UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

2. Rappresentazione Intermedia

Linguaggi e Compilatori [I215-011]

Corso di Laurea in INFORMATICA
(D.M.270/04) [16-262]
Anno accademico 2023/2024

Prof. Andrea Marongiu
andrea.marongiu@unimore.it

Copyright note

Credits

- Cooper, Torczon, “Engineering a Compiler”, Elsevier
- Aho, Lam, Sethi, Ullman, “Compilatori: principi, tecniche e strumenti – seconda edizione”, Pearson
- Gibbons, Carnegie Mellon University, “Optimizing Compilers”
- Pekhimenko, University of Toronto, “Compiler Optimization”

Rappresentazione Intermedia

- Abbiamo visto che il middle-end è organizzato come una sequenza di *passi*
  - *Passi* di analisi
    - Consumano la IR
  - *Passi* di trasformazione
    - Producono nuova IR
- Per analizzare il codice e trasformarlo occorre una IR espressiva che mantenga tutte le informazioni importanti da trasmettere da un *passo* all'altro

Proprietà di una IR

- Facilità di generazione
- Facilità di manipolazione
- Costo di manipolazione
- Livello di astrazione
- Livello di dettaglio esposto

- Sottili decisioni di progetto di una IR possono avere effetti molto intricati sulla velocità ed efficienza di un compilatore

Tipi di IR

- In un compilatore possono esserci tanti tipi diversi di IR:
  - Abstract syntax trees (AST)
  - Directed acyclic graphs (DAG)
  - 3-address code (3AC)
  - Static single assignment (SSA)
  - Control flow graphs (CFG)
  - Call graph (CG)
  - Program dependence graphs (PDG)

Categorie di IR

- **Grafiche** (o strutturali)
  - Orientate ai grafi
  - Molto usate nella *source-to-source translation*
  - Tendono a essere voluminose

- **Esempi:**
  - AST, DAG

Ex: da C to C

- **Lineari** → N. Registri infinito
  - Pseudocodice per macchine astratte
  - Il livello di astrazione varia
  - Strutture dati semplici e compatte
  - Facile da riarrangiare

- **Esempi:**
  - 3-address code

Tra cui SSA

- **Ibride**
  - Sfruttano una combinazione di forme grafiche e lineari

- **Esempi:**
  - Control flow graph

Sintassi Concreta (testo)

- La maniera più semplice di rappresentare un programma è il testo, ovvero la sintassi concreta

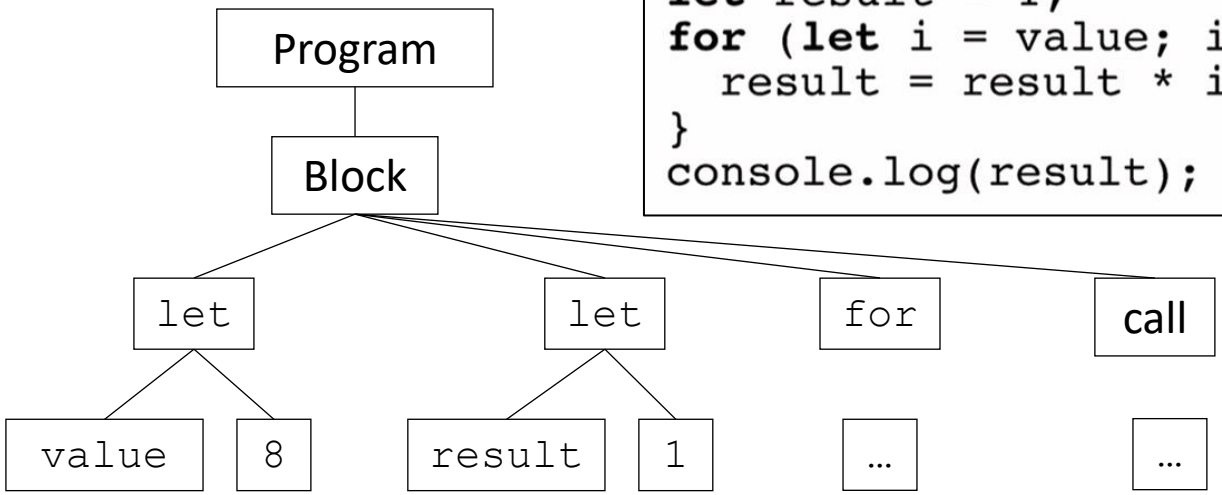
```
let value = 8;
let result = 1;
for (let i = value; i > 0; i = i - 1) {
  result = result * i;
}
console.log(result);
```

- **PRO**
  - Molto semplice
  - Vicina al livello di astrazione con cui un umano ragiona sul programma
- **CONTRO**
  - Pessima per analizzare e trasformare il codice
  - Non è il livello di astrazione corretto per comprendere la semantica del programma

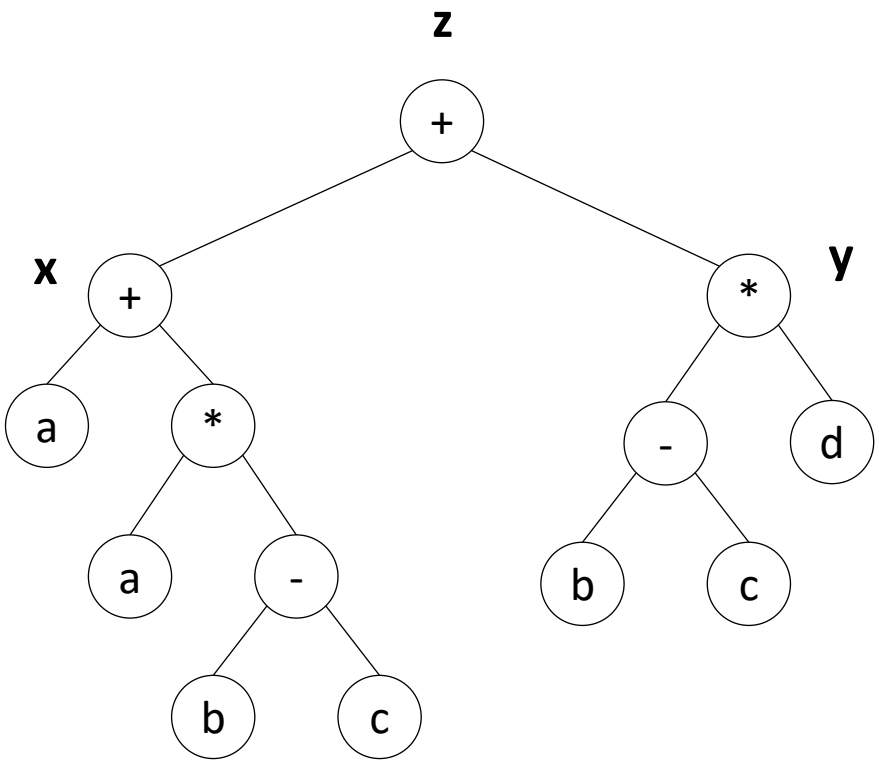
Abstract Syntax Trees (AST)

- Un albero i cui nodi rappresentano diverse parti del programma
- Il nodo radice è la nozione del programma che contiene un blocco di istruzioni, da cui discendono tanti figli quante sono le istruzioni

```
let value = 8;
let result = 1;
for (let i = value; i > 0; i = i - 1) {
  result = result * i;
}
console.log(result);
```



Abstract Syntax Trees (AST)




```
x = a + a * (b – c)
y = (b – c ) * d
z = x + y
```

- AST per espressioni

Abstract Syntax Trees

PRO

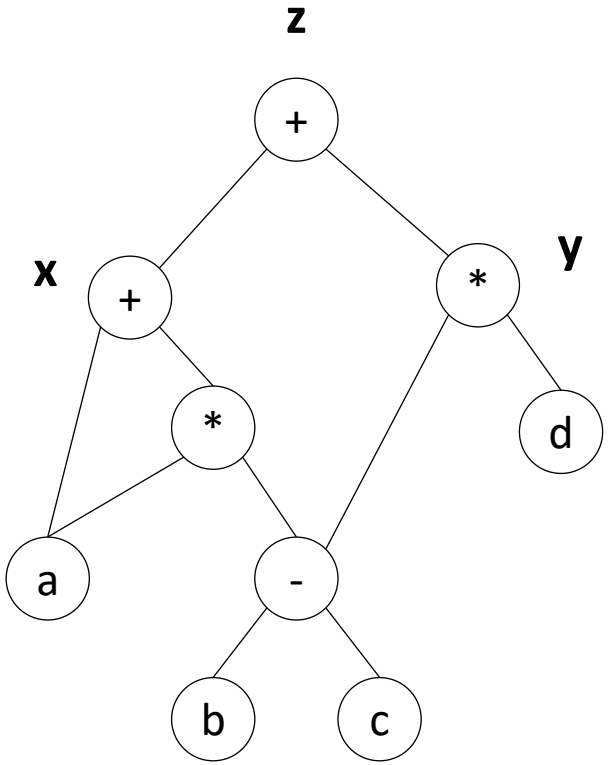
- Molto comodo per scrivere un interprete
- Basta usare una funzione ricorsiva per processare l'albero

CONTRO

- Tutti i diversi tipi di nodo nell'albero hanno un comportamento diverso.
- Scrivere un passo di analisi che sfrutti questa IR richiede di ragionare costantemente sulla differenza nella semantica dei vari tipi di nodo

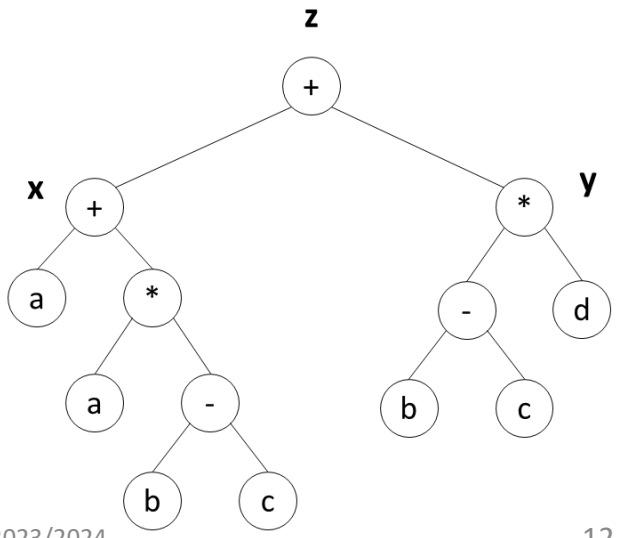
Directed Acyclic Graph (DAG)

- Un DAG è una contrazione di un AST che evita la duplicazione delle espressioni




```
x = a + a * (b – c)
y = (b – c ) * d
z = x + y
```

Stesso esempio dell'AST



Directed Acyclic Graph (DAG)

PRO

- Rappresentazione più compatta dell'AST
- Per espressioni prive di assegnamento si può generare codice che evita duplicazioni (*Common Subexpression Elimination*)

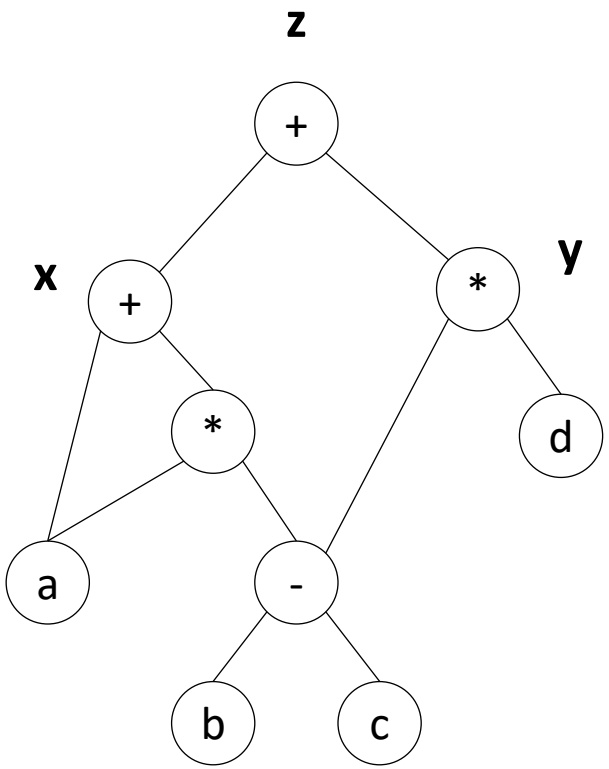
CONTRO

- Il riuso di un'espressione è possibile solo se si dimostra che il suo valore non è cambiato
- Assegnamenti e chiamate sono frequentissimi e possono alterare il valore delle espressioni
- Il DAG non ha una nozione di come le espressioni cambiano valore nel tempo !
  - Ne osserva solo la rappresentazione testuale

No runtime

Directed Acyclic Graph (DAG)

- Il codice generato da questo DAG è corretto?




```
x = a + a * (b – c)
y = (b – c ) * d
z = x + y
```

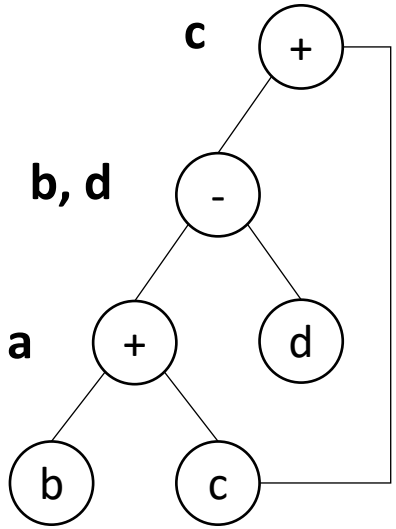
```
t1 = b – c
t2 = a * t1
x = a + t2
y = t1 * d
z = x + y
```

- Sì, perché **t1** (ovvero l'espressione **b-c**) non è cambiata prima del suo riuso

Directed Acyclic Graph (DAG)

- **Vediamo un altro esempio:**

```
a = b+c;
b = a-d;
c = b+c;
d = a-d;
```



- Il codice generato da questo DAG è corretto?

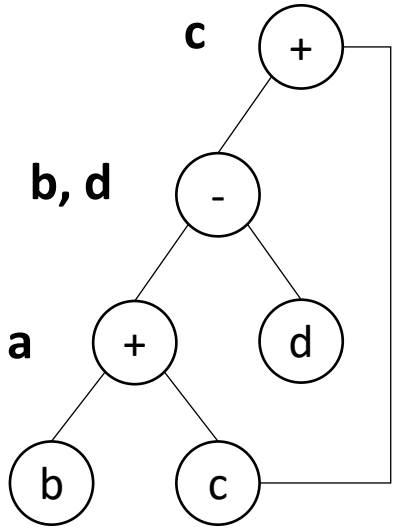
```
a = b + c
d = a - d
c = d + c
```

Directed Acyclic Graph (DAG)

- **Vediamo un altro esempio:**

```
a = b+c;
b = a-d;
c = b+c;
d = a-d;
```

No, perché il valore di **b+c** è cambiato prima del suo riuso



- Il codice generato da questo DAG è corretto?

```
a = b + c
d = a - d
c = d + c
```

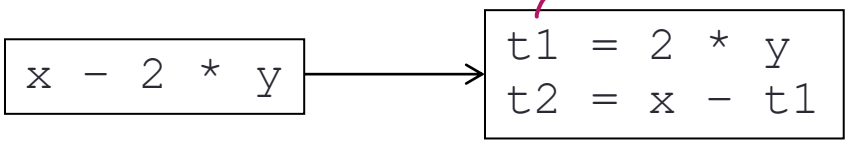
Serve una IR che renda l'analisi (e la trasformazione) del codice più regolare e predicibile

3-address code

Ci avviciniamo ad Assembly. Poichè esso è un linguaggio che ha solo massimo 2 operandi. (Ex : ADD S0 , S0 , S1)

L'unica differenza che avrò registri illimitati. Mentre in Assembly no.

- Le istruzioni hanno la forma: `x = y op z`
  - Un unico operatore e massimo tre operandi



PRO

- Espressioni complesse vengono spezzate
- Forma compatta e molto simile all'assembly
- Vengono introdotti dei temporanei per i risultati intermedi
- Registri virtuali (illimitati)

3-address code (Istruzioni principali)

assignments	`x = y op z`
	`x = op y`
	`x = y[i]`
	`x = y`
branches	`goto L`
conditional branches	`if x relop y goto L`

procedure calls	`param x`
	`param y`
	`call p`

address and pointer assignments	`x = &y` `*y = z`

3-address code – due varianti

Quadruple

`x - 2 * y`

(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

Triple

`x - 2 * y`

(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

- Semplice struttura record
- Facile da riordinare
- Nomi espliciti

- Solo 3 campi + SALVO MEMORIA
- Difficile da riordinare
- L'indice è il nome (implicito)

Ricorda:
Constant Propagation (CP)

- Per le variabili con valore costante (es., *b = 3*)
  - Sostituisce gli usi futuri di *b* con la costante
    - Se *b* non è cambiato nel frattempo

```
b = 3
c = 1 + b
d = b + c
```

```
b = 3
c = 1 + 3
d = 3 + c
```

Ricorda:
Constant Propagation (CP)

- Per le variabili con valore costante (es., *b = 3*)
  - Sostituisce gli usi futuri di *b* con la costante
    - Se *b* non è cambiato nel frattempo

```
b = 3
c = 1 + b
d = b + c
```

```
b = 3
c = 1 + 3
d = 3 + c
```

Ricorda:
Constant Propagation (CP)

- Per le variabili con valore costante (es., *b = 3*)
  - Sostituisce gli usi futuri di *b* con la costante
    - Se *b* non è cambiato nel frattempo

```
b = 3
c = 1 + b
d = b + c
```

```
b = 3
c = 1 + 3
d = 3 + c
```

- Il codice è in forma 3AC, eppure non ho certezze di poter propagare le costanti senza un'analisi dell'evoluzione temporale dei valori

Torniamo alla **Constant Propagation (CP)**

- Per le variabili con valore costante (es., *b = 3*)
  - Sostituisce gli usi futuri di *b* con la costante
    - Se *b* non è cambiato nel frattempo

```
B = 3
c = 1 + b
d = b + c
```

```
b = 3
c = 1 + 3
d = 3 + c
```

- Osserviamo questo esempio:

Qui i due **usi** di *b* si riferiscono a due **definizioni** diverse

```
b = 3
c = 1 + b
b = 4
d = b + c
```

```
b = 3
c = 1 + 3
b = 4
d = ? + c
```

3-address code

- Con la IR 3AC dunque non è immediatamente possibile applicare le ottimizzazioni viste senza aver prima analizzato l'evoluzione temporale delle espressioni
  - Vedremo in seguito come farlo → Algoritmo di value ...

- Esiste un'evoluzione della 3AC che semplifica questo tipo di ottimizzazioni
  - Static Single Assignment (SSA) form

Static Single Assignment (SSA)

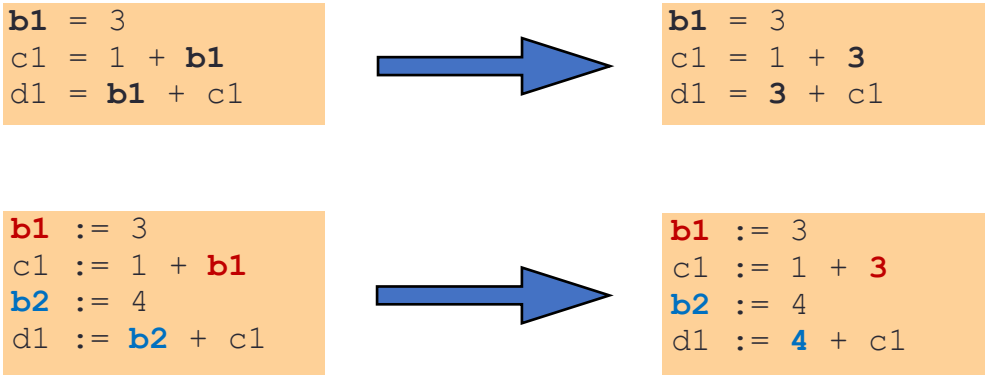
Semplifica costant propagation.

- Ogni variabile è definita (assegnata) solo una volta
- Definizioni multiple della stessa variabile originale sono tradotti in multiple versioni della variabile

- PRO
  - Ogni definizione ha associata una lista di tutti i suoi usi
  - Ogni variabile operando di un'istruzione (espressione) è un uso di una qualche definizione, ed è ad essa direttamente collegata
  - Semplifica analisi e trasformazione del codice (quasi sempre)

Forma SSA e **Constant Propagation (CP)**

- Con la forma SSA diventa immediato propagare il valore costante di una definizione ai suoi usi
  - (es., *b = 3*)



- Gli stessi vantaggi si hanno con altre ottimizzazioni
  - Copy propagation, dead code elimination, …

Scelta della IR

- Occorre scegliere la IR col giusto livello di dettaglio per il compito da svolgere
- Occorre tenere a mente i costi legati alla manipolazione dei vari formati
- Non esiste un'unica IR perfetta per tutti gli scopi
- Tipicamente ne serve più d'una
  - Completamente separate, per funzioni diverse
  - **Forme ibride**, che combinano grafi e forme lineari
    - Es., **Control Flow Graph** con 3-Address Code

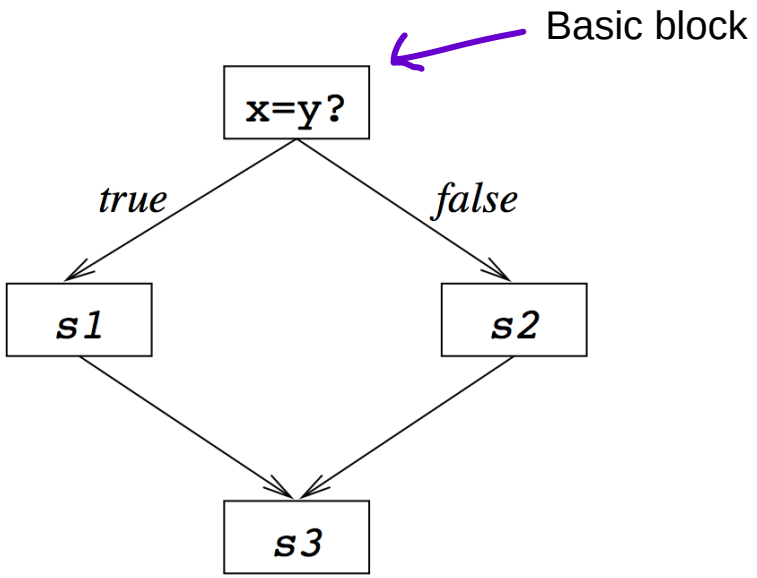
Control Flow Graph (CFG)

(Forma ibrida di Rappr. Intermedia -> ha sia codice che control flow)

- Fin qui abbiamo visto solo esempi di sequenze lineari di codice. Cosa succede se consideriamo istruzioni di salto?
- Occorre una IR che sia in grado di rappresentare il flusso di controllo (*control flow*) del programma

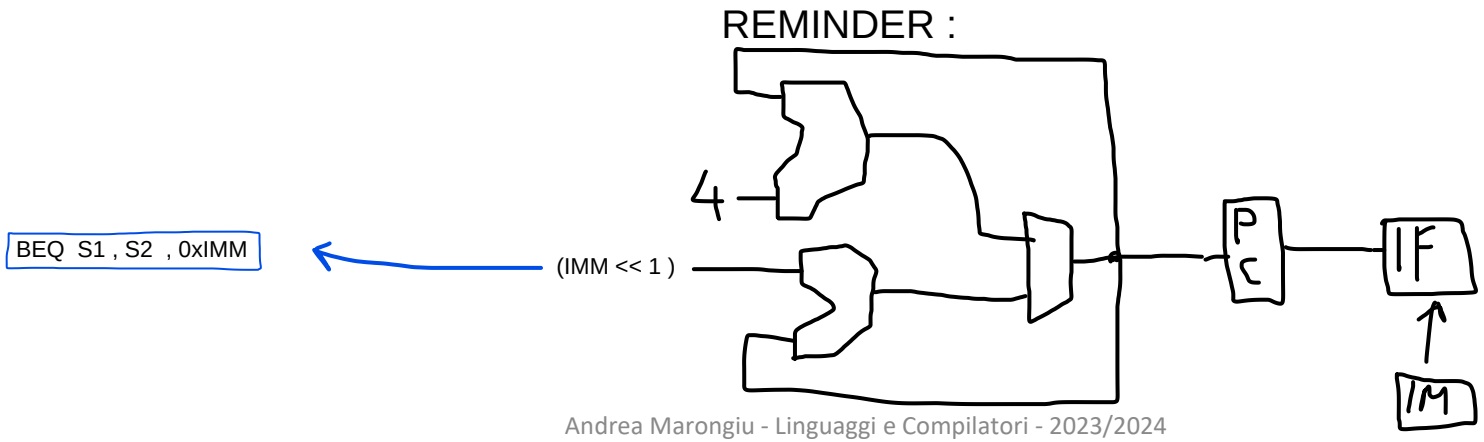
```
if (x == y)
   S1;
else
   S2;

S3
```



Control Flow Graph (CFG)

- Un CFG modella il trasferimento (flusso) del controllo in un programma tra **blocchi** di istruzioni
- I suoi nodi sono dei ***basic blocks***, che contengono una sequenza lineare di istruzioni, terminata da un'istruzione di trasferimento del controllo
- I suoi archi rappresentano il flusso di controllo (loop, if/else, goto, ecc.)



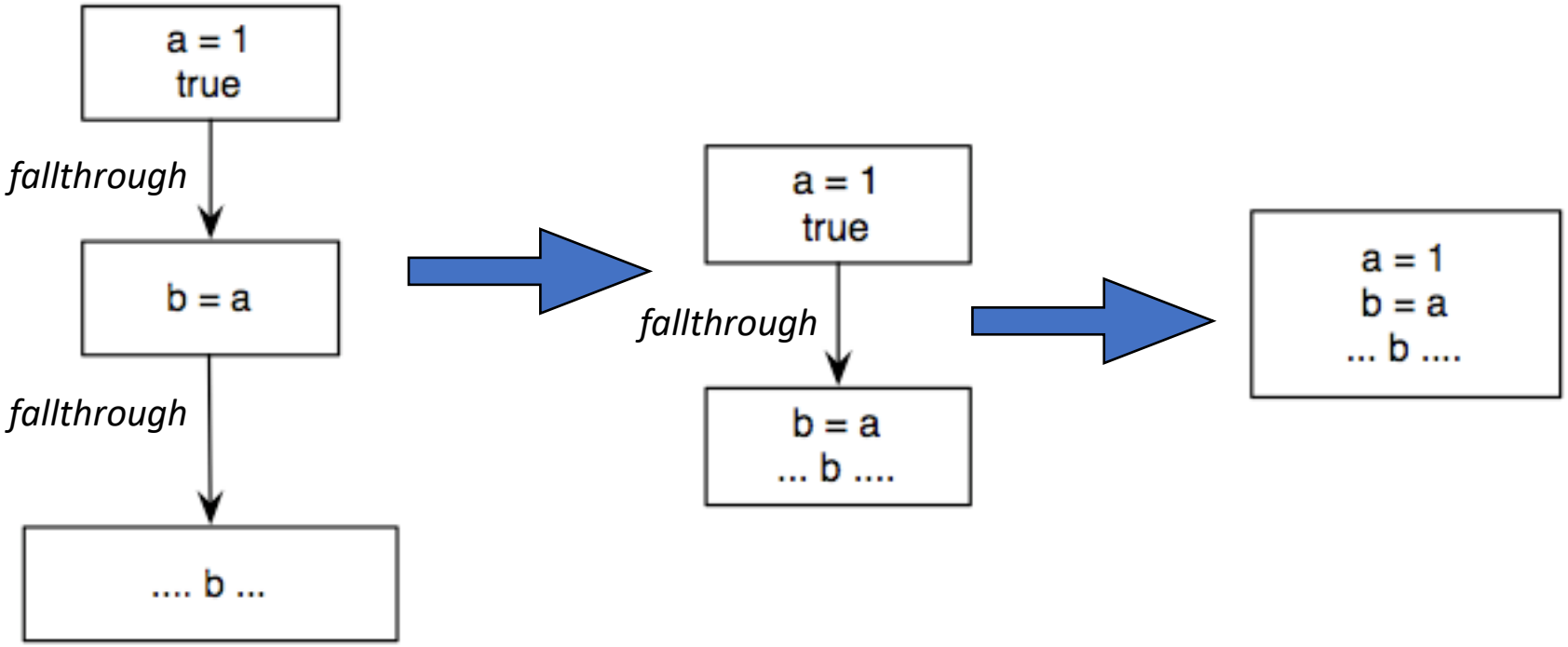
SESE -> Single Entry , Single Exit

Control Flow Graph (CFG)

- Un ***basic block*** $\mathbf{B_i}$ è una sequenza di istruzioni in forma 3AC
  - Solo la prima istruzione può essere raggiunta dall'esterno (gli archi non possono puntare a un'istruzione nel mezzo del blocco)
    - **Singolo entry point**
  - Tutte le istruzioni nel blocco sono eseguite se viene eseguita la prima (non è possibile eseguire un'istruzione di salto se non come ultima istruzione del blocco)
    - **Singolo exit point**
- Un arco connette due nodi $\mathbf{B_i} \rightarrow \mathbf{B_j}$ se e solo se $\mathbf{B_j}$ può eseguire dopo $\mathbf{B_i}$ in qualche percorso del flusso di controllo del programma
  - La prima istruzione di $\mathbf{B_j}$ è il target dell'istruzione di salto al termine di $\mathbf{B_i}$
  - $\mathbf{B_j}$ è l'unico successore di $\mathbf{B_i}$, che non ha un'istruzione di salto come ultima istruzione (*fallthrough*)

Control Flow Graph (CFG)

- Un CFG normalizzato ha i *basic blocks* **massimali**
  - Non possono essere resi più grandi senza violare le condizioni



Control Flow Graph (CFG)

Algoritmo per la costruzione del CFG

1. Identificare il LEADER di ogni *basic block*
   - La prima istruzione
   - Il target di un salto
   - Ogni istruzione che viene dopo un salto

2. Il *basic block* comincia con il LEADER e termina con l'istruzione immediatamente precedente un nuovo LEADER
   - o l'ultima istruzione

3. Connettere i *basic blocks* tramite archi di tre tipi
   - *fallthrough* (o fallthru): esiste solo un percorso che collega i due blocchi
   - *true*: il secondo blocco è raggiungibile dal primo se un condizionale è TRUE
   - *false*: il secondo blocco è raggiungibile dal primo se un condizionale è FALSE

Control Flow Graph (CFG) (Example)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

```
    i := n-1
S5: if i<1 goto s1
    j := 1
s4: if j>i goto s2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]    ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    ;A[j+1]
    if t3<=t7 goto s3
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
s3: j := j+1
    goto S4
S2: i := i-1
    goto s5
s1:
```

Control Flow Graph (CFG)

NOTA: Anche se sembrerebbe un unico BB devo spezzarlo in due perché la seconda istruzione ha una label (può essere raggiunta dall'esterno)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

```
    i := n-1                    BB1
S5: if i<1 goto s1              BB2
    j := 1
s4: if j>i goto s2
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]    ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    ;A[j+1]
    if t3<=t7 goto s3
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
s3: j := j+1
    goto S4
S2: i := i-1
   goto s5
s1:
```

Control Flow Graph (CFG)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

```
    i := n-1                    BB1
S5: if i<1 goto s1              BB2
    j := 1                      BB3
s4: if j>i goto s2              BB4
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]     ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]     ;A[j+1]
    if t3<=t7 goto s3
```

Stesso caso di prima

```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
s3: j := j+1
    goto S4
S2: i := i-1
   goto s5
s1:
```

Control Flow Graph (CFG)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

```
    i := n-1                    BB1
S5: if i<1 goto s1              BB2
    j := 1                      BB3
s4: if j>i goto s2              BB4
    t1 := j-1                   BB5
    t2 := 4*t1
    t3 := A[t2]     ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]     ;A[j+1]
    if t3<=t7 goto s3
```

Questo è un BB standard

```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
s3: j := j+1
    goto S4
S2: i := i-1
   goto s5
s1:
```

Control Flow Graph (CFG)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

Stessa situazione di prima

BB1
```
    i := n-1
```

BB2
```
S5: if i<1 goto s1
```

BB3
```
    j := 1
```

BB4
```
s4: if j>i goto s2
```

BB5
```
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]    ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    ;A[j+1]
    if t3<=t7 goto s3
```

BB6
```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
```

BB7
```
s3: j := j+1
    goto S4
S2: i := i-1
   goto s5
s1:
```

Control Flow Graph (CFG)

Costruire il CFG dal 3AC

1. Identificare i *basic blocks*

BB1
```
    i := n-1
```

BB2
```
S5: if i<1 goto s1
```

BB3
```
    j := 1
```

BB4
```
s4: if j>i goto s2
```

BB5
```
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]    ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    ;A[j+1]
    if t3<=t7 goto s3
```

BB6
```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
```

BB7
```
s3: j := j+1
    goto S4
```

BB8
```
S2: i := i-1
   goto s5
```

BB9
```
s1:
```

Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi

BB1
```
    i := n-1
```

BB2
```
S5: if i<1 goto s1
```

BB3
```
    j := 1
```

BB4
```
s4: if j>i goto s2
```

BB5
```
    t1 := j-1
    t2 := 4*t1
    t3 := A[t2]    ;A[j]
    t4 := j+1
    t5 := t4-1
    t6 := 4*t5
    t7 := A[t6]    ;A[j+1]
    if t3<=t7 goto s3
```

BB6
```
    t8 :=j-1
    t9 := 4*t8
    temp := A[t9]   ;A[j]
    t10 := j+1
    t11:= t10-1
    t12  := 4*t11
    t13 := A[t12]   ;A[j+1]
    t14 := j-1
    t15 := 4*t14
    A[t15] := t13 ;A[j]:=A[j+1]
    t16 := j+1
    t17 := t16-1
    t18 := 4*t17
    A[t18]:=temp   ;A[j+1]:=temp
```

BB7
```
s3: j := j+1
    goto S4
```

BB8
```
S2: i := i-1
   goto s5
```

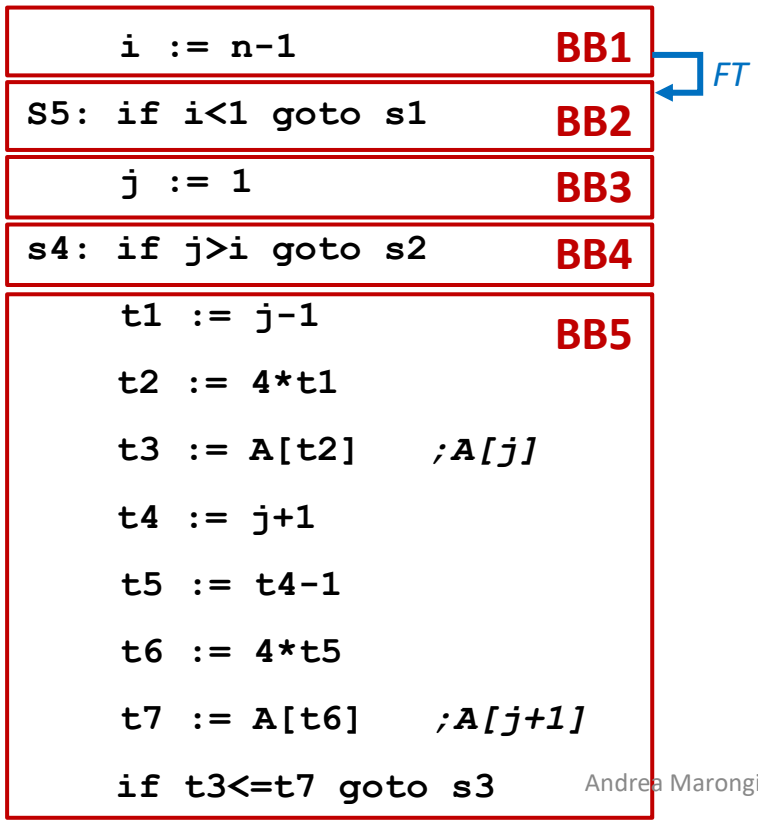
BB9
```
s1:
```

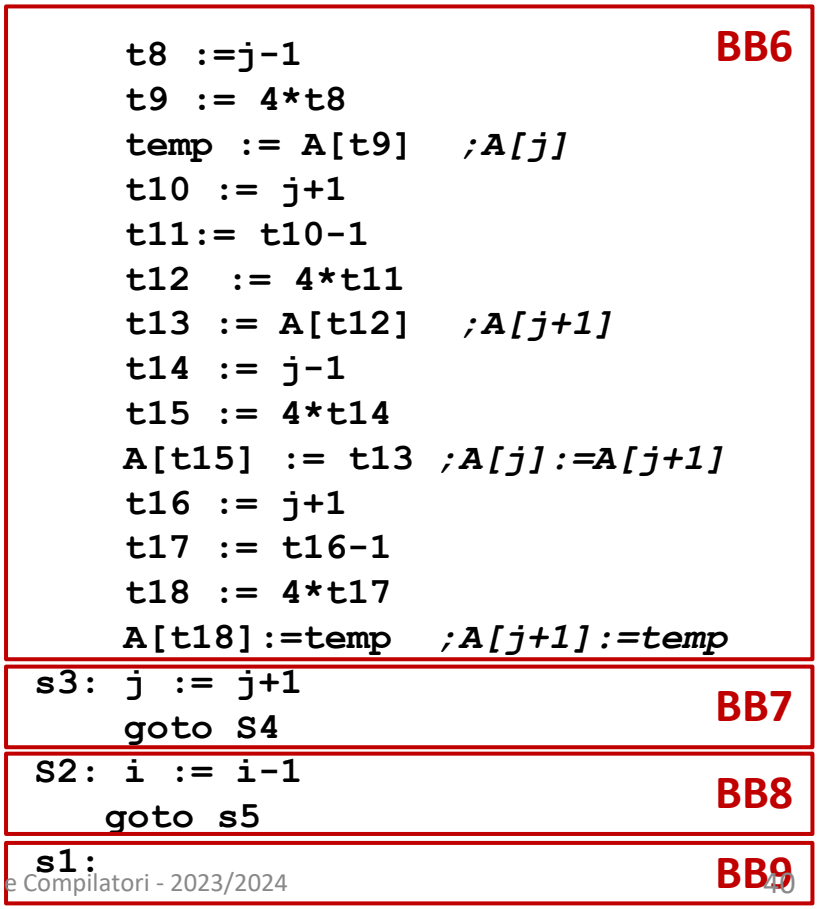
Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi



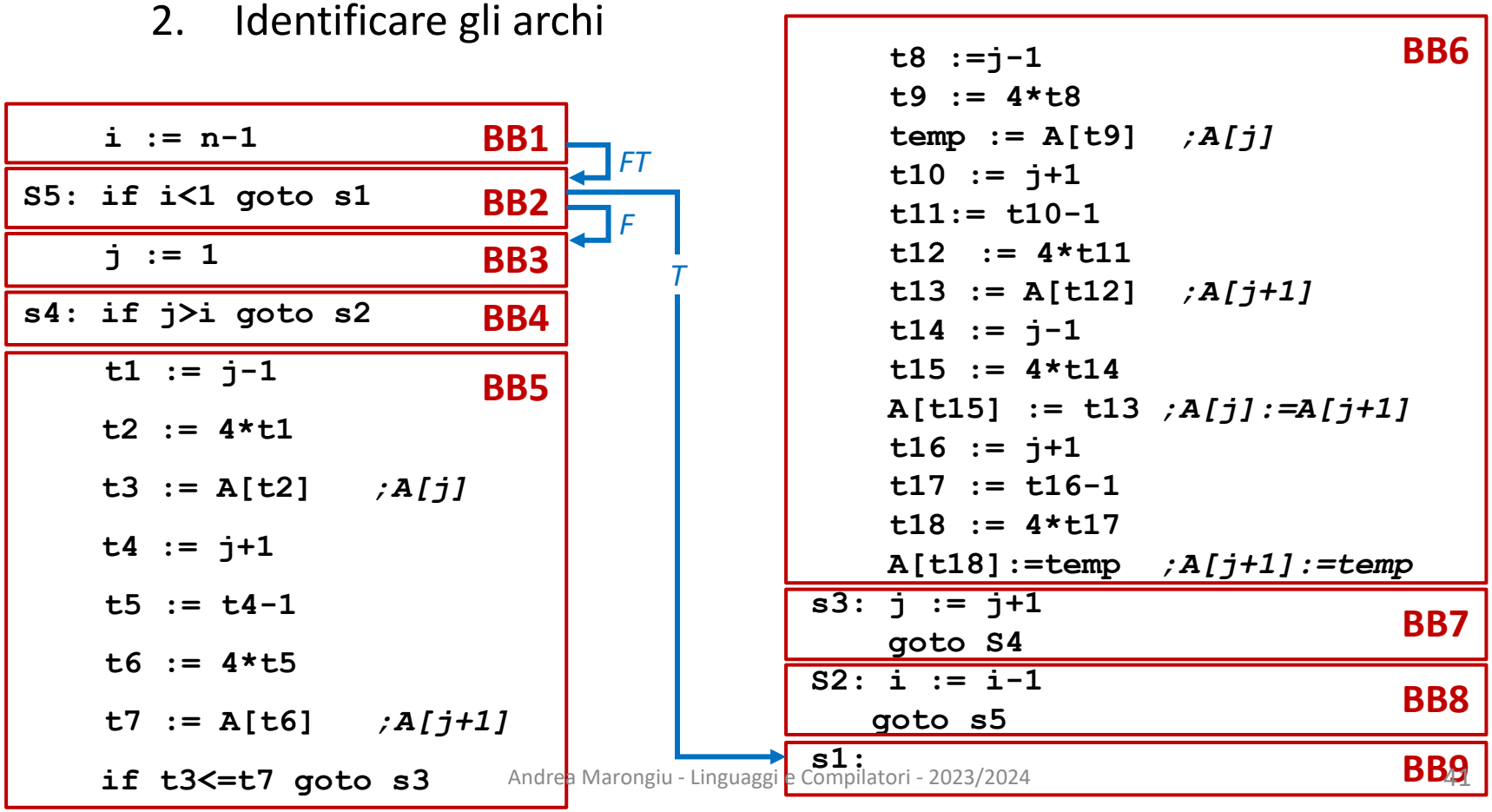


Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi

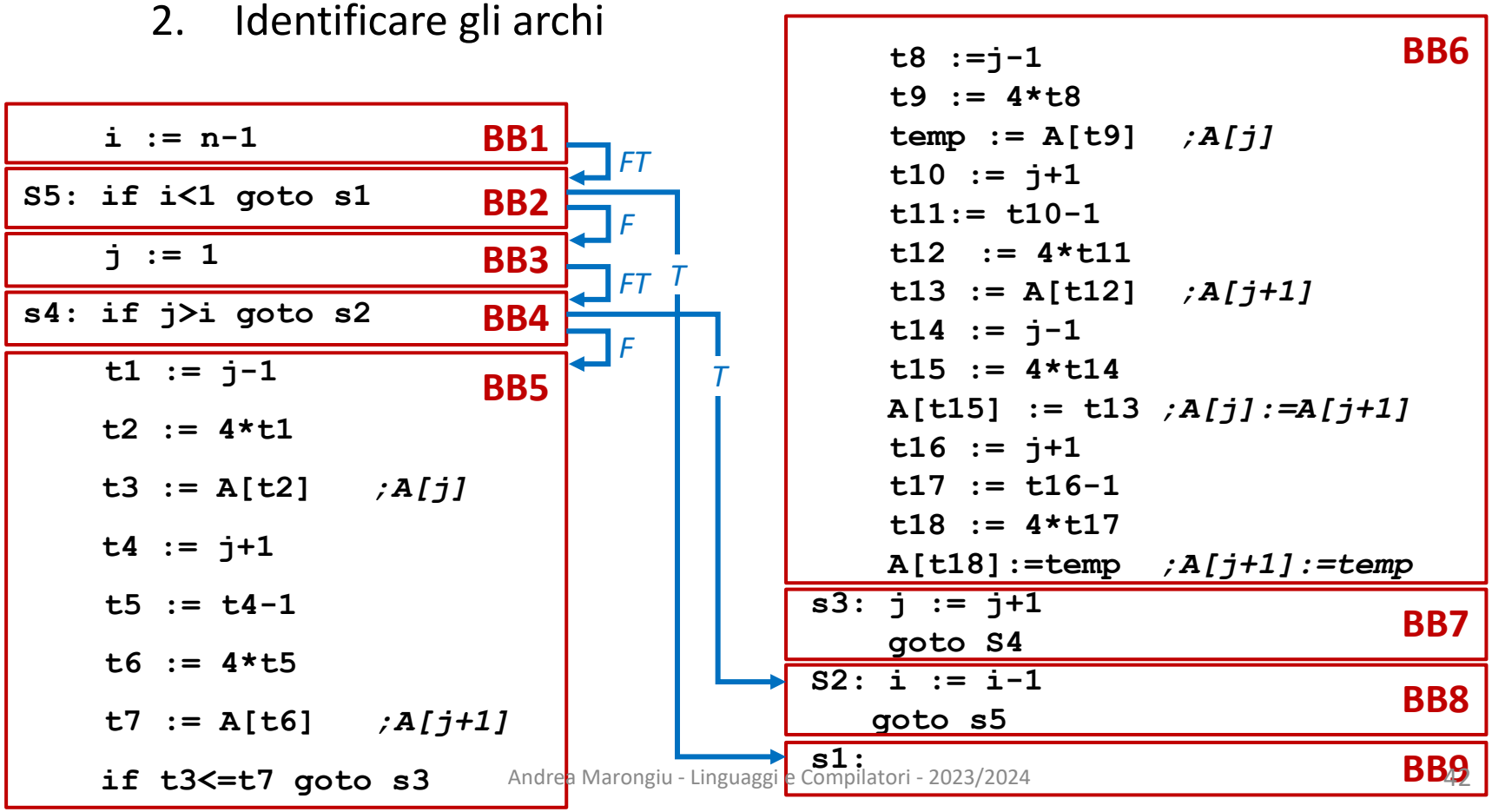


Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi

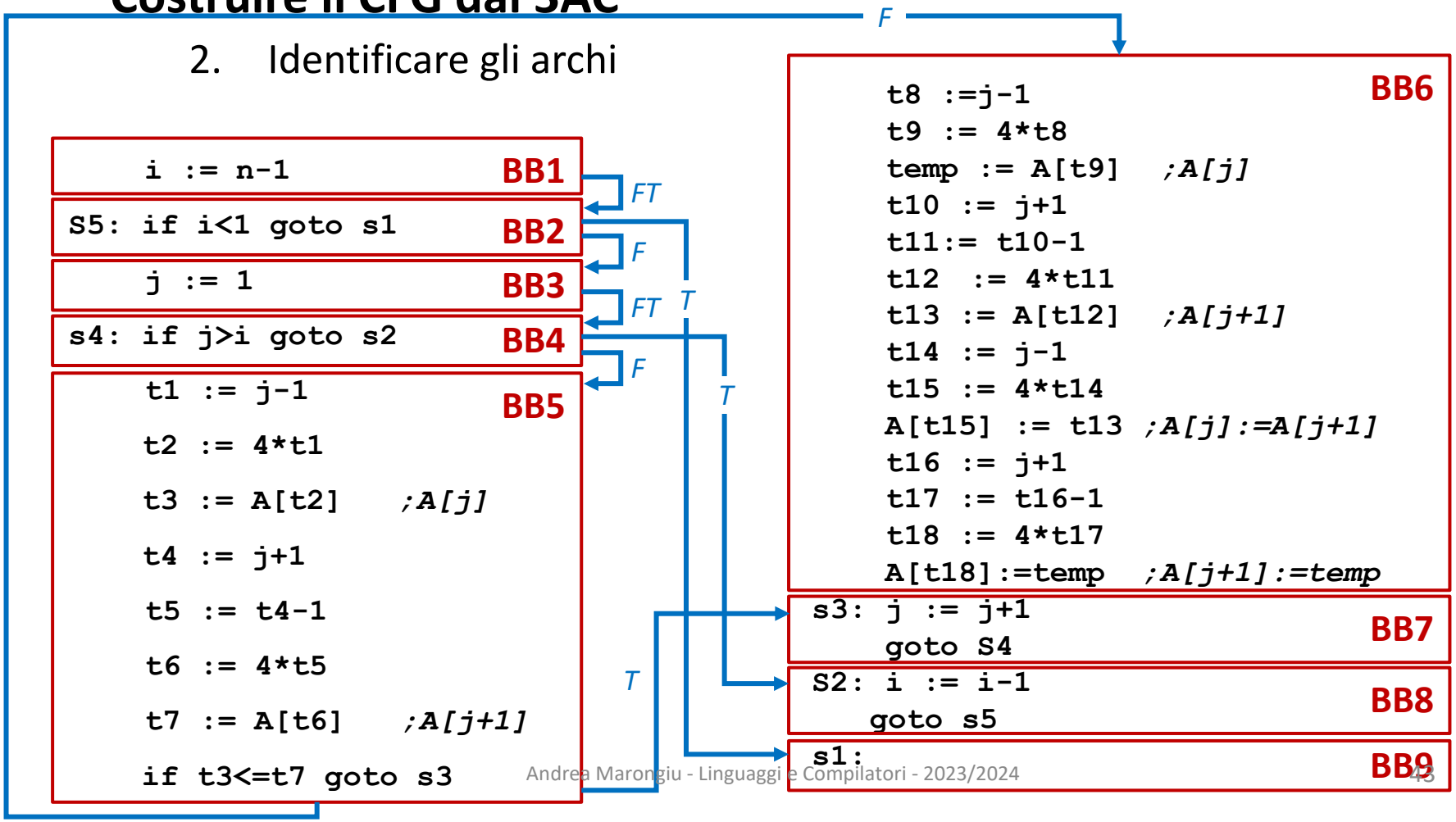


Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi

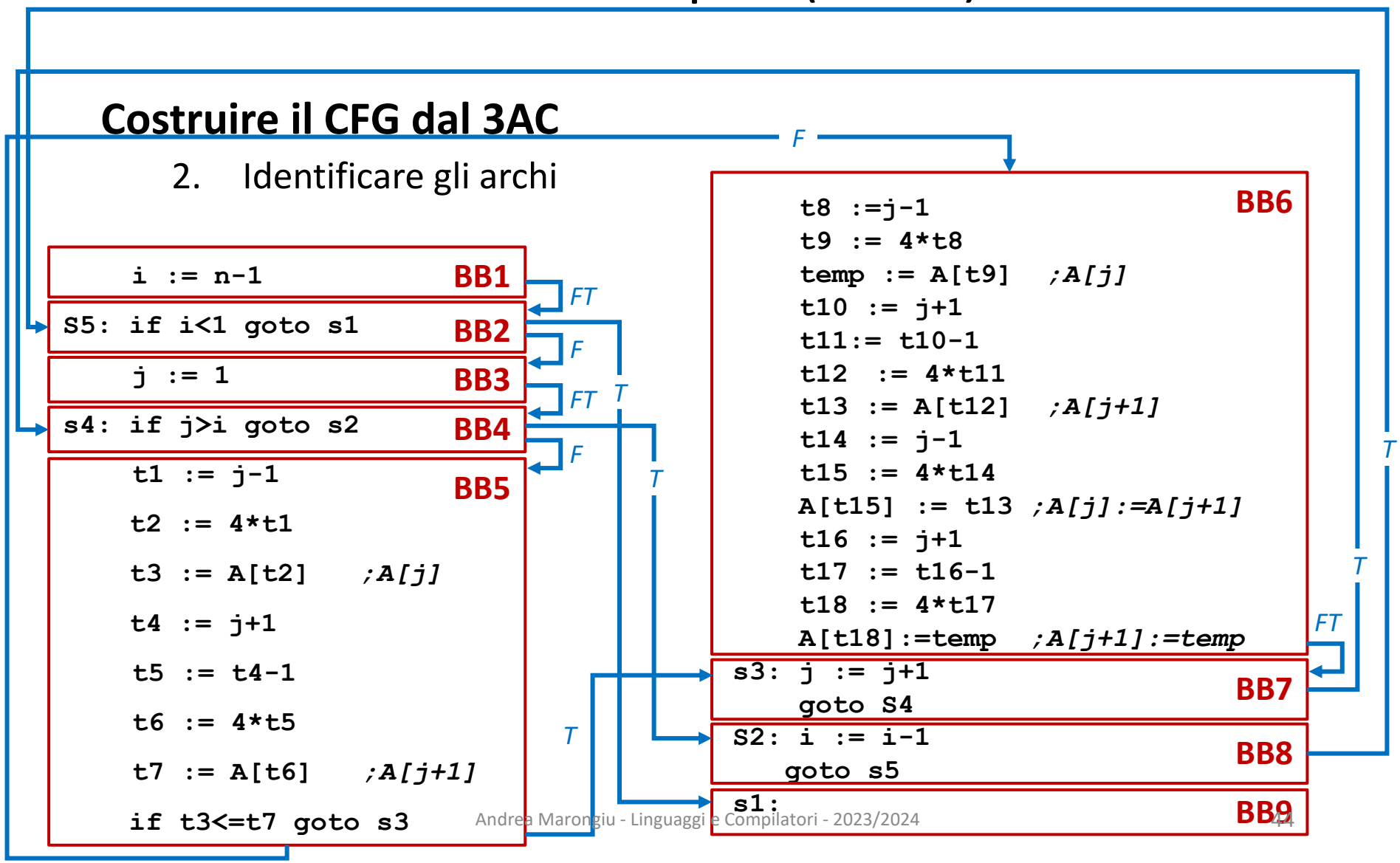


Control Flow Graph (CFG)

Tipi di arco:
FT – fallthrough
F – false
T – true

Costruire il CFG dal 3AC

2. Identificare gli archi

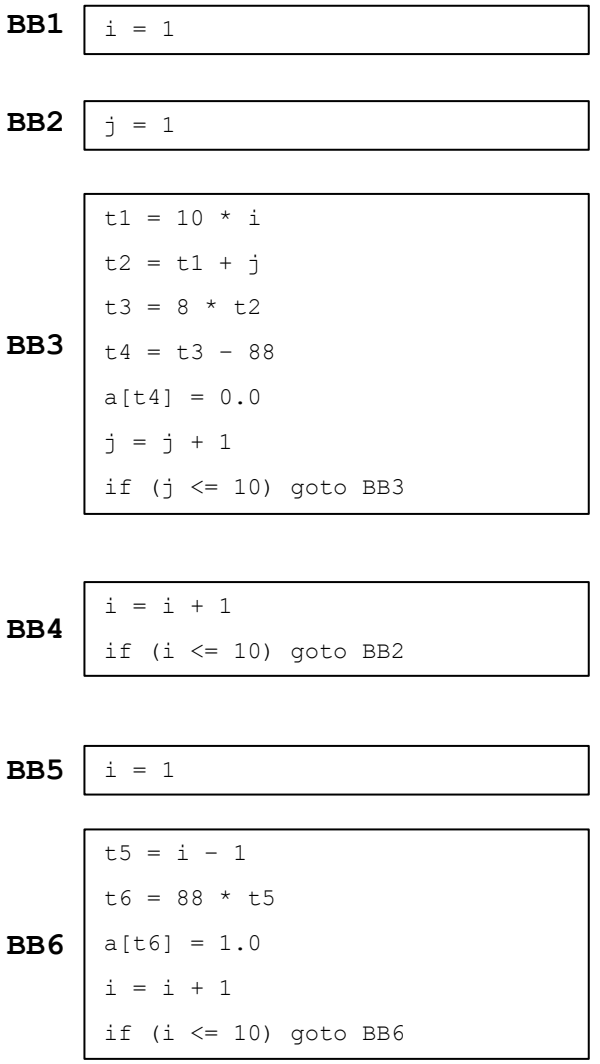


Esercizio 1: Ricavare il CFG

```
          i = 1
L1:       j = 1
L2:       t1 = 10 * i
          t2 = t1 + j
          t3 = 8 * t2
          t4 = t3 - 88
          a[t4] = 0.0
          j = j + 1
          if (j <= 10) goto L2
          i = i + 1
          if (i <= 10) goto L1
          i = 1
L3:       t5 = i - 1
          t6 = 88 * t5
          a[t6] = 1.0
          i = i + 1
          if (i <= 10) goto L3
```

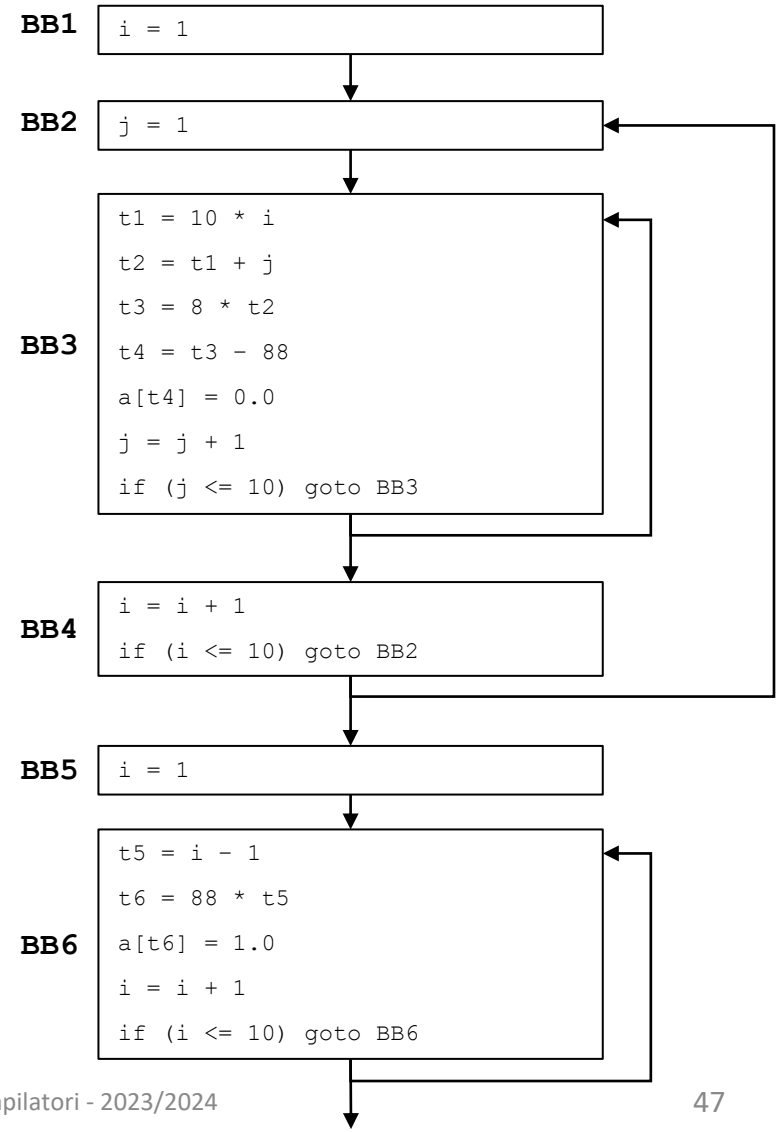
Esercizio 1: Ricavare il CFG

```
        i = 1
L1:     j = 1
L2:     t1 = 10 * i
        t2 = t1 + j
        t3 = 8 * t2
        t4 = t3 - 88
        a[t4] = 0.0
        j = j + 1
        if (j <= 10) goto L2
        i = i + 1
        if (i <= 10) goto L1
        i = 1
L3:     t5 = i - 1
        t6 = 88 * t5
        a[t6] = 1.0
        i = i + 1
        if (i <= 10) goto L3
```



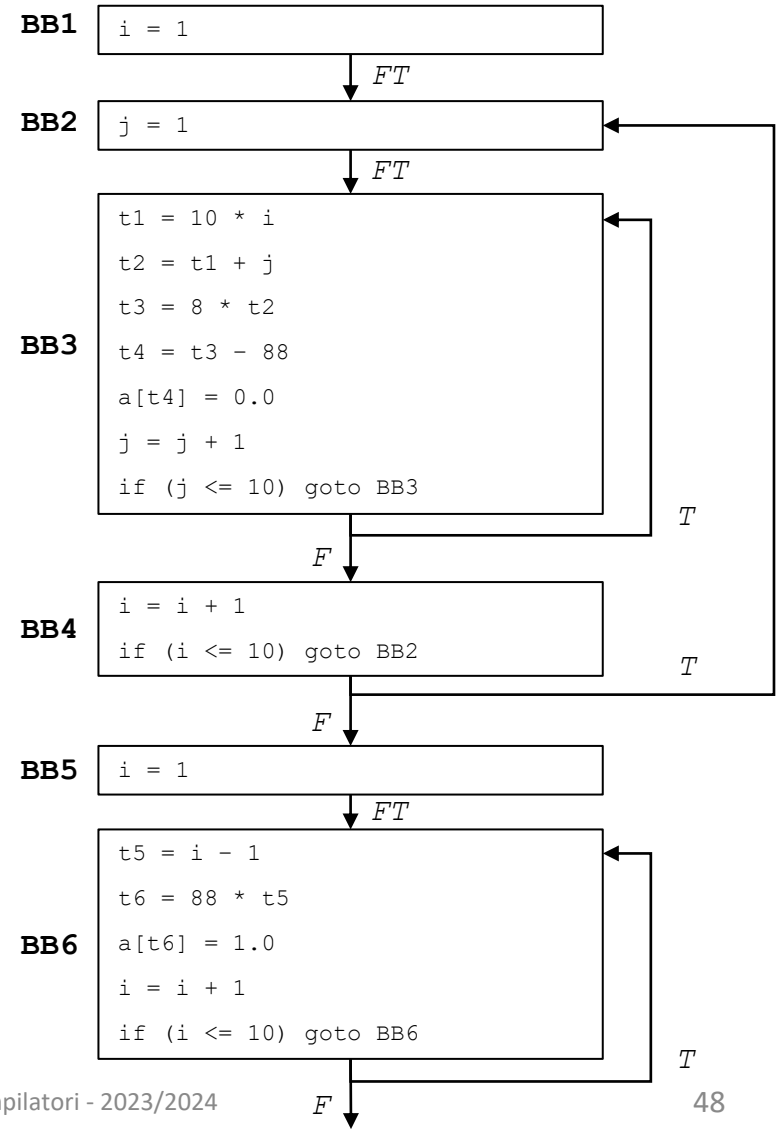
Esercizio 1: Ricavare il CFG

```
        i = 1
L1:     j = 1
L2:     t1 = 10 * i
        t2 = t1 + j
        t3 = 8 * t2
        t4 = t3 - 88
        a[t4] = 0.0
        j = j + 1
        if (j <= 10) goto L2
        i = i + 1
        if (i <= 10) goto L1
        i = 1
L3:     t5 = i - 1
        t6 = 88 * t5
        a[t6] = 1.0
        i = i + 1
        if (i <= 10) goto L3
```



Esercizio 1: Ricavare il CFG

```
        i = 1
L1:     j = 1
L2:     t1 = 10 * i
        t2 = t1 + j
        t3 = 8 * t2
        t4 = t3 - 88
        a[t4] = 0.0
        j = j + 1
        if (j <= 10) goto L2
        i = i + 1
        if (i <= 10) goto L1
        i = 1
L3:     t5 = i - 1
        t6 = 88 * t5
        a[t6] = 1.0
        i = i + 1
        if (i <= 10) goto L3
```

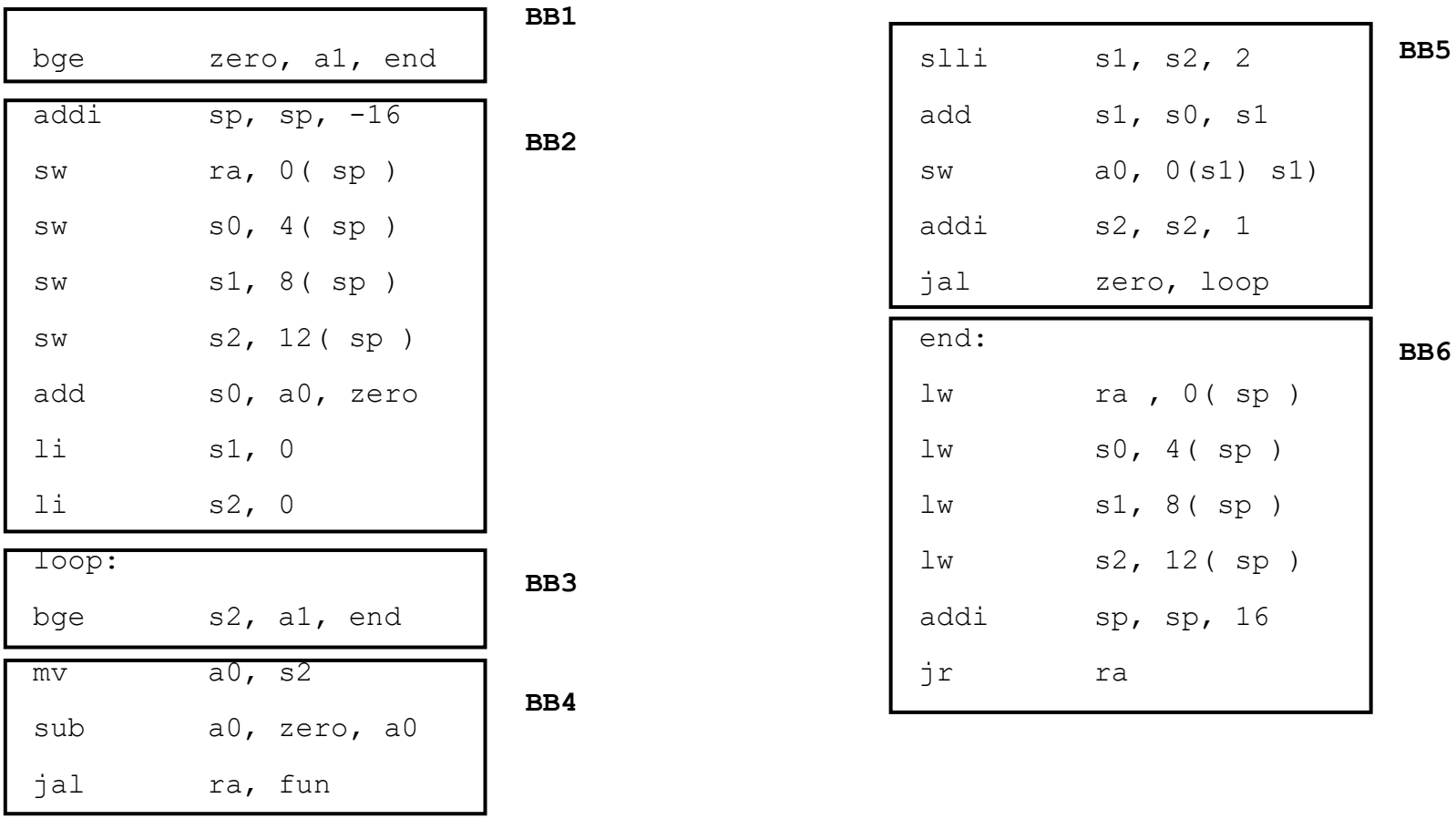


Esercizio 2: Ricavare il CFG

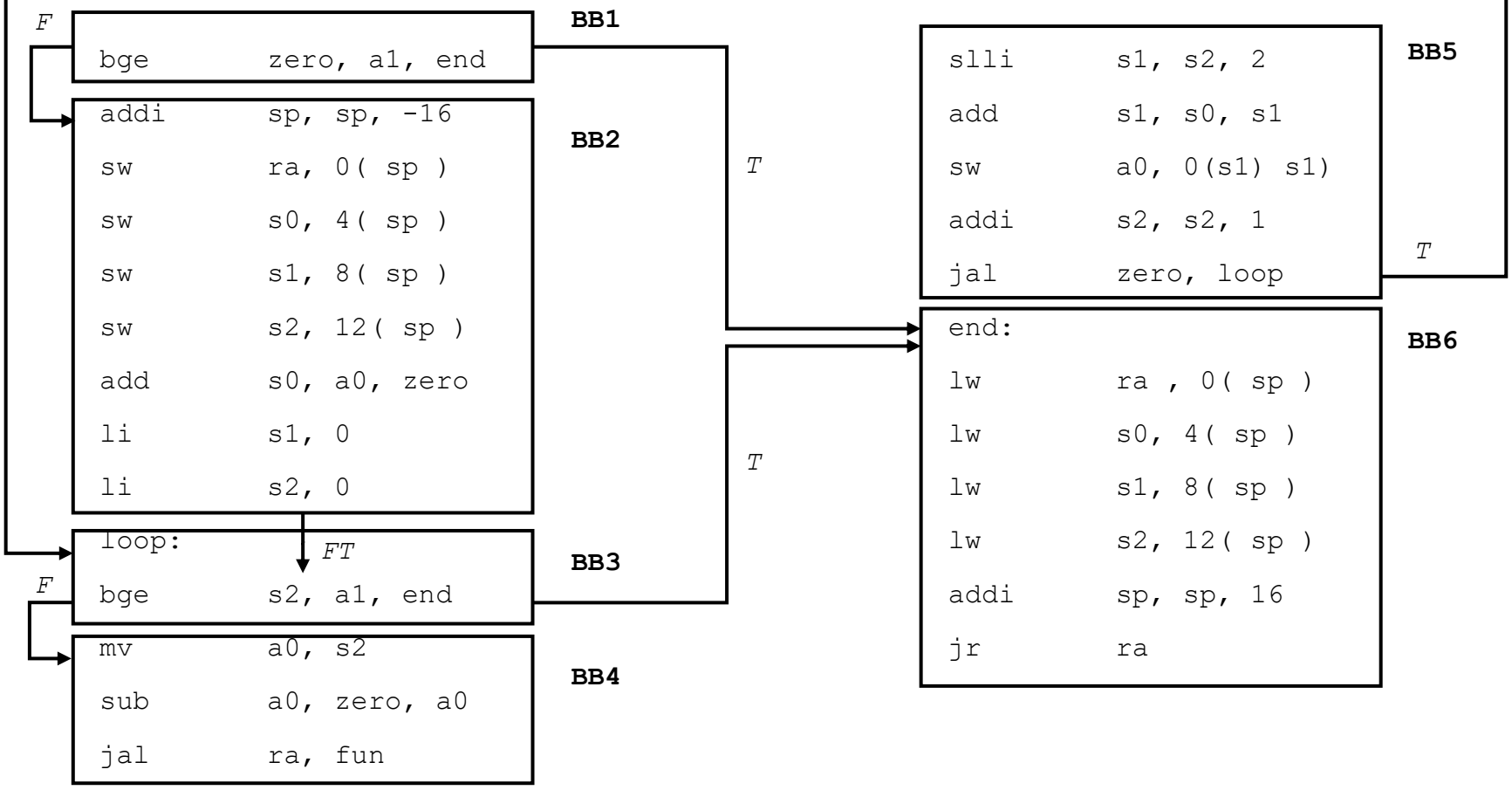
```
bge       zero, a1, end
addi      sp, sp, -16
sw        ra, 0( sp )
sw        s0, 4( sp )
sw        s1, 8( sp )
sw        s2, 12( sp )
add       s0, a0, zero
li        s1, 0
li        s2, 0
loop:
bge       s2, a1, end
mv        a0, s2
sub       a0, zero, a0
jal       ra, fun
```

```
slli      s1, s2, 2
add       s1, s0, s1
sw        a0, 0(s1) s1)
addi      s2, s2, 1
jal       zero, loop
end:
lw        ra , 0( sp )
lw        s0, 4( sp )
lw        s1, 8( sp )
lw        s2, 12( sp )
addi      sp, sp, 16
jr        ra
```

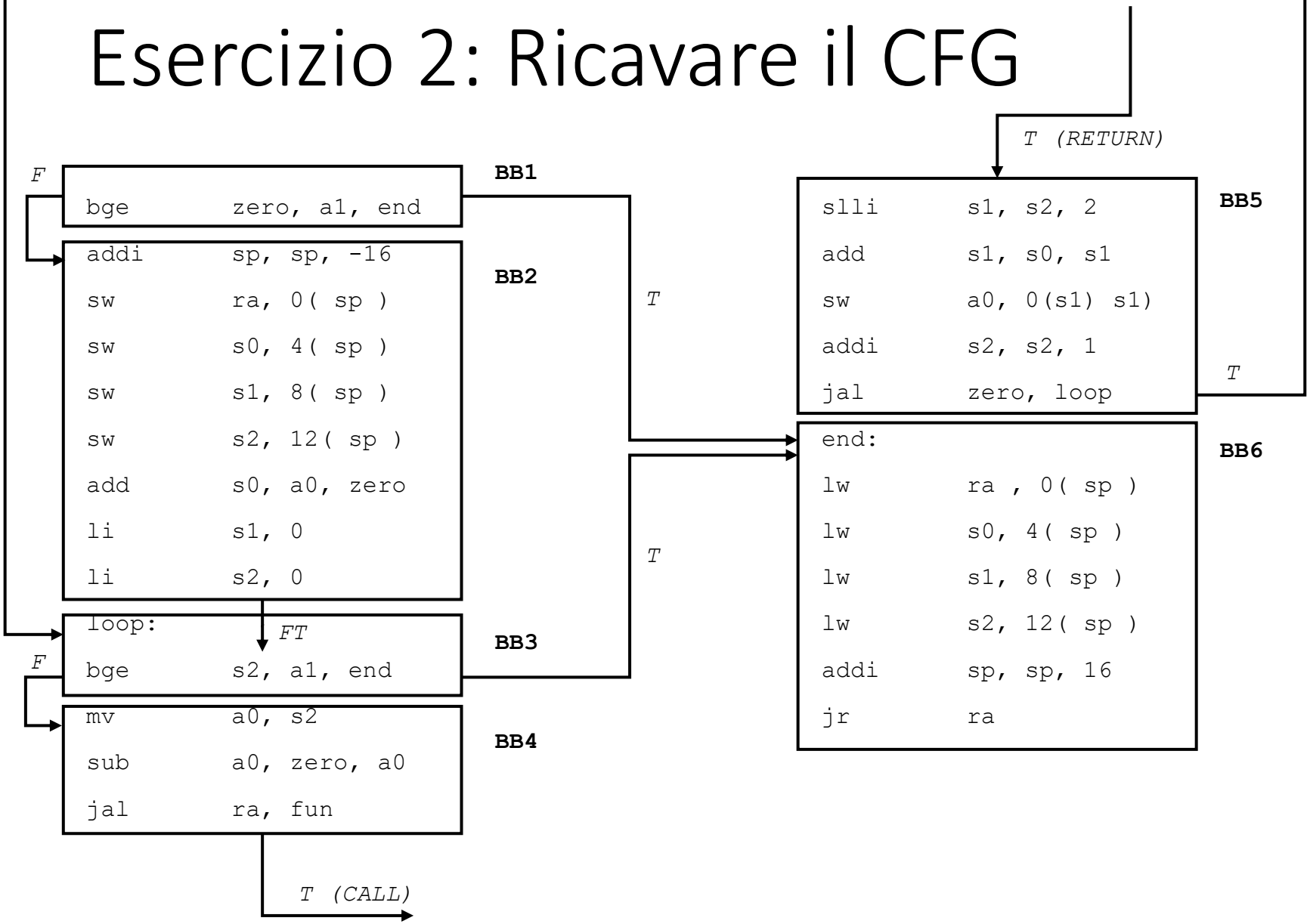
Esercizio 2: Ricavare il CFG



Esercizio 2: Ricavare il CFG



Esercizio 2: Ricavare il CFG

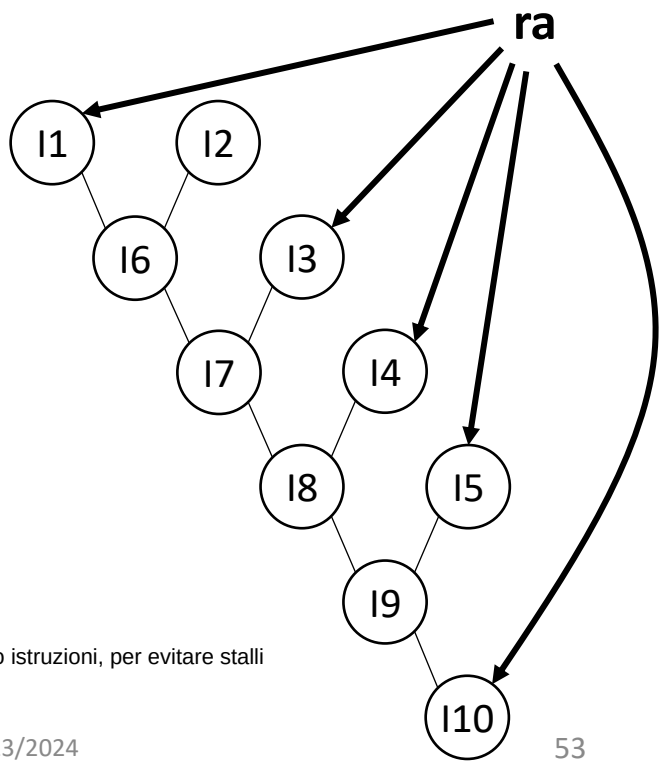


Dependency Graph

a livello di istruzioni

- I nodi di un DG sono istruzioni, che *usano* dei valori e ne *definiscono* un altro
- Gli archi di un DG connettono due nodi, uno dei quali *usa* i risultati *definiti* dall'altro

```
I1:   ld rw, ra, 0
I2:   li r2, 2
I3:   ld rx, ra, 12
I4:   ld ry, ra, 16
I5:   ld rz, ra, 20
I6:   mul rw, rw, r2
I7:   mul rw, rw, rx
I8:   mul rw, rw, ry
I9:   mul rw, rw, rz
I10: sd  rw, ra, 0
```



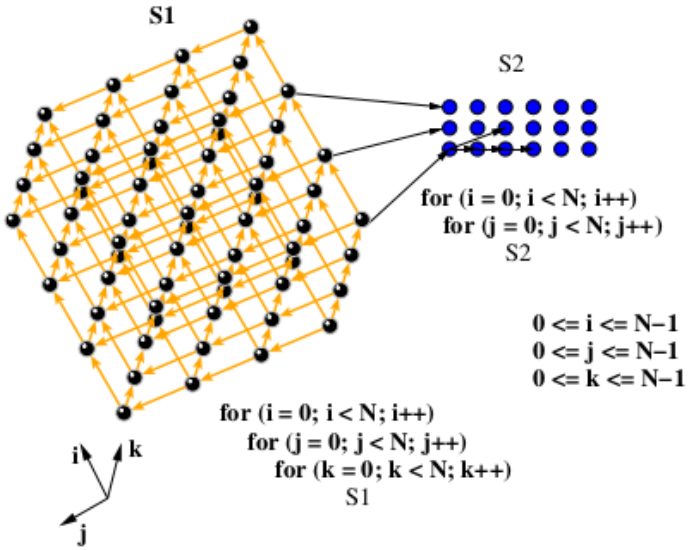
Indispensabili per *instruction scheduling*

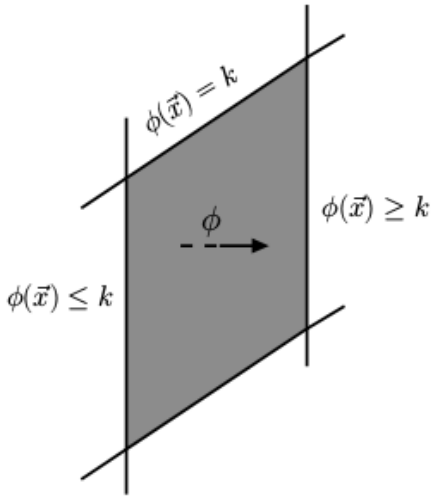
Riordino istruzioni, per evitare stalli

In LLVM c'è un modulo chiamato POLLY.
in GCC c'è un modulo chiamato GRAPHITE.

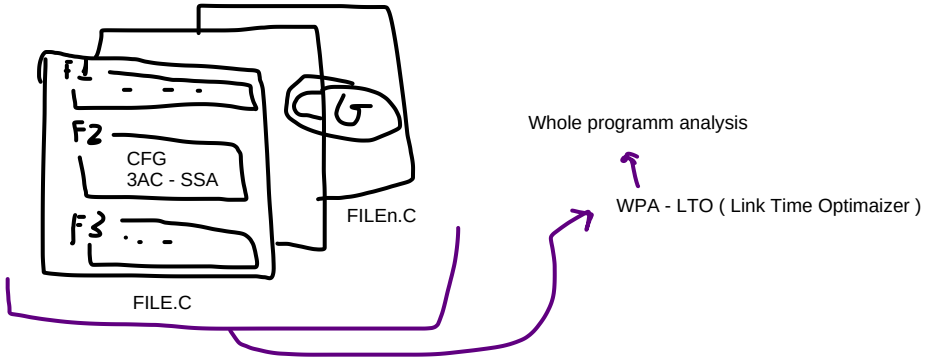
Data Dependency Graph (DDG)

- I loop innestati sono dei buoni candidati per il parallelismo, purché non ci siano dipendenze di dato tra le iterazioni
- Il *polyhedral model* è una sorta di DDG avanzato che tratta ogni iterazione del loop come un punto all'interno di un modello matematico (il poliedro) che semplifica la trasformazione dello spazio delle iterazioni
- In questo modo vengono individuati ordini di attraversamento dello spazio delle iterazioni che non sono soggetti a dipendenze
  - **E quindi possono essere parallelizzati**





Call Graph



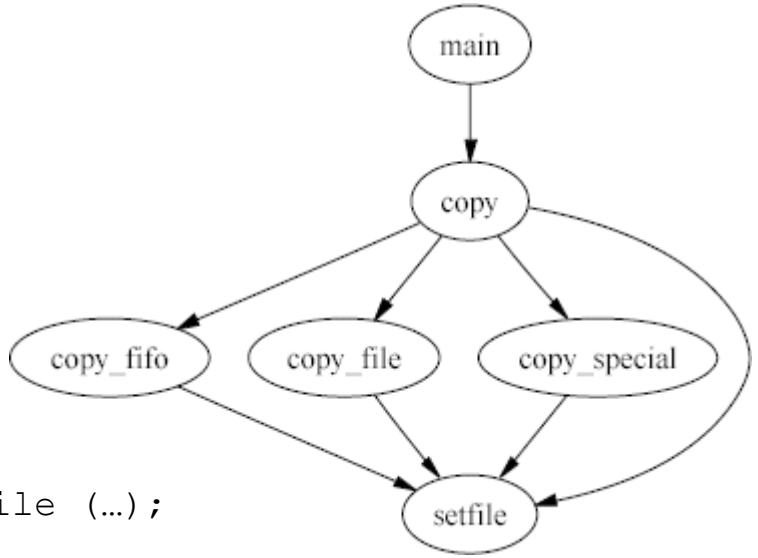
- Mostra l'insieme delle (potenziali) chiamate tra le funzioni, gerarchicamente

```
void main ()
{
     return copy (…);
}

int copy (…)
{
     copy_fifo (…);
     copy_file (…);
     copy_special (…);
     return setfile (…);
}
int copy_fifo ()
{
     return setfile (…);
}
```

```
int copy_file ()
{
     return setfile (…);
}

int copy_special ()
{
     return setfile (…);
}
```



Ricorda: Proprietà di una IR

- In un compilatore ci possono essere (e di fatto ci sono) diversi tipi di IR che coesistono
- Trade-off tra vari parametri:
  - Facilità di generazione
  - Facilità di manipolazione
  - Costo di manipolazione
  - Livello di astrazione
  - Livello di dettaglio esposto

- L'implementazione di una IR ha effetti molto intricati sulla velocità ed efficienza di un compilatore

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

2. Rappresentazione Intermedia

Linguaggi e Compilatori [I215-011]

Corso di Laurea in INFORMATICA
(D.M.270/04) [16-262]
Anno accademico 2022/2023

Prof. Andrea Marongiu
andrea.marongiu@unimore.it