

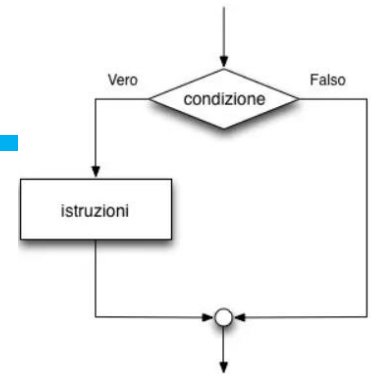
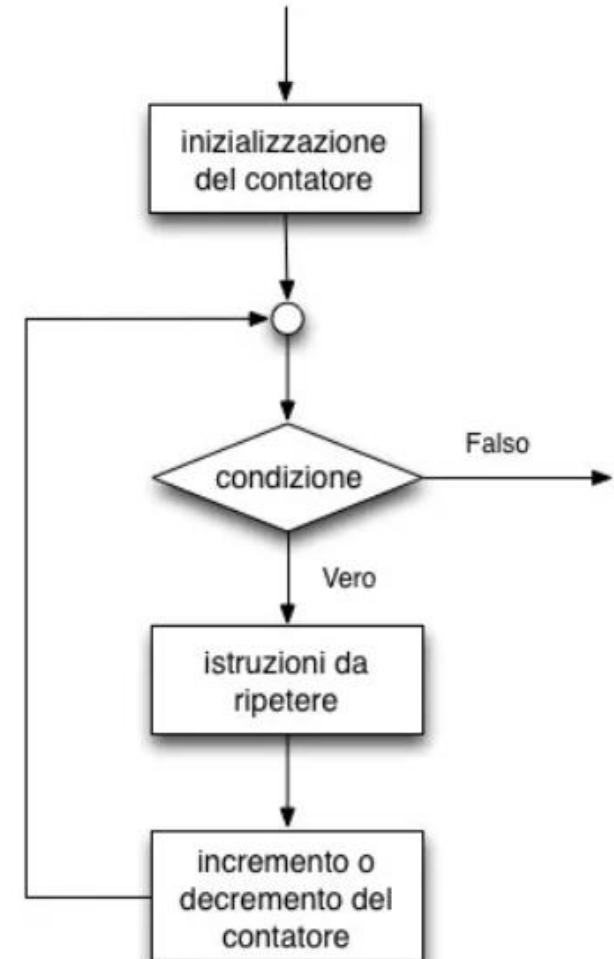
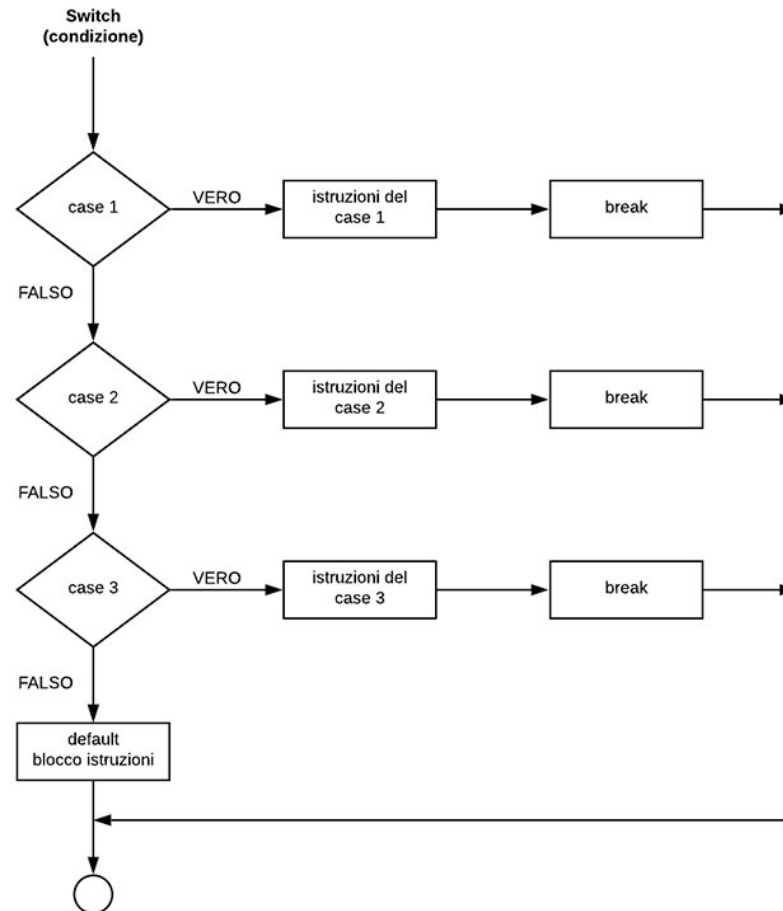
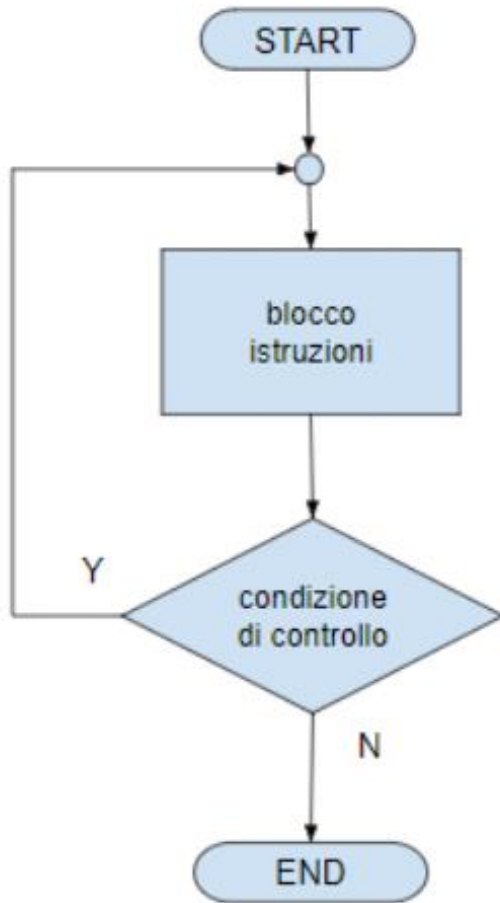
Il Set di Istruzioni RISC-V

Seconda parte

Autore Principale

Controllo di flusso

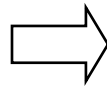
Una struttura di controllo è un costrutto sintattico di un linguaggio di programmazione che serve a specificare se, quando, in quale ordine e quante volte devono essere eseguite le istruzioni del codice sorgente.



Controllo del flusso del programma: salti incondizionati

- Se devo implementare del codice come:

```
if (b!=c)
    a=b+c;
else
    a=b-c;
```



$s3 \equiv a$, $s4 \equiv b$, $s5 \equiv c$

```
Lab1:  beq s4, s5, Lab1
        add s3, s4, s5
        beq x0, x0, Lab2
Lab2:  sub s3, s4, s5
        ...
```

- Normalmente si usa la condizione opposta
 - Di solito ci si aspetta che il test sia vero, anche per motivi di prestazioni
- L'istruzione `beq x0, x0, Lab2` realizza un salto **incondizionato**, dato che la condizione `x0==x0` e' ovviamente sempre vera
- I compilatori contengono molte più etichette rispetto a quelle strettamente necessarie dato il programma sorgente

Etichette

```
_Z14sommaduenumeriii:  
.LFB1522:  
.cfi_startproc  
endbr64  
pushq    %rbp    #  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq     %rsp, %rbp    #,  
.cfi_def_cfa_register 6  
movl     %edi, -4(%rbp) # a, a  
movl     %esi, -8(%rbp) # b, b  
# hellofun.cpp:4:      return a+b;  
movl     -4(%rbp), %edx # a, tmp84  
movl     -8(%rbp), %eax # b, tmp85  
addl     %edx, %eax    # tmp84, _3  
# hellofun.cpp:5: }  
popq     %rbp    #  
.cfi_def_cfa 7, 8  
ret  
.cfi_endproc
```

```
#include <iostream>  
  
int sommaduenumeri(int a, int b){  
    return a+b;  
}  
  
int main(int argc, char *argv[])  
{  
    int a = 4, b, c;  
    b = 5;  
    //c = a+b;  
    c = sommaduenumeri(a,b);  
    std::cout << "Hello World! " << c << "\n";  
    exit(0);  
}
```

```
dago@workstation:~$ nm hellofun | grep sommadue  
0000000000000129d t _GLOBAL__sub_I__Z14sommaduenumeriii  
000000000000011c9 T _Z14sommaduenumeriii
```

Controllo del flusso di programma

In generale abbiamo sei istruzioni di salto

- beq e bne già viste
- blt (branch if less than) $x1, x2, \text{label}$, salto se $x1 < x2$
- bge (branch if greater than or equal to), salto se $x1 \geq x2$ (notare !=)
- bltu and bgeu trattano $x1$ e $x2$ come numeri unsigned

Pseudoistruzioni utili

- slt, sltu rd r1 r2 $rd = 1$ se $r1 < r2$, oppure $rd = 0$ (unsigned)
- slti, sltiu rd r1 cost $rd = 1$ se $r1 < \text{cost}$, oppure $rd = 0$

Esempio

- Tradurre in assembler l'istruzione
if (i == j) f=g+h; else f = g-h; con f-i in x19-x23

Esempio

- Tradurre in assembler l'istruzione
if (i == j) f=g+h; else f = g-h; con f-j in x19-x23

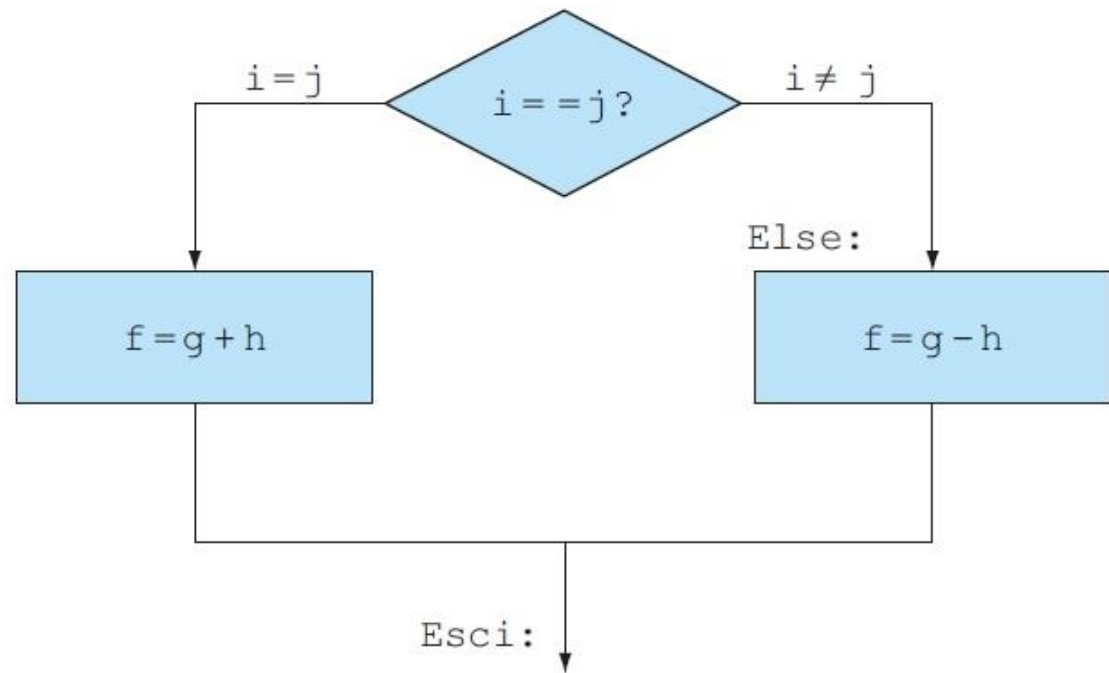
bne x22, x23 Else

add x19, x20, x21

beq x0, x0, End

Else: sub x19, x20, x21

Esci:



Ciclo While

while (salva[i] == k) i++;

Con i in x22, k in x24 e salva in x25, double salva[]

```
addi x22 zero zero
add t0, x25, x22
ld t1, 0(t0)
while: BNE t1, x24, end
add t0, t0, 8
addi x22 x22 1
beq z, z, while
```


Ciclo While

while (salva[i] == k) i++;

Con i in x22, k in x24 e salva in x25, double salva[]

Ciclo: slli, x10, x22, 3 //i*8
 add x10, x10, x25 //salva+i
 ld x9, 0(x10)
 bne x9, x24, Esci
 addi x22, x22, 1
 beq x0, x0, Ciclo

Esci:

Come avevamo fatto in precedenza per A[k] ?

Cicli FOR, WHILE, DO..WHILE

for (k=0; k<100; ++k) {...} ...

ST1 ST2 ST3 ST4 ST5

```

add s0,x0,x0      ST1

ini_for:          ST2
    slt t0,s0,s1
    beq t0,x0,fin_for

...              ST4

add s0,s0,s2      ST3
    beq x0,x0,ini_for

fin_for:          ST5
...
    
```

while (k<100) {...} ...

ST1 ST2 ST3

```

ini_wh:          ST1
    slt t0,s0,s1
    beq t0,x0,fin_wh

...              ST2

    beq x0,x0,ini_wh

fin_wh:          ST3
    ...
    
```

do {...} while (k<100); ...

ST1 ST2 ST3

```

ini_do:          ST1
    ...

    slt t0,s0,s1  ST2
    bne t0,x0,ini_do

...              ST3
    
```

Case/Switch

High-Level Code

```
switch (button) {  
    case 1:  amt = 20; break;
```

```
  
    case 2:  amt = 50; break;
```

```
  
    case 3:  amt = 100; break;
```

```
  
    default: amt = 0;  
}
```

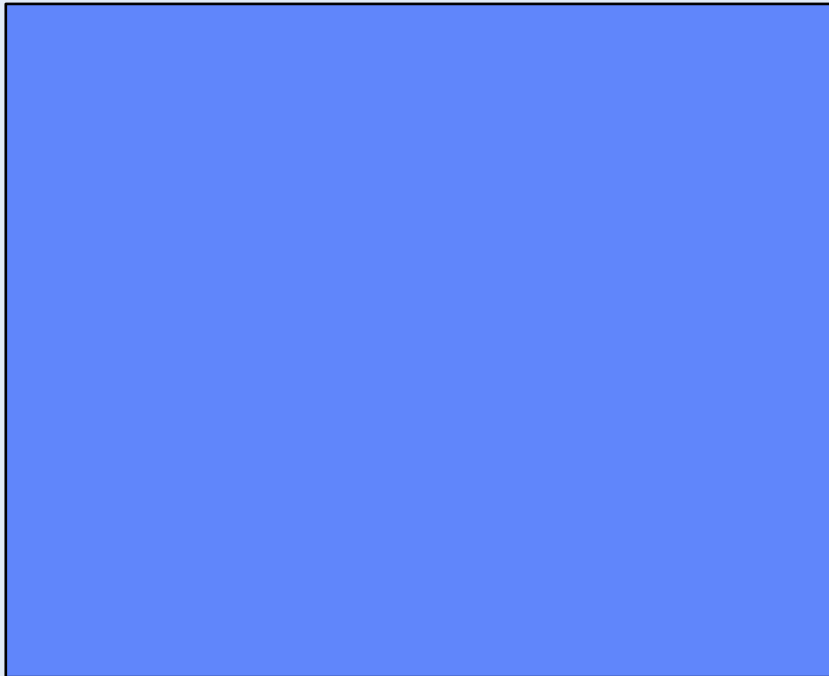
```
// equivalent function using  
// if/else statements
```

```
if      (button == 1)  amt = 20;  
else if (button == 2)  amt = 50;  
else if (button == 3)  amt = 100;  
else                    amt = 0;
```

RISC-V Assembly Code

```
# s0 = button, s1 = amt
```

```
case1:
```



```
done:
```

Case/Switch

High-Level Code

```
switch (button) {  
    case 1:  amt = 20; break;  
  
  
  
  
  
  
  
  
  
    case 2:  amt = 50; break;  
  
  
  
  
  
  
  
  
  
    case 3:  amt = 100; break;  
  
  
  
  
  
  
  
  
  
    default: amt = 0;  
}
```

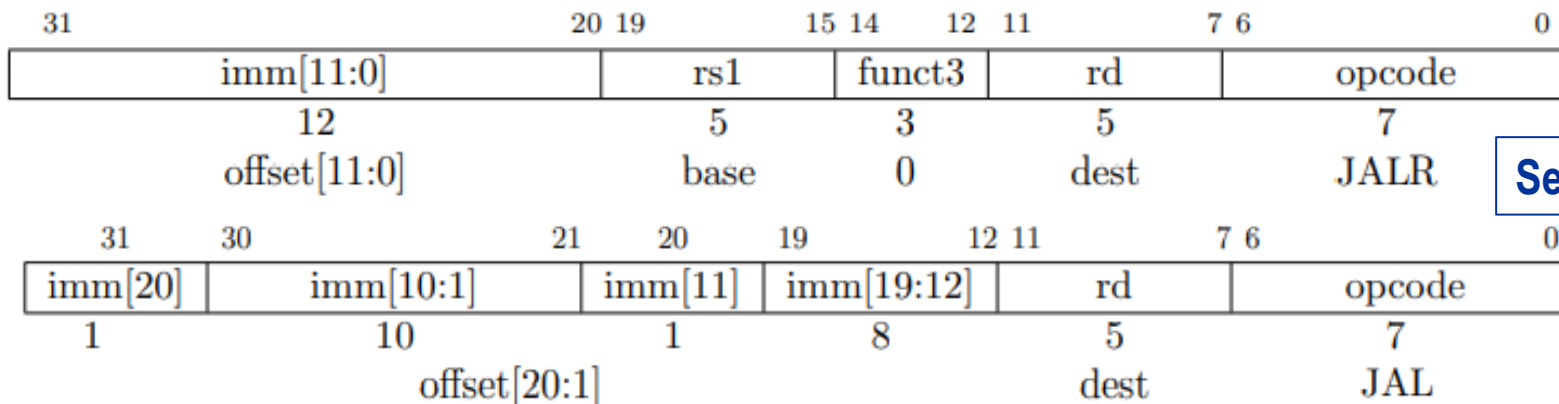
```
// equivalent function using  
// if/else statements  
if      (button == 1)  amt = 20;  
else if (button == 2)  amt = 50;  
else if (button == 3)  amt = 100;  
else                   amt = 0;
```

RISC-V Assembly Code

```
# s0 = button, s1 = amt  
  
case1:  
    addi  t0, zero, 1      # t0 = 1  
    bne   s0, t0, case2    # button == 1?  
    addi  s1, zero, 20     # if yes, amt = 20  
    j     done             # break out of case  
case2:  
    addi  t0, zero, 2      # t0 = 2  
    bne   s0, t0, case3    # button == 2?  
    addi  s1, zero, 50     # if yes, amt = 50  
    j     done             # break out of case  
case3:  
    addi  t0, zero, 3      # t0 = 3  
    bne   s0, t0, default  # button == 3?  
    addi  s1, zero, 100    # if yes, amt = 100  
    j     done             # break out of case  
default:  
    add   s1, zero, zero   # amt=0  
done:
```

Case/Switch

- Potremmo memorizzare in una tabella gli indirizzi dell'inizio dei vari frammenti di codice
 - La tabella si chiama branch address table (tabella di salto), etichette-indirizzo
- Metto l'indirizzo in un registro e lo uso per il salto
- Dal momento che è un'opzione usata per la gestione delle procedure, è stata introdotta un'istruzione di salto indiretto
 - JALR jump and link register
 - Abbiamo anche la JAL, che «salta» senza usare un registro
- (UJ) JAL rd, const $rd=PC+4$, $PC=const(\text{bit } 0 \text{ aggiunto})$
- (I) JALR rd, rs1, offset $rd=PC+4$, $PC=offset(rs1)$
- Saranno più chiare in seguito



Se $rd == x0$ non ritorno...

Branch address / Jump Tables

```
#include <stdio.h>
#include <stdlib.h>

typedef void (*Handler)(void);    /* A pointer to a handler f

/* The functions */
void func3 (void) { printf( "3\n" ); }
void func2 (void) { printf( "2\n" ); }
void func1 (void) { printf( "1\n" ); }
void func0 (void) { printf( "0\n" ); }

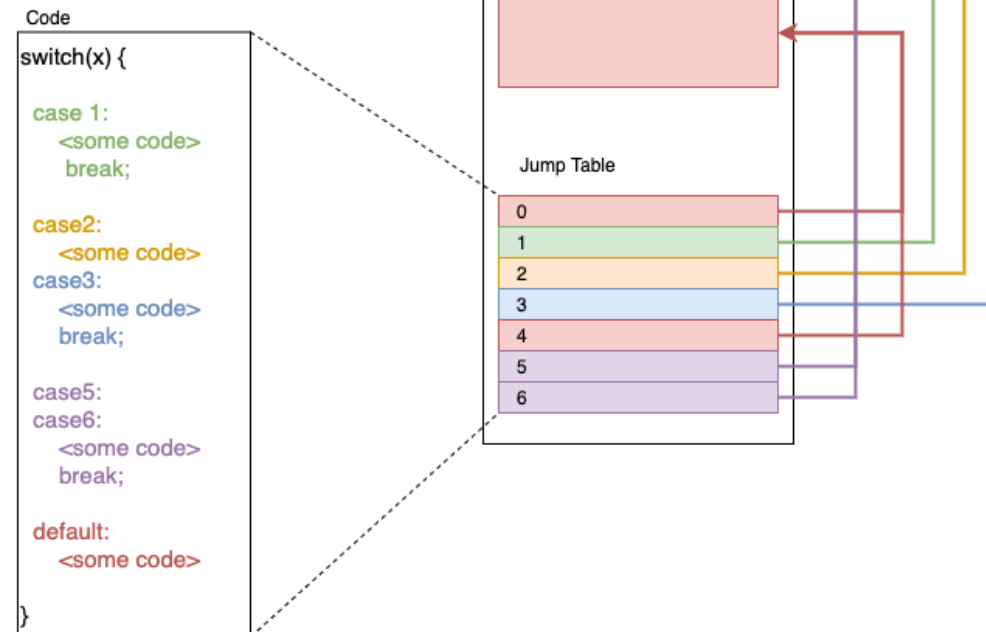
Handler jump_table[4] = {func0, func1, func2, func3};

int main (int argc, char **argv) {
    int value;

    /* Convert first argument to 0-3 integer (modulus) */
    value = atoi(argv[1]) % 4;

    /* Call appropriate function (func0 thru func3) */
    jump_table[value]();

    return 0;
}
```



Costanti piccole e grandi

- Con i formati I ed S posso inserire costanti **PICCOLE**
 - La costante piccola è un numero da -2048 a 2047 (intero con segno su 12 bit)
- Con i formati U e UJ ho costanti da 20 bit
 - Lui + ori = la, carico costanti da 32 bit, [-2 GiB, 2 GiB)
- Le istruzioni di salto condizionato utilizzano il formato SB con 13 bit
 - Ma il LSB è sempre 0 e notate la posizione di 11 e 12



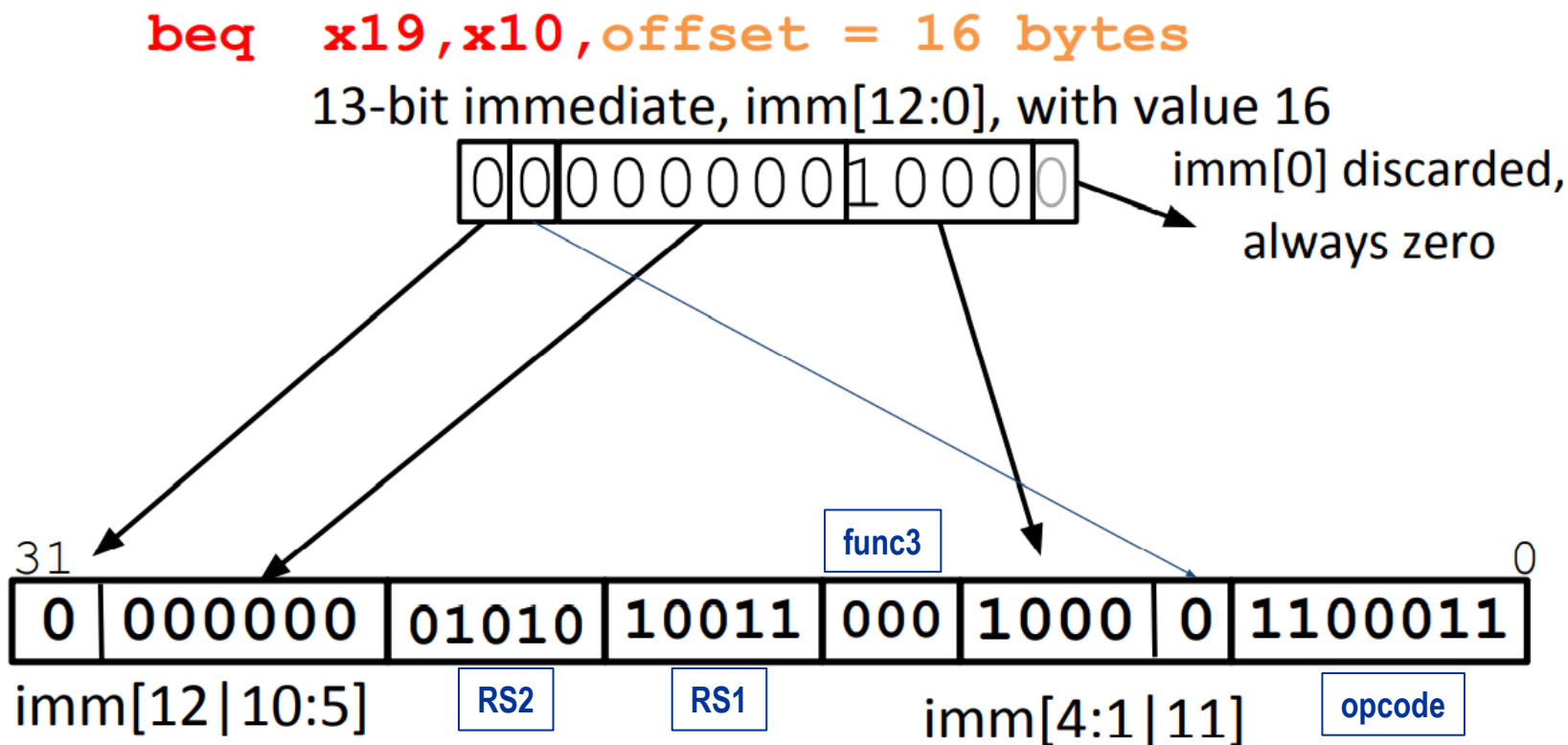
Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12, 10:5]	rs2	rs1	funz3	immed[4:1, 11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

Figura 2.19 Formati delle istruzioni RISC-V.

- How to encode label, i.e., where to branch to?
- Use the immediate field as a two's complement offset to PC
 - Can specify $\pm 2^{11}$ addresses from the PC
- Instructions are on 32 bits and "word-aligned": Address is always a multiple of 4 (in bytes)
 - Let immediate specify #words instead of #bytes
 - we will now specify $\pm 2^{11}$ words = $\pm 2^{13}$ byte addresses around PC
- If we don't take the branch: $PC = PC + 4 = \text{next instruction}$
- If we do take the branch: $PC = PC + (\text{immediate} * 4)$

BUT

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length
- So immediate values address HALF WORDS in terms of bytes
 - RISC-V conditional branches can only reach $\pm 2^{10} \times 32$ -bit instructions either side of PC
 - immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments
 - LSB is always 0



Dalle specifiche ufficiali

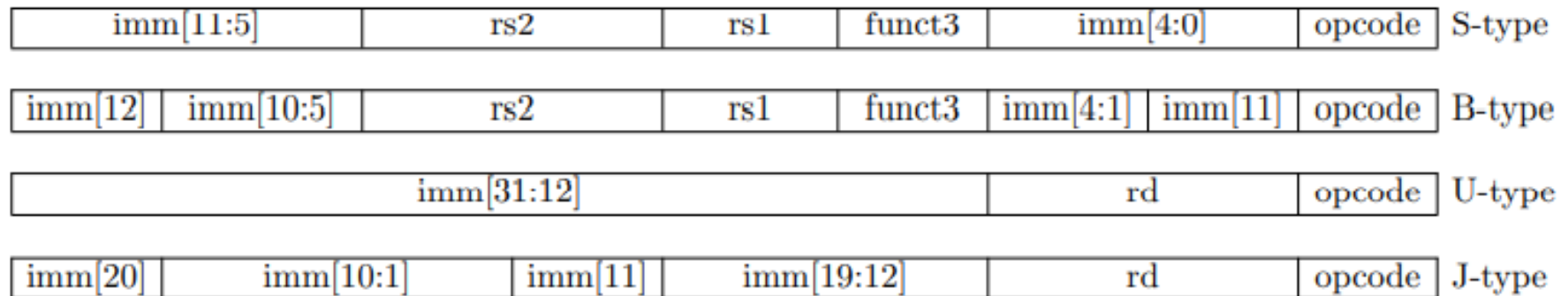


Figure 2.3: RISC-V base instruction formats showing immediate variants.

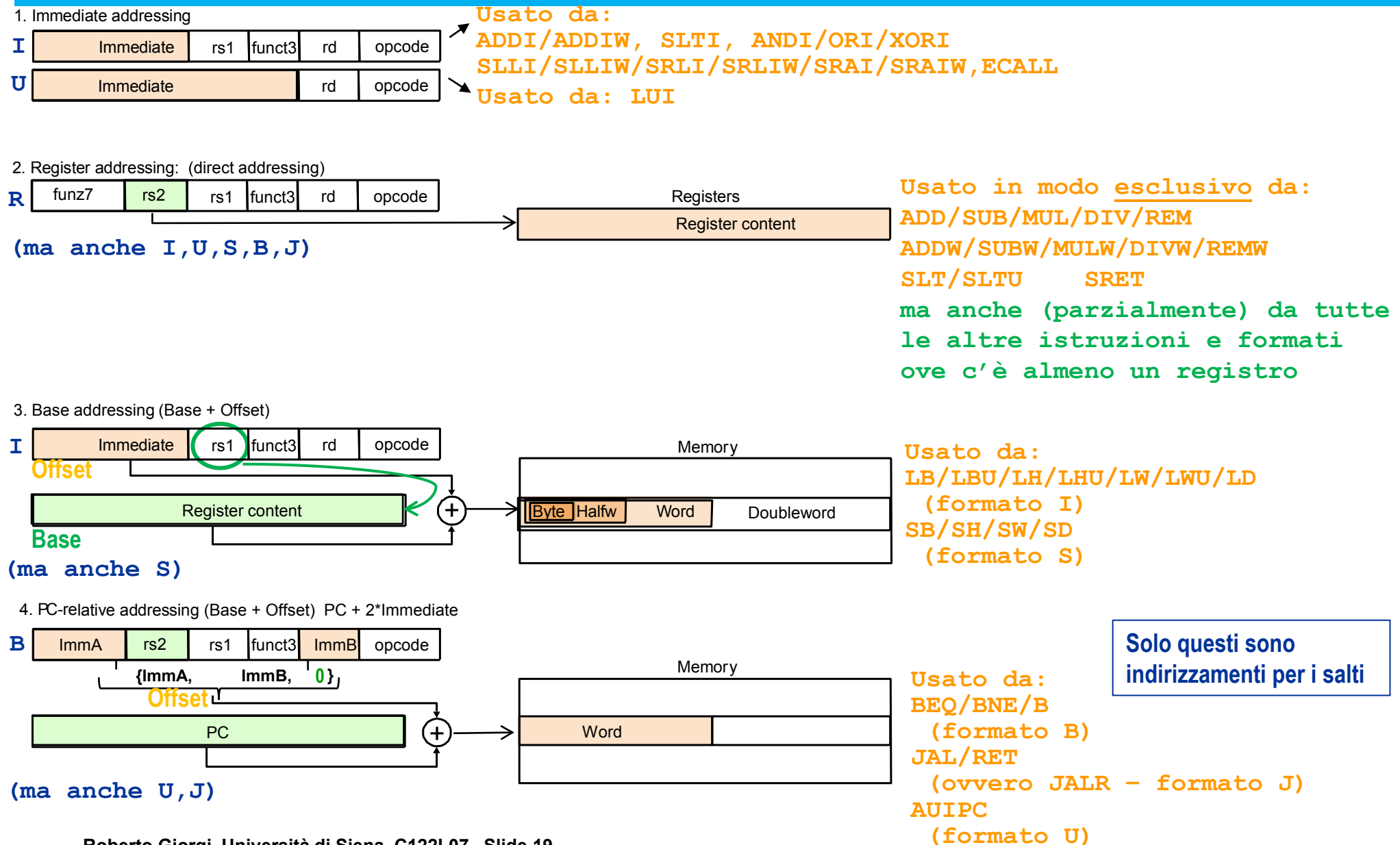
The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Quindi i bit da 1 a 10 sono nella stessa identica posizione sia per le store che per i branch. Anche il bit di segno è nella stessa posizione, solo che è il bit 11 per S ed il 12 per B.

A questo punto dove metto il bit 11 di B? Nell'unica posizione rimanente, la 0 di S, perché tanto lo 0 di B è sempre 0.



Tabella riassuntiva modi indirizzamento del RISC-V:



Esercizio 1


High-Level Code

```
// determines the power
// of x such that  $2^x = 128$ 
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

RISC-V Assembly Code

```
# s0 = pow, s1 = x
```



```
done:
```

Soluzione

High-Level Code

```
// determines the power
// of x such that  $2^x = 128$ 
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

RISC-V Assembly Code

```
# s0 = pow, s1 = x
    addi s0, zero, 1      # pow = 1
    add  s1, zero, zero   # x = 0

    addi t0, zero, 128    # t0 = 128
while: beq  s0, t0, done   # pow = 128?
        slli s0, s0, 1     # pow = pow * 2
        addi s1, s1, 1     # x = x + 1
        j    while        # repeat loop
done:
```

Esercizio 2

High-Level Code

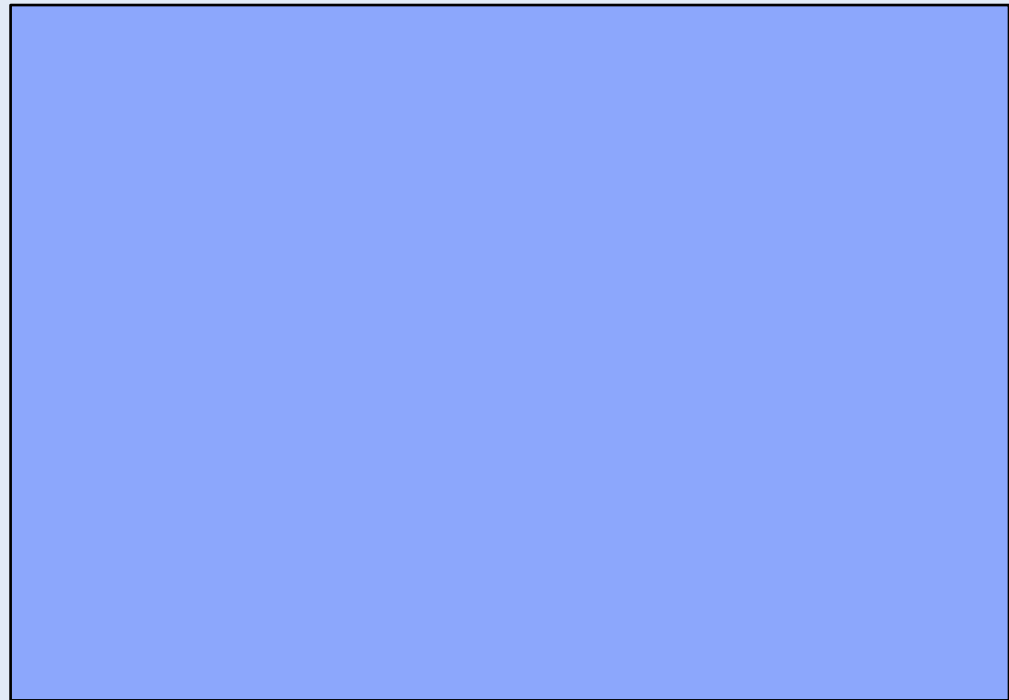
```
int i;  
int scores[200];
```

```
for (i = 0; i < 200; i = i + 1)
```

```
    scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i
```



```
done:
```

Esercizio 2

High-Level Code

```
int i;  
int scores[200];  
  
for (i = 0; i < 200; i = i + 1)  
  
    scores[i] = scores[i] + 10;
```

RISC-V Assembly Code

```
# s0 = scores base address, s1 = i  
  
addi s1, zero, 0    # i = 0  
addi t2, zero, 200  # t2 = 200  
  
for:  
    bge s1, t2, done # if i >= 200 then done  
    slli t0, s1, 2    # t0 = i * 4  
    add t0, t0, s0     # address of scores[i]  
    lw t1, 0(t0)      # t1 = scores[i]  
    addi t1, t1, 10    # t1 = scores[i] + 10  
    sw t1, 0(t0)      # scores[i] = t1  
    addi s1, s1, 1     # i = i + 1  
    j for              # repeat  
done:
```

- ```
// high-level code
// chararray[10] was declared and initialized earlier
int i;

for (i = 0; i < 10; i = i + 1)
 chararray[i] = chararray[i] - 32;
```

```
RISC-V assembly code
s0 = base address of chararray (initialized earlier), s1 = i
 addi s1, zero, 0 # i = 0
 addi t3, zero, 10 # t3 = 10
for: bge s1, t3, done # i >= 10 ?
 add t4, s0, s1 # t4 = address of chararray[i]
 lb t5, 0(t4) # t5 = chararray[i]
 addi t5, t5, -32 # t5 = chararray[i] - 32
 sb t5, 0(t4) # chararray[i] = t5
 addi s1, s1, 1 # i = i + 1
 j for # repeat loop
done:
```

