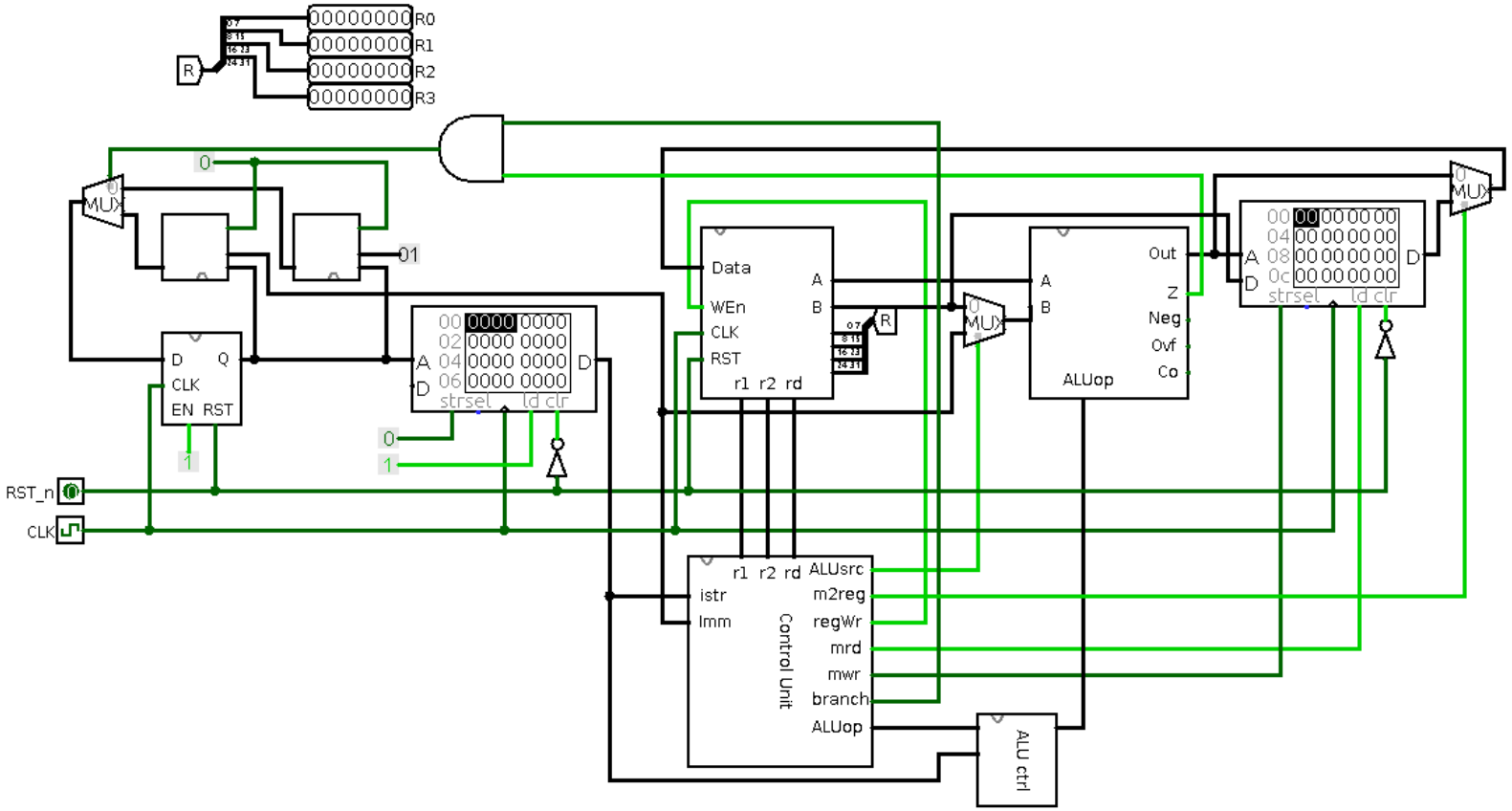
- Codifica delle istruzioni ALU.



Parte 05 – circuito finale

- Partire dal circuito realizzato nella parte 04, importare la control unit realizzata al passo precedente.
- Successivamente sostituire tutti i pin nei segnali di controllo con le uscite della control unit.
- Infine collegare i reset di tutte le memorie ed aggiungere il segnale di clock.

Parte 05 – circuito finale



Esercizio per casa

- Estendere l'architettura realizzata in questa esercitazione ed implementare la seguente istruzione:
  - *addi X1, X2, imm*
- Suggerimento: la somma tra immediato e registro è già implementata a livello di datapath, vedere le istruzioni load / store.

Esercizio per casa - Soluzione

- Codificare la nuova istruzione e aggiungerla alla tabella delle istruzioni implementate:

Formato	Istruzione	Opcode	funct3	funct7	ALU Action	ALU Control
R-Type	add	10	00	00	add	10
	sub	10	00	01	sub	11
	and	10	11	00	and	00
	or	10	10	00	or	01
SB-Type	beq	01	xx	xx	sub	11
I-Type	lb	00	00	xx	add	10
	addi	00	01	xx	add	10
S-Type	sb	11	xx	xx	add	10

Esercizio per casa - Soluzione

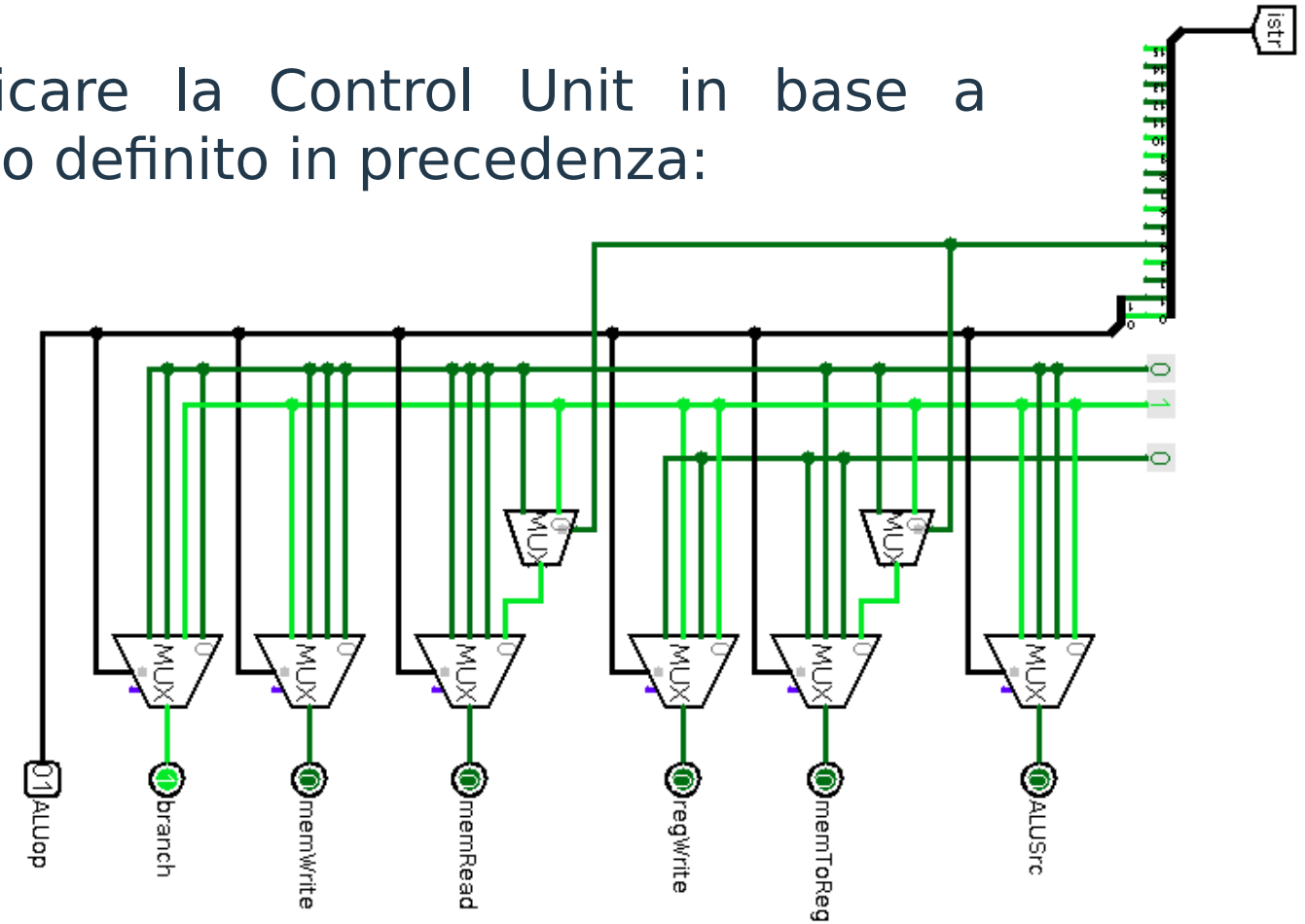
- Specificare il comportamento della Control Unit rispetto all'istruzione aggiunta:

Istruzione	ALUsrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUop1	ALUop0
R-Type	0	0	1	0	0	0	1	0
lb	1	**1**	1	**1**	0	0	0	0
sb	1	x	0	0	1	0	1	1
beq	0	x	0	0	0	1	0	1
addi	1	**0**	1	**0**	0	0	0	0

- Notare le differenze rispetto alla load byte.

Esercizio per casa - Soluzione

- Modificare la Control Unit in base a quanto definito in precedenza:



Esercizio Finale

- Esercizio di riepilogo, attraverso il quale metteremo in pratica tutte le cose viste durante le esercitazioni.
- Scrivere un programma assembly che vada ad implementare una **somma di vettori**, il programma deve essere eseguibile sul **processore progettato** in questa esercitazione.
- Il programma deve essere **assemblato a mano** e deve essere composto solo dalle istruzioni supportate (*add, sub, and, or, lb, sb, addi, beq*).

Esercizio Finale - Soluzione

```
.section .data
    a: .byte  1, 2, 3, 4
    b: .byte  5, 6, 7, 8
    c: .space 4

.section .text

        addi    R1, R0, 4

    loop:   beq     R2, R1, endl

        lb      R0, 0(R2)
        lb      R3, 4(R2)
        add     R3, R0, R3
        sb      R3, 8(R2)

        addi    R2, R2, 1
        beq     R0, R0, loop

    endl:   beq     R0, R0, endl
```

Esercizio Finale - Soluzione

- Assemblare il segmento dati:

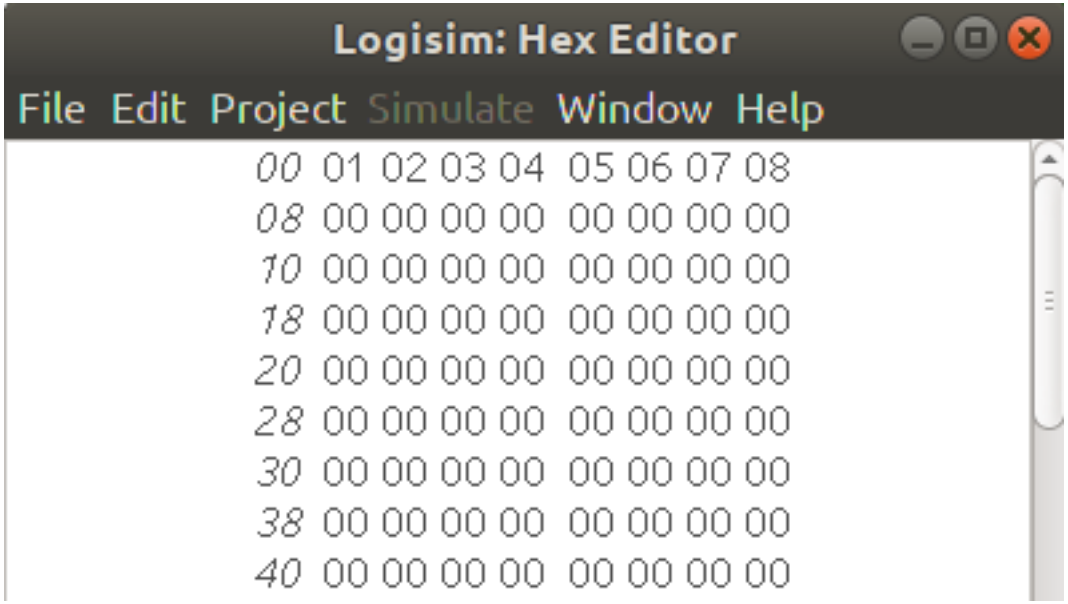
```
.section .data
    a: .byte  1, 2, 3, 4
    b: .byte  5, 6, 7, 8
    c: .space 4
```

- Semplicemente sostituire le label “a”, “b” e “c” con gli indirizzi in memoria:

```
.section .data
    0x00: 0x01 0x02 0x03 0x04
    0x04: 0x05 0x06 0x07 0x08
```

Esercizio Finale - Soluzione

- Posizionare infine il contenuto del segmento dati, all'interno della **data memory** della CPU, come in figura:



Esercizio Finale - Soluzione

- Assemblare il segmento text, sostituendo per ogni istruzione il corrispettivo codice esadecimale. Per esempio:
- *addi R1, R0, 4* → Istruzione I-Type, avente il seguente formato:

15		*…*		*0*
Imm[7:0]	**rs1**	**funct3**	**rd**	**op**
8bit	*2bit*	*2bit*	*2bit*	*2bit*

- Nel nostro caso diventa:
- 0000.0100.0001.0100 → 0x0414

Esercizio Finale - Soluzione

- Procedere in questo modo per tutte le altre istruzioni del programma.
- Tenere a mente che per le istruzioni di salto (*beq*), la label va tradotta in un **indirizzo relativo**, ad esempio:
- *beq R2, R1, endl* → in questo caso la label endl è posizionata 7 istruzioni dopo, quindi in caso di salto il Program Counter si deve incrementare di 7.
- 1000.0110.0100.1001 → 0x8649

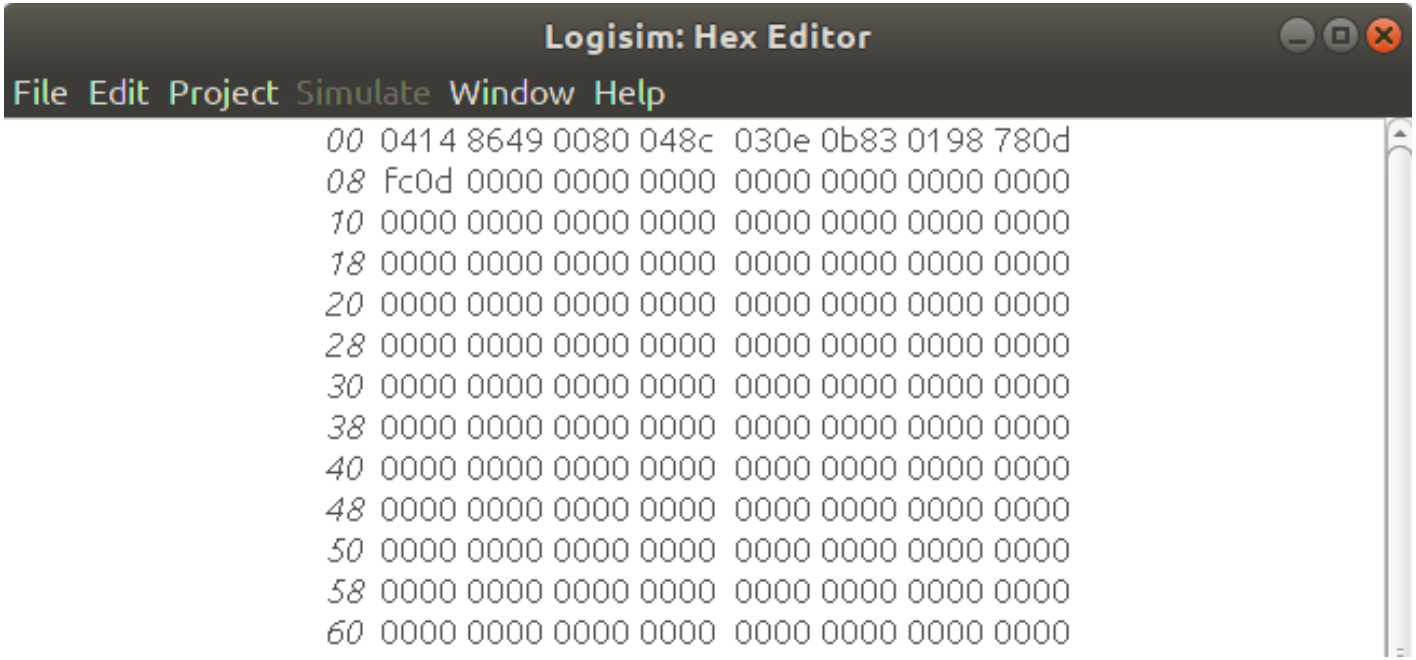
Esercizio Finale - Soluzione

```
.section .data
    0x00: 0x01 0x02 0x03 0x04
    0x04: 0x05 0x06 0x07 0x08

.section .text
    0x00:     0x0414
    0x01:     0x8649
    0x02:     0x0080
    0x03:     0x048C
    0x04:     0x030E
    0x05:     0x0B83
    0x06:     0x0198
    0x07:     0x780D
    0x08:     0xFC0D
```

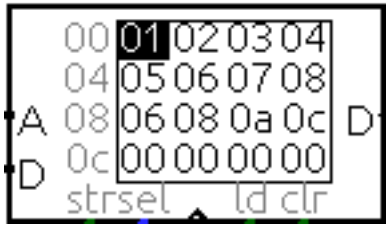
Esercizio Finale - Soluzione

- Inizializzare infine anche le istruzioni assemblate all'interno della **instruction memory**:



Esercizio Finale - Soluzione

- Memoria dati dopo l'esecuzione del programma:



  - Notiamo i due vettori che abbiamo inizializzato (*0x00, 0x04*).
  - Ed il vettore somma posizionato all'indirizzo *0x08.*