UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

1. Introduzione al corso

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

- Marwedel, Embedded System Design, Springer 2018,
- Wolf, Computers as Components 4th Ed., Morgan Kaufmann 2016
- Wolf, High Performance Embedded Computing 2nd Ed., Morgan Kaufmann 2014
- Lee, Seshia, Introduction to Embedded Systems, A Cyber-Physical Systems Approach, 2nd Ed., MIT Press, 2017
- Denker, University of Bern: “Compiler Construction”
- Hua, University of Science and Technology of China: “Compiler”
- Pekhimenko, University of Toronto, “Compiler Optimization”

Prof. Andrea Marongiu

- **Professore Associato**
  andrea.marongiu@unimore.it
  http://personale.unimore.it/rubrica/dettaglio/amarongiu

- **Ufficio**: MO-18-02-027 (secondo piano)
- PhD dall'University of Bologna
- PostDoc presso ETH Zurich
- Visiting Researcher presso INRIA (Paris, FR), Brown University (Providence, RI – USA), NVIDIA (Santa Clara, CA – USA)
- **Interessi di ricerca**: computer architecture, heterogeneous embedded systems, programming models, compilers, hardware acceleration

- **Ricevimento studenti**: Per appuntamento, da concordare via email

Materiali didattici

- **Slides e codice (pubblicato su Moodle e Github):**
  - Le slides contengono tutta l'informazione necessaria per preparare l'esame
  - Laboratori pratici a cadenza settimanale per un'esperienza *hands-on* sul codice

- **Libri di testo (per riferimento, opzionali):**
  1. Engineering a Compiler
     Cooper, Torczon
     Elsevier
  2. Compilatori: principi, tecniche e strumenti – seconda edizione
     Aho, Lam, Sethi, Ullman
     Pearson

- Ulteriori materiali verranno forniti durante il corso delle lezioni

Prerequisiti e Obiettivi

- **Prerequisiti (!!!)**
  - Architettura dei Calcolatori
  - Programmazione C++

- **Obiettivi**
  - Conoscenza di base delle principali ottimizzazioni nei compilatori
  - Esperienza pratica nell'implementazione di alcune semplici ottimizzazioni in un compilatore reale allo stato dell'arte (LLVM)
  - Principi fondamentali e infrastruttura per lo sviluppo di nuove ottimizzazioni

Laboratori

- Sono pensati per dare un’esperienza sul campo del processo di analisi/ottimizzazione del codice
- Si svolgeranno sui vostri PC, basandoci su tracce fornite di volta in volta
- Si assume (e si richiede) che abbiate installata la versione di riferimento di LLVM
  - più informazioni nelle prossime lezioni

Esame

- L’esame pesa il 50% del voto finale per l’insegnamento di Linguaggi e Compilatori
- Il voto preso in questa parte fa media con quello della parte precedente (Prof. Leoncini)
- Consiste in:
  - Lo svolgimento e la presentazione di varie attività progettuali da svolgere nel corso del semestre sottoforma di *assignments*
    - Si possono svolgere da soli o in piccoli gruppi
  - Una prova orale

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Motivazione

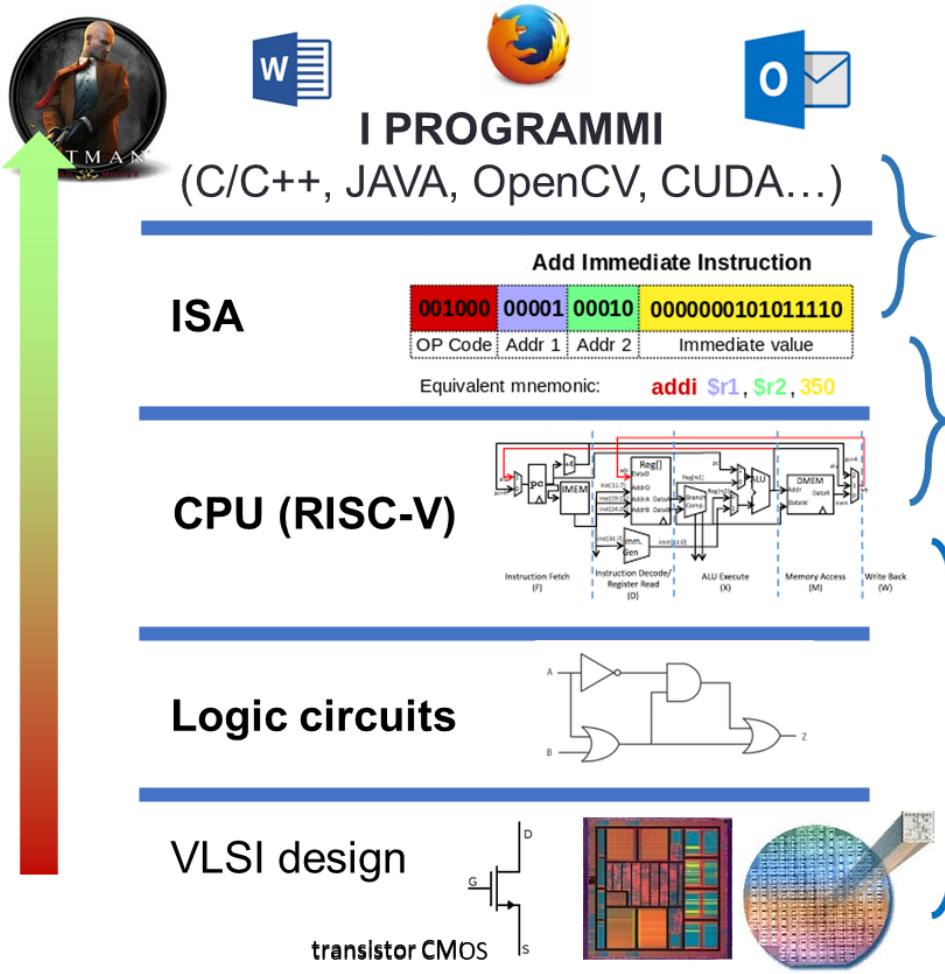
A cosa servono i compilatori?

- I computer sono **pervasivi** e **onnipresenti** in tutti gli aspetti della nostra vita quotidiana

- **Lo studio** di come sono progettati e programmati i computer è **fondamentale** in un mondo (e un mercato) che è dominato da questa tecnologia
  - *Per questo abbiamo i corsi di Architettura dei Calcolatori e Programmazione*
  - *Dove si inseriscono i **Compilatori** in questo scenario?*

L'astrazione nei calcolatori

- Dai **transistor** ai **programmi**



Ricordate il corso di Architettura dei Calcolatori?

Compilatori: Come i programmi sono tradotti in linguaggio macchina

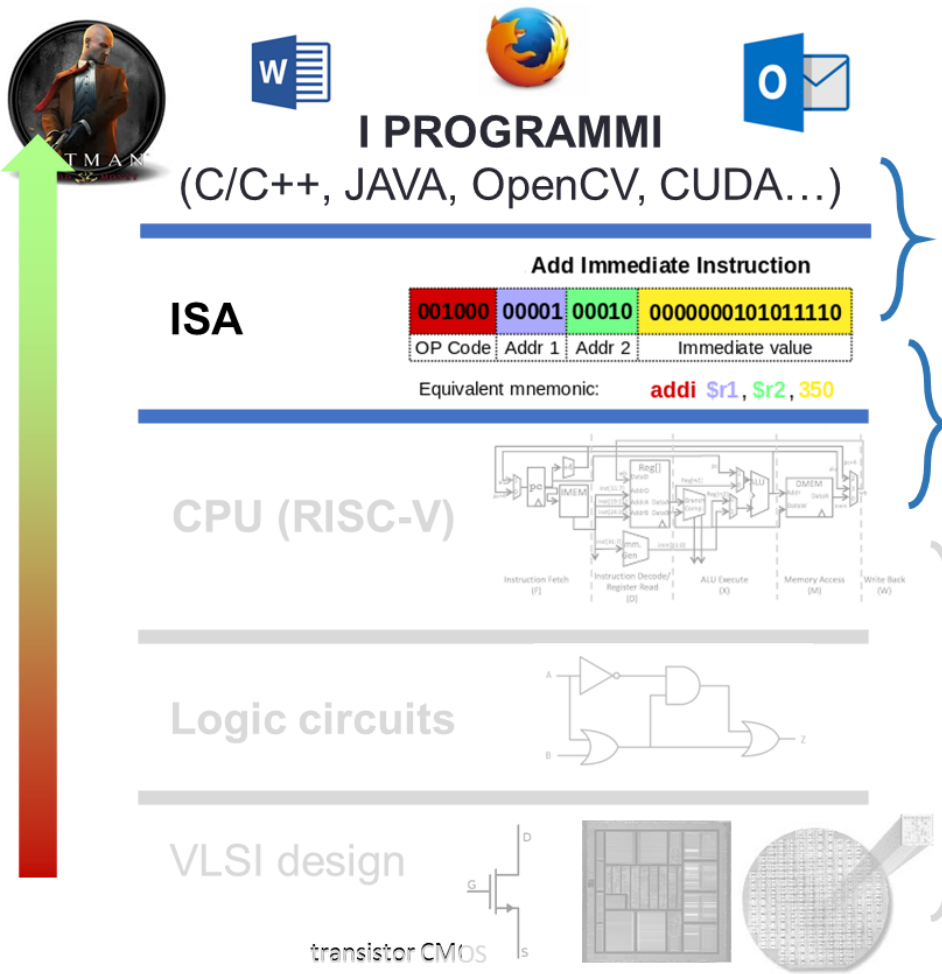
L'interfaccia hardware/software

- L'**Instruction Set Architecture** (ISA)

- Come l'hardware esegue il programma
- Ciò che determina la performance del programma e del sistema
- Come I progettisti hardware migliorano la performance

L'astrazione nei calcolatori

- Dai **transistor** ai **programmi**



I livelli coinvolti in questo corso

Compilatori: Come i programmi sono tradotti in linguaggio macchina

L'interfaccia hardware/software

– L'**Instruction Set Architecture** (ISA)

- Come l'hardware esegue il programma
- Ciò che determina la performance del programma e del sistema
- Come I progettisti hardware migliorano la performance

A cosa servono i compilatori?

- La loro funzione più nota è quella di **trasformare** il codice da un linguaggio all'altro
  - Es., convertono codice C in codice assembly RISC-V

A cosa servono i compilatori?

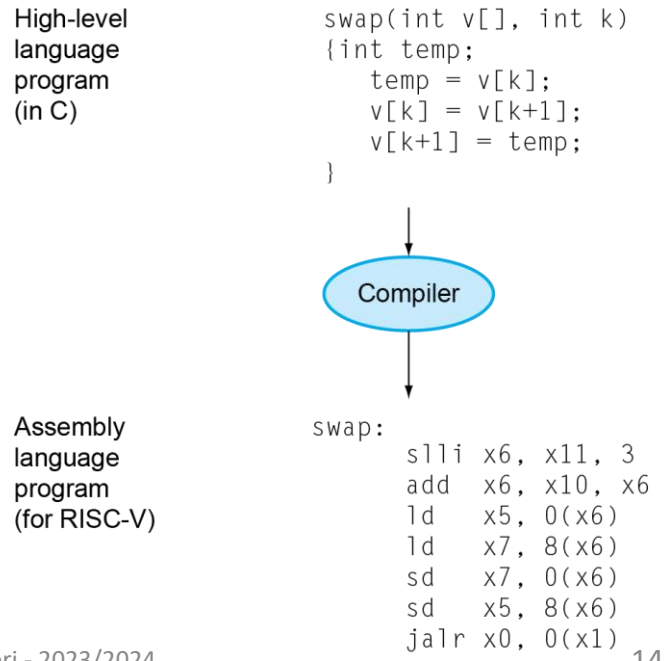
- La loro funzione più nota è quella di **trasformare** il codice da un linguaggio all'altro
  - Es., convertono codice C in codice assembly RISC-V

Linguaggio di alto livello (C, C++, Java)

Livello di astrazione prossimo al dominio del problema
Facilita la produttività e la portabilità

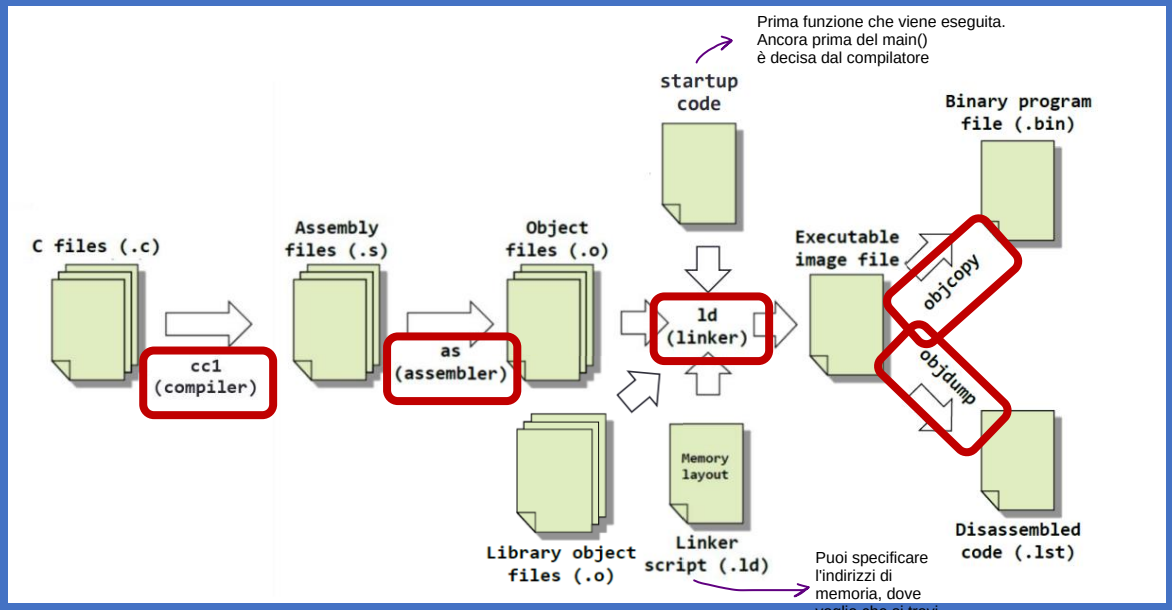
Linguaggio assembly

Una rappresentazione delle istruzioni macchina comprensibile agli umani



A cosa servono i compilatori?

NOTA: Il compilatore propriamente detto è solo uno degli strumenti di una *toolchain*, il cui scopo finale è quello di produrre un eseguibile per la CPU target



formare il

RISC-V

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

```
swap:
        slli x6, x11, 3
        add  x6, x10, x6
        ld   x5, 0(x6)
        ld   x7, 8(x6)
        sd   x7, 0(x6)
        sd   x5, 8(x6)
        jalr x0, 0(x1)
```

- Ricorda, quando la CPU fa una fetch, prende blocchi di bit da 32(x86) o 64(x64) bit

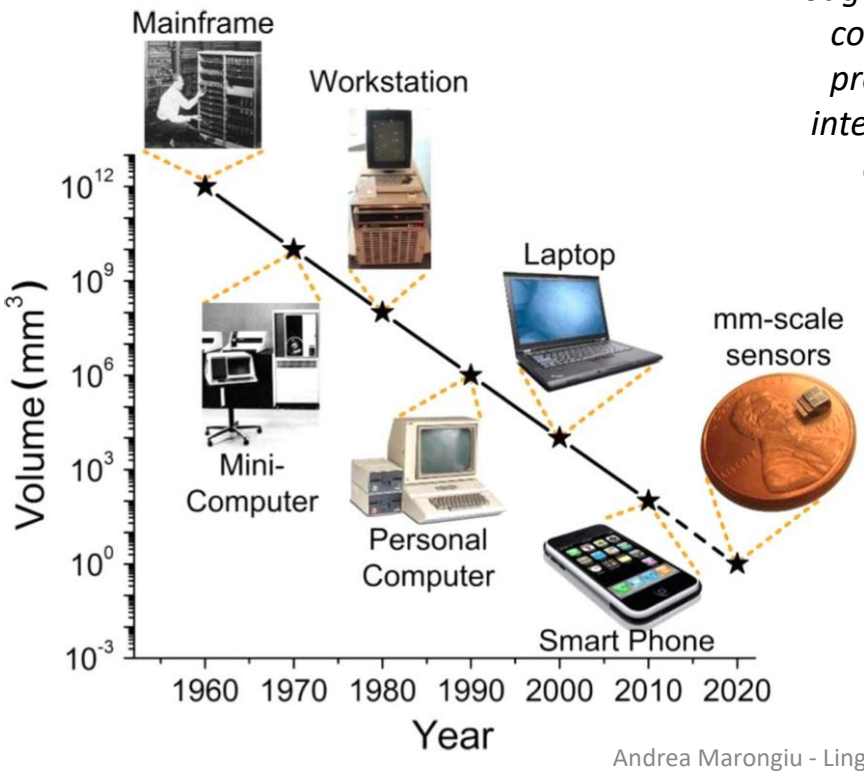
A cosa servono i compilatori?

- La loro funzione più nota è quella di **trasformare** il codice da un linguaggio all'altro
  - Es., convertono codice C in codice assembly RISC-V

- La loro altra funzione principale è quella di migliorare (**ottimizzare**) il codice
  - *Perché esegua più velocemente*
  - *Perché occupi meno spazio*
  - *Perché abbia migliore efficienza energetica*
  - *Perché sfrutti determinate caratteristiche architetturali (es., cache)*
  - *…*

Come evolvono i compilatori?

- Il ruolo dei compilatori, così come il loro progetto e implementazione, evolve con l'industria dei calcolatori



"Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry."

- Gordon Bell [1972 – 2008]

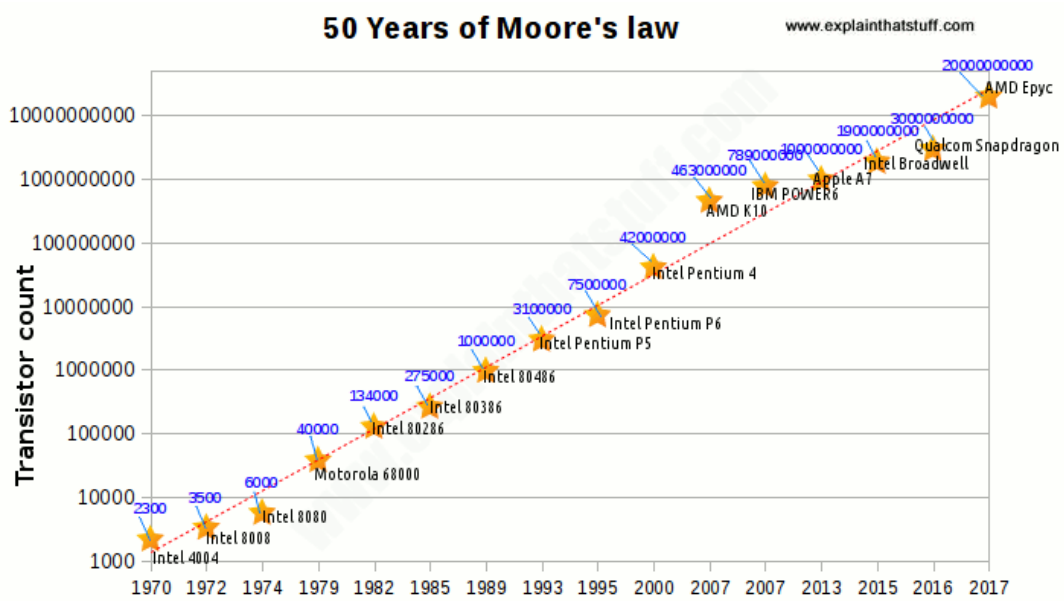
Direttore del progetto DEC VAX
Direttore della sezione ingegneria del NSF (1986-1987)
Ricercatore Emerito di Microsoft (1995-2015)

La legge di Moore

- Il numero di transistor in un circuito integrato (IC) raddoppia ogni 18 mesi
  - Un numero sempre maggiore di transistor sempre più piccoli ad ogni generazione di processori

Gordon E. Moore [1929 -]

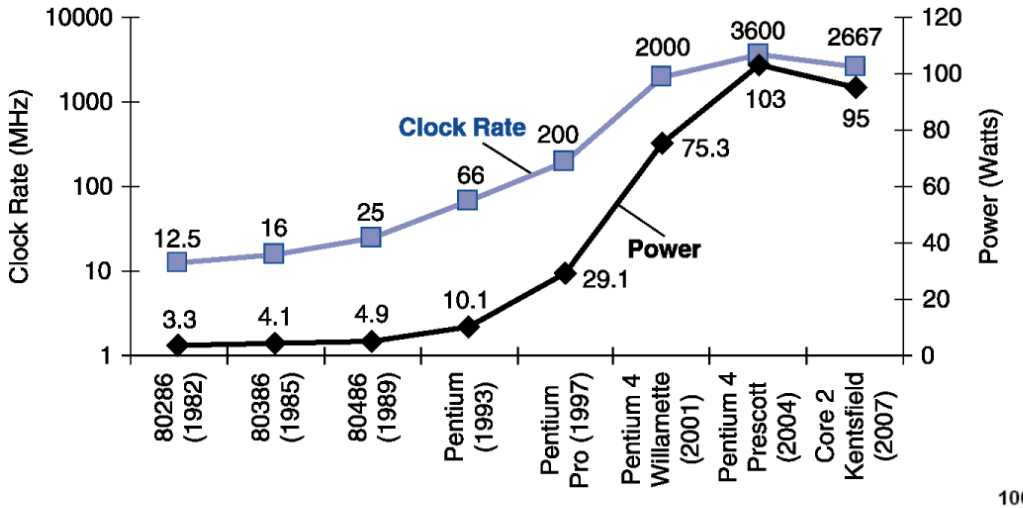
Fondatore di Intel (1968)



Miniaturizzazione del processo produttivo dei transistor

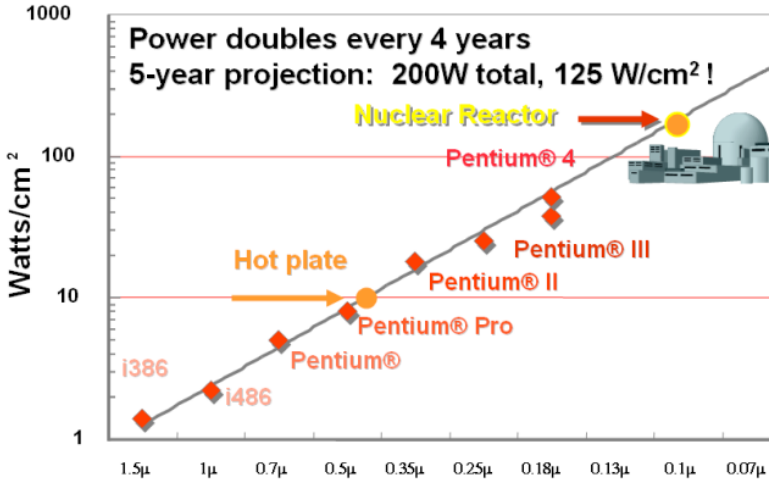
10	µm – 1971
6	µm – 1974
3	µm – 1977
1.5	µm – 1982
1	µm – 1985
800	nm – 1989
600	nm – 1994
350	nm – 1995
250	nm – 1997
180	nm – 1999
130	nm – 2001
90	nm – 2004
65	nm – 2006
45	nm – 2008
32	nm – 2010
22	nm – 2012
14	nm – 2014
10	nm – 2017
7	nm – ~2019
5	nm – ~2021

Il *Power Wall*



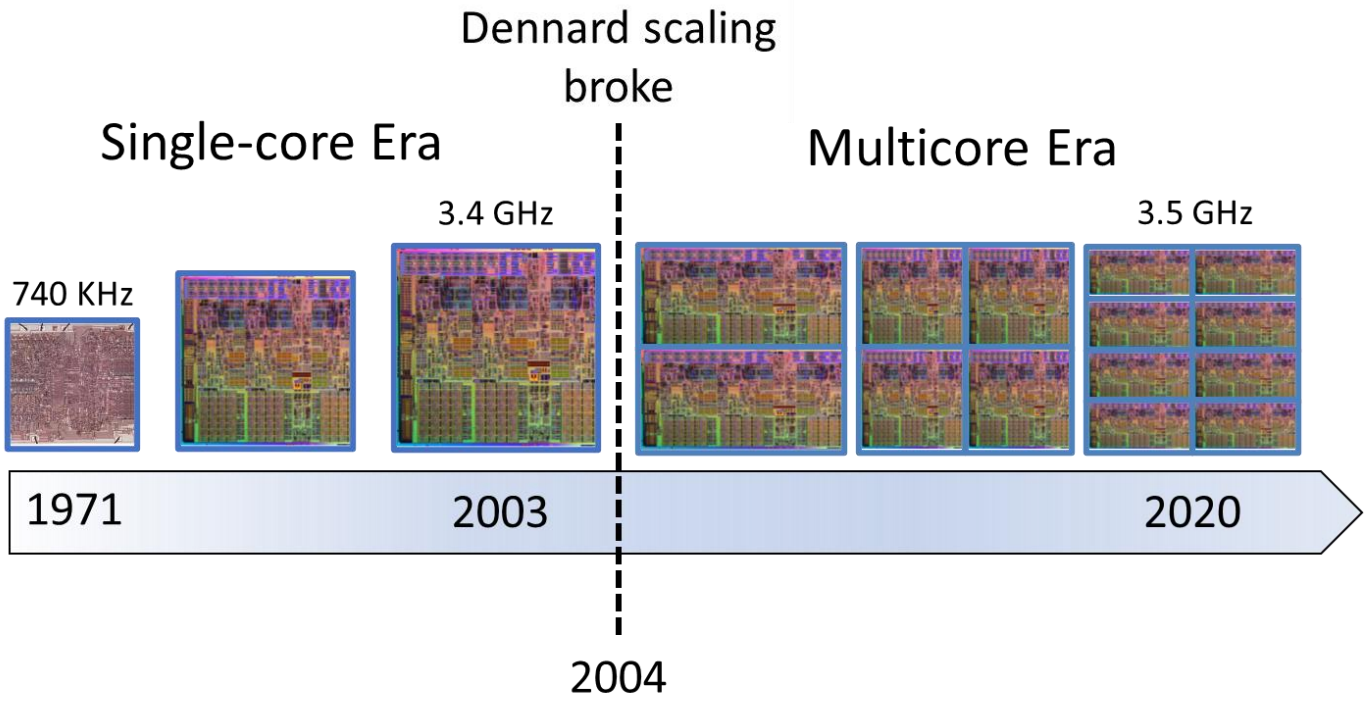
[Patterson & Hennessy]

- L'obiettivo dei chip designers alla fine degli anni novanta e nei primi 2000 era quello di riuscire ad ottenere frequenze sempre più alte
- *Sfruttando i transistor aggiuntivi per migliorare le pipelines di CPU single-core*



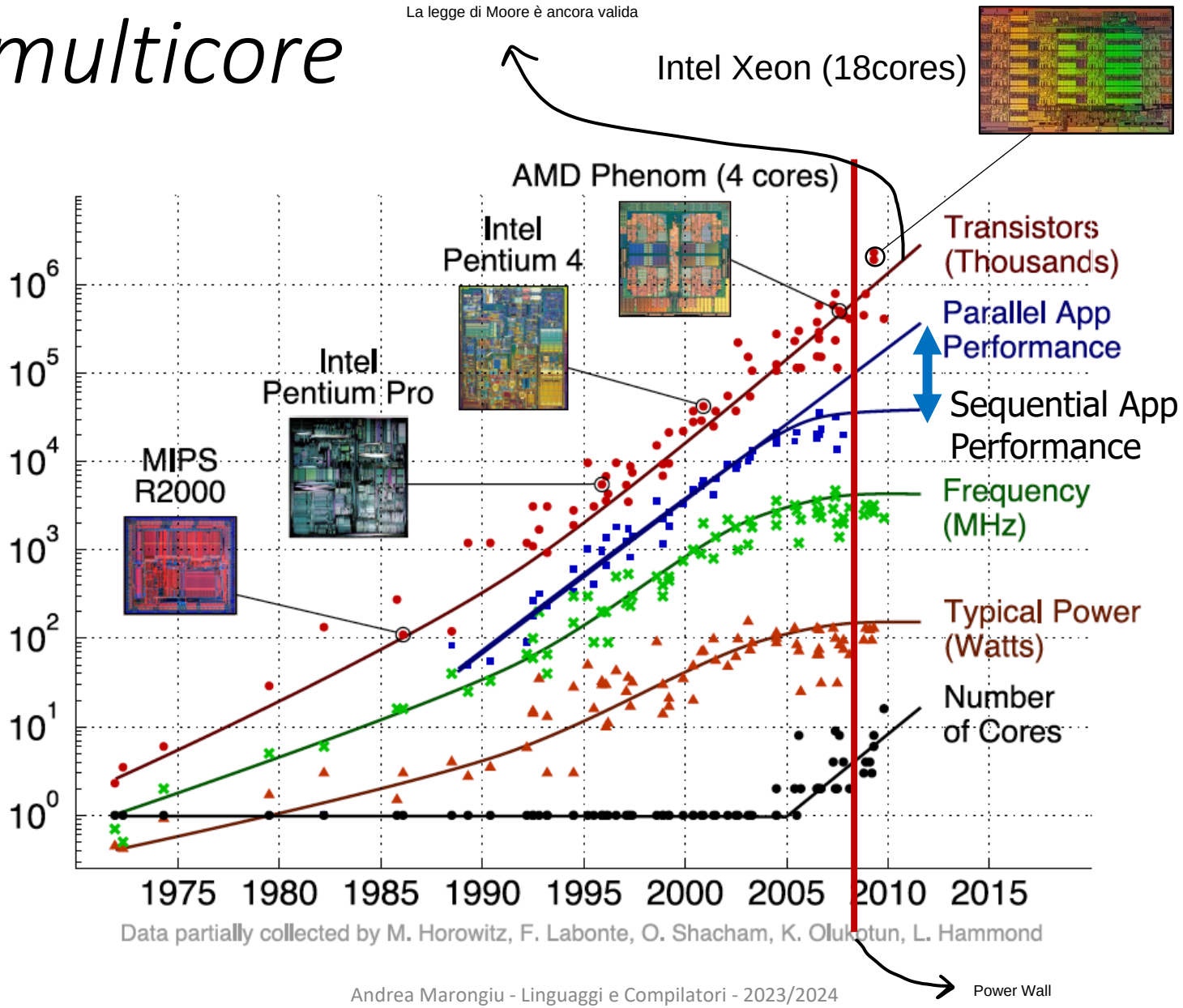
Sfortunatamente questo ha portato ad un punto dove la densità di potenza generava quantità di calore impossibili da dissipare con tecniche di raffreddamento economiche

I *multicore*



- Meglio sfruttare tanti cores con frequenze più basse che uno solo con frequenze sempre crescenti

I *multicore*


La legge di Moore è ancora valida
Intel Xeon (18cores)
AMD Phenom (4 cores)
Intel Pentium 4
Intel Pentium Pro
MIPS R2000
Transistors (Thousands)
Parallel App Performance
Sequential App Performance
Frequency (MHz)
Typical Power (Watts)
Number of Cores
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975 1980 1985 1990 1995 2000 2005 2010 2015
Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
Power Wall

I *multicore*

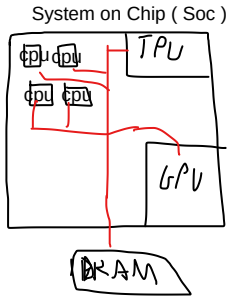
- La perdita di performance dovuta al mancato incremento di frequenza è recuperata dal parallelismo (numero di *cores*)
- Potenzialmente la performance dei chip continua a crescere ad ogni generazione
- Non è però sfruttabile senza un cambio di paradigma nella maniera in cui si scrive il software
- I compilatori rivestono un ruolo fondamentale in questo contesto

I compilatori per *multicore*

- Da un lato, per rendere la transizione alla programmazione parallela meno traumatica, i compilatori *auto-parallelizzanti* diventano una tematica di ricerca (e non solo) prioritaria

- Dall'altro, una pletora di *nuovi parallel programming models* vede la luce
  - Espongono al programmatore una semplice interfaccia verso il parallelismo
  - Il compilatore traduce i semplici costrutti dell'interfaccia in codice parallelo eseguibile

L’evoluzione dei compilatori



- La programmazione parallela e il parallelismo architetturale sono paradigmi ormai consolidati
- Questo significa che le architetture e i compilatori per queste architetture hanno smesso di evolvere?
- No, i sistemi di calcolo sono in continua evoluzione

L’**eterogeneità** architetturale è la coesistenza nel sistema di CPU general purpose e acceleratori di vario tipo (GPU, FPGA). Ciascuna di queste unità è **specializzata** in un compito che svolge molto più efficientemente delle altre unità

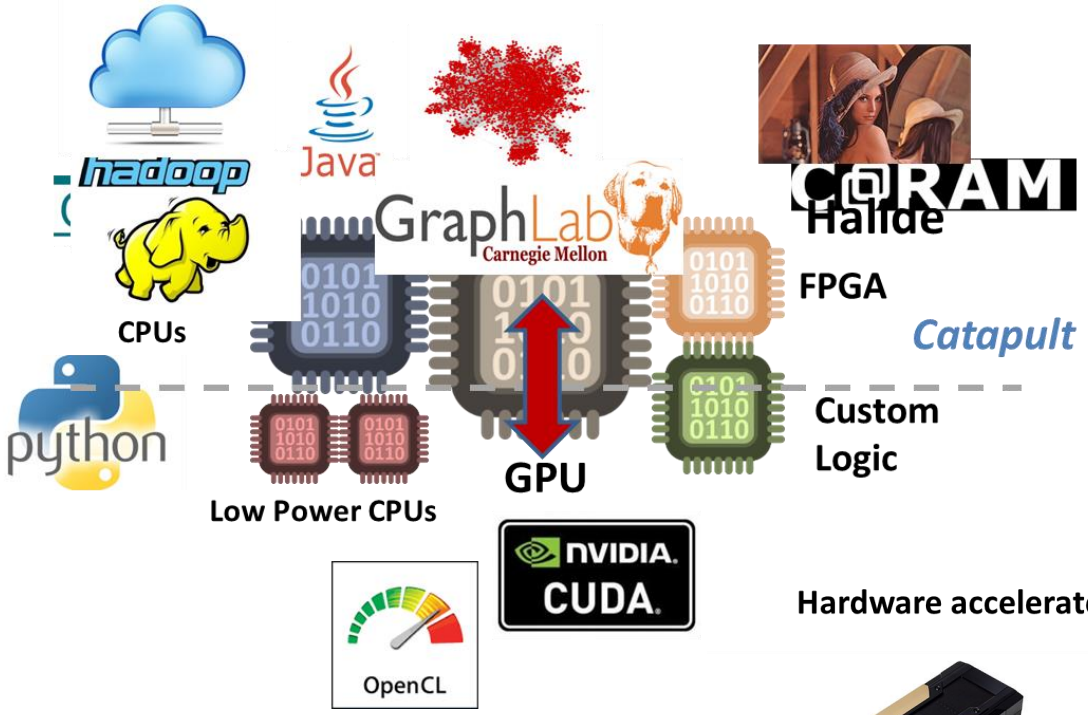
Hardware accelerators:

GPUs (Graphics Processing Units)

FPGAs (Field Programmable Gate Arrays)

TPUs (Tensor Processing Units)

Eterogeneità e Specializzazione



L'**eterogeneità** complica ulteriormente la scrittura di software. Questo richiede continue evoluzioni alla struttura interna di un compilatore e alle sue capacità di ottimizzazione

Hardware accelerators:

GPUs (Graphics Processing Units)

FPGAs (Field Programmable Gate Arrays)

TPUs (Tensor Processing Units)

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Ottimizzazione

Come i compilatori migliorano la performance?

- Ricordiamo alcune metriche:

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instruction\ Count \times CPI}{Frequency}$$

Numero istruzioni del programma

Cicli Per istruzione

$f = [Hz] \frac{\text{Cicli}}{\text{Sec}}$

- Frequency -> è scelta in fase di design, il programmatore non ha controllo.
- CPI -> Dipende dalle istruzioni che scegliamo (ex : sum,sottr -> costano poco) (ex : branch -> miss prediction load -> miss rate = costrano tanto). Ma dato che è un valore medio dipende dall' hardware
- Instruction Count -> Modificabile dal programmatore, cerco di eseguire meno istruzioni.

Come i compilatori migliorano la performance?

- Ricordiamo alcune metriche:

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instruction\ Count \times CPI}{Frequency}$$

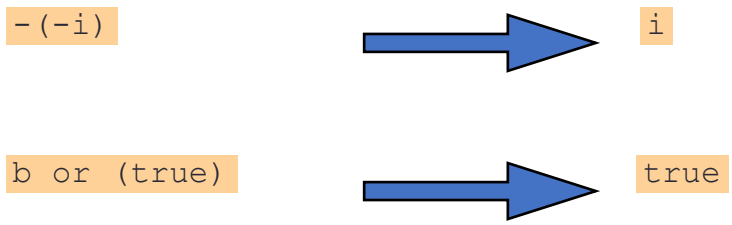
Parametro di programma (migliorabile dal compilatore)

Parametri architetturali (migliorabili solo modificando l'hardware)

- Minimizzando il numero di istruzioni
  - Operazioni aritmetiche, accessi in memoria
- Rimpiazzando operazioni costose con altre più semplici
  - Es., rimpiazzare una *moltiplicazione* (4 cicli) con uno *shift* (1 ciclo)

Esempi di ottimizzazione: **Algebraic Simplifications (AS)**

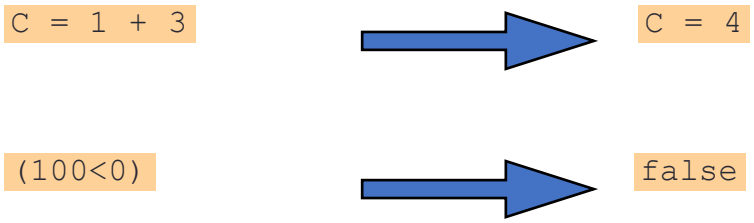
- Utilizza proprietà algebriche per semplificare le espressioni



Semplifica il codice per ottimizzazioni successive

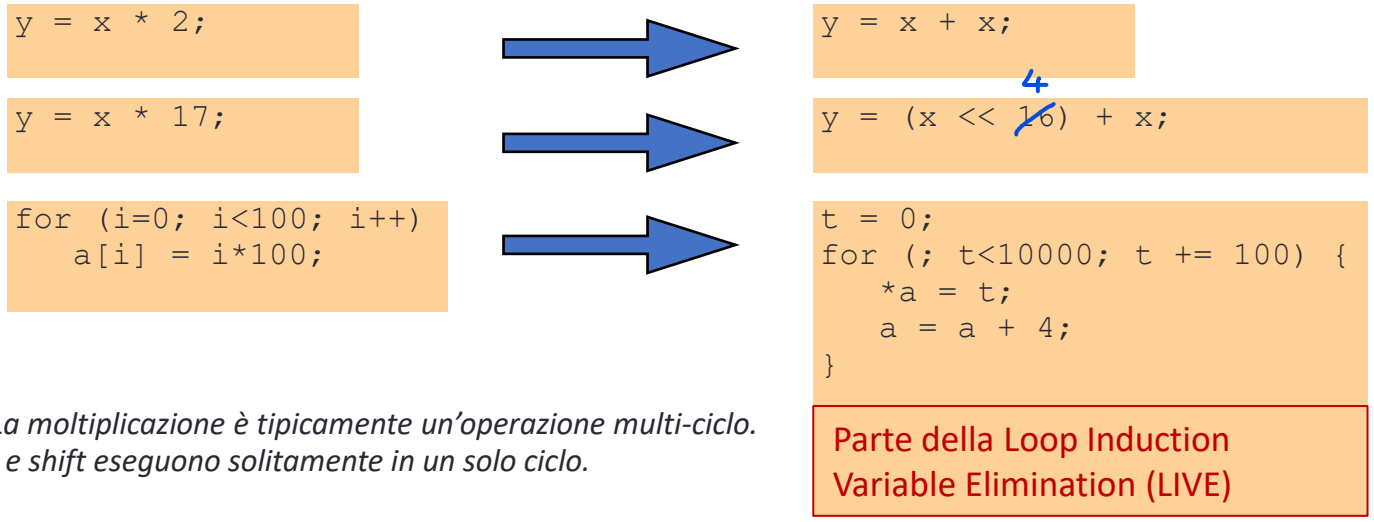
Esempi di ottimizzazione: **Constant Folding (CF)**

- Valuta ed espande le espressioni costanti a tempo di compilazione



Esempi di ottimizzazione: **Strength Reduction (SR)**

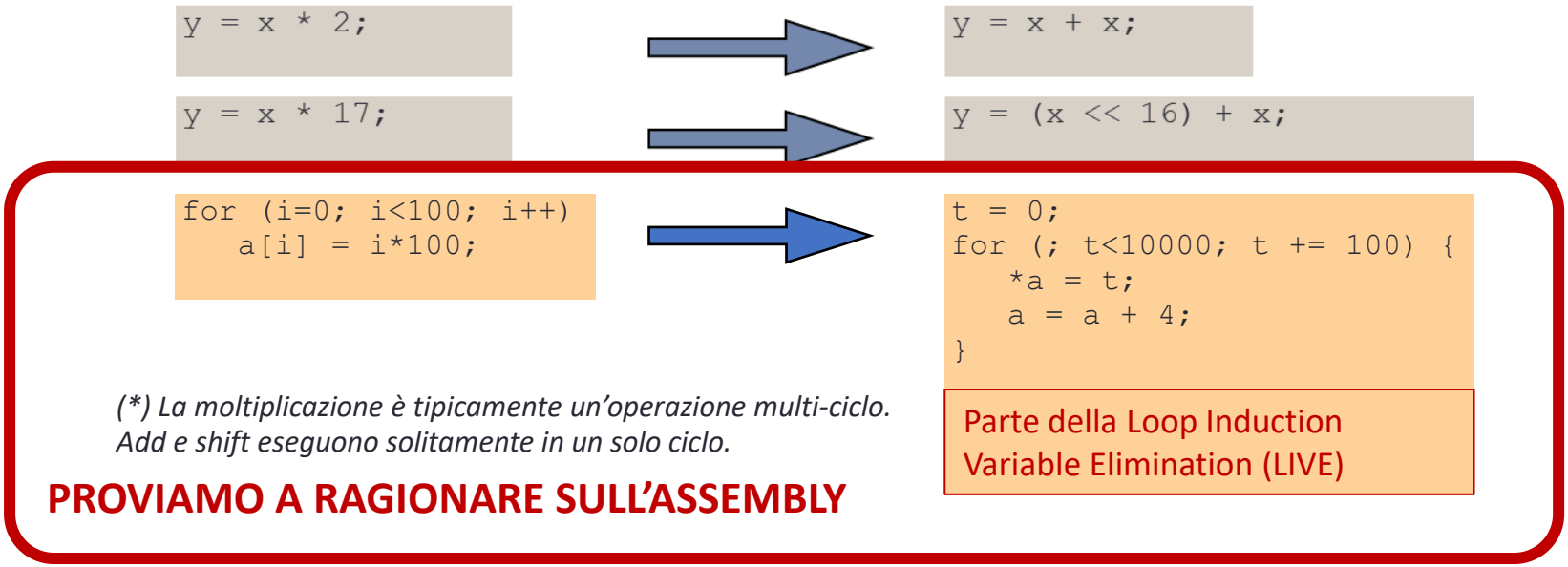
- Sostituisce operazioni costose con altre più semplici
- Es., MUL rimpiazzate da ADD/SHIFT (*)



(*) La moltiplicazione è tipicamente un'operazione multi-ciclo. Add e shift eseguono solitamente in un solo ciclo.

Esempi di ottimizzazione: **Strength Reduction (SR)**

- Sostituisce operazioni costose con altre più semplici
- Es., MUL rimpiazzate da ADD/SHIFT (*)



Esempi di ottimizzazione: **Strength Reduction (SR)**

- Assumiamo che l'indirizzo dell'array `a` si trovi sul registro `a0`

Int a*;

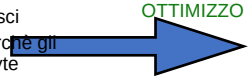
```
for (i=0; i<100; i++)
   a[i] = i*100;
```

```
t = 0;
for (; t<10000; t += 100) {
   *a = t;
   a = a + 4;
}
```

```
li     s0, 0  // i = 0
li     s1, 100
LOOP:
bge    s0, s1, EXIT   Se i > 100 Esci
slli   s2, s0, 2      Moltiplico per 4 , perchè gli interi occupano 4 Byte
add    s2, s2, a0     Ho aggiunto l'offset all'array
mul    s3, s0, 100
sw     s3, 0(s2)
addi   s0, s0, 1      Incremento i
jal    zero, LOOP
EXIT:
```

1. Ho tolto la moltiplicazione
2. Ho ottimizzato l'incremento

OTTIMIZZO



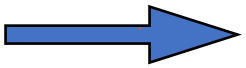
```
li     s0, 0  // t = 0
li     s1, 10000
LOOP:
bge    s0, s1, EXIT
sw     s0, 0(a0)
addi   a0, a0, 4
jal    zero, LOOP
EXIT:
```

CHE BENEFICI ABBIAMO OTTENUTO?

Esempi di ottimizzazione: **Common Subexpression Elimination (CSE)**

- Elimina calcoli ridondanti di una stessa espressione usata in più istruzioni (*statements*).

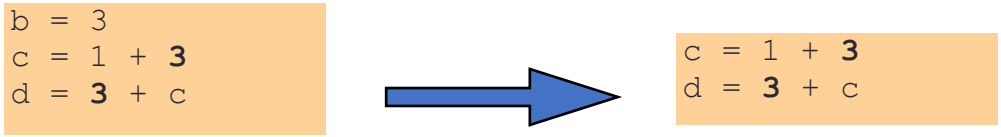
```
y = b * c + 4
z = b * c - 1
```



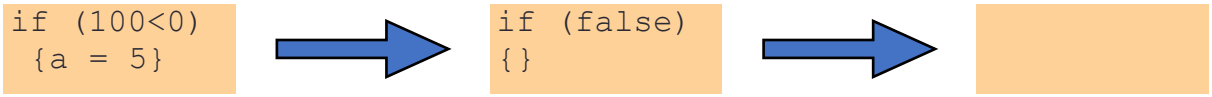
```
x = b * c
y = x + 4
z = x - 1
```

Esempi di ottimizzazione: **Dead Code Elimination (DCE)**

- Rimuove codice non necessario
  - es., variabili assegnate ma mai lette (usate)



  - es., codice irraggiungibile



Esempi di ottimizzazione: **Copy Propagation**

- Per uno statement *x = y*
- Sostituisce gli usi futuri di *x* con *y*
  - se *x* e *y* non sono cambiati nel frattempo

```
x = y
c = 1 + x
d = y + c
```

```
x = y
c = 1 + y
d = y + c
```

Esempi di ottimizzazione: **Copy Propagation**

- Per uno statement *x = y*
- Sostituisce gli usi futuri di *x* con *y*
  - se *x* e *y* non sono cambiati nel frattempo

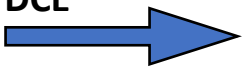
```
x = y
c = 1 + x
d = y + c
```

```
x = y
c = 1 + y
d = y + c
```

  - Spesso propedeutico alla DCE

```
x = y
c = 1 + y
d = y + c
```

DCE



```
c = 1 + y
d = y + c
```

Esempi di ottimizzazione: **Constant Propagation (CP)**

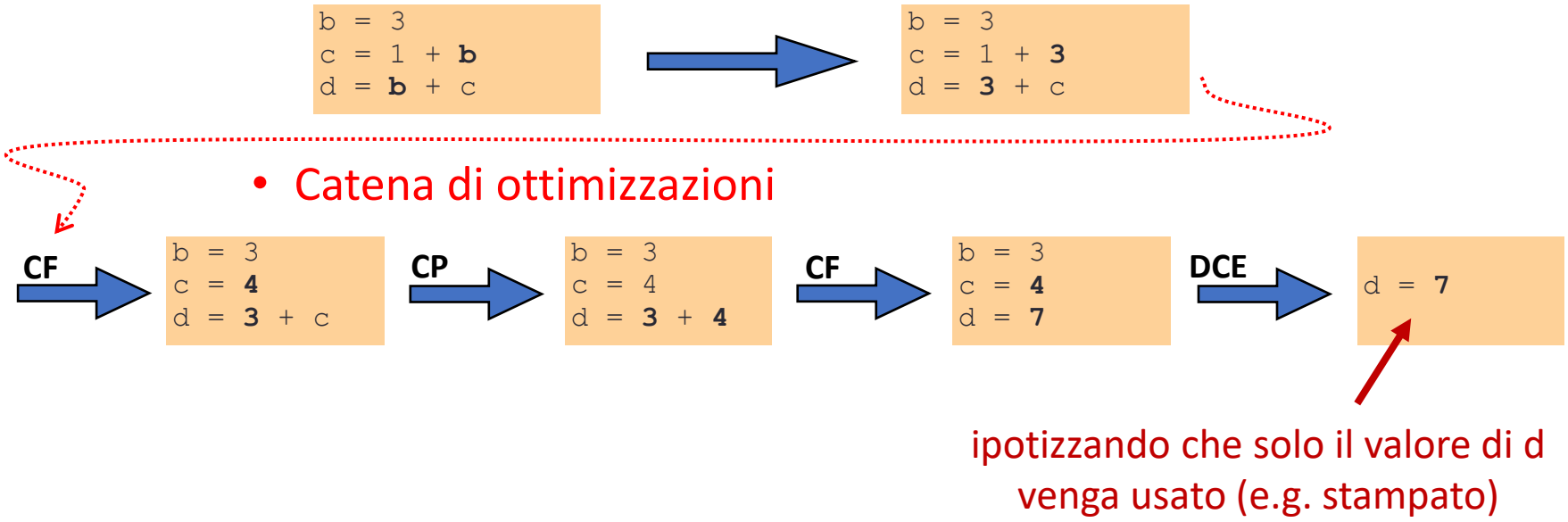
- Per le variabili con valore costante (es., *b = 3*)
  - Sostituisce gli usi futuri di *b* con la costante
    - Se *b* non è cambiato nel frattempo

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

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

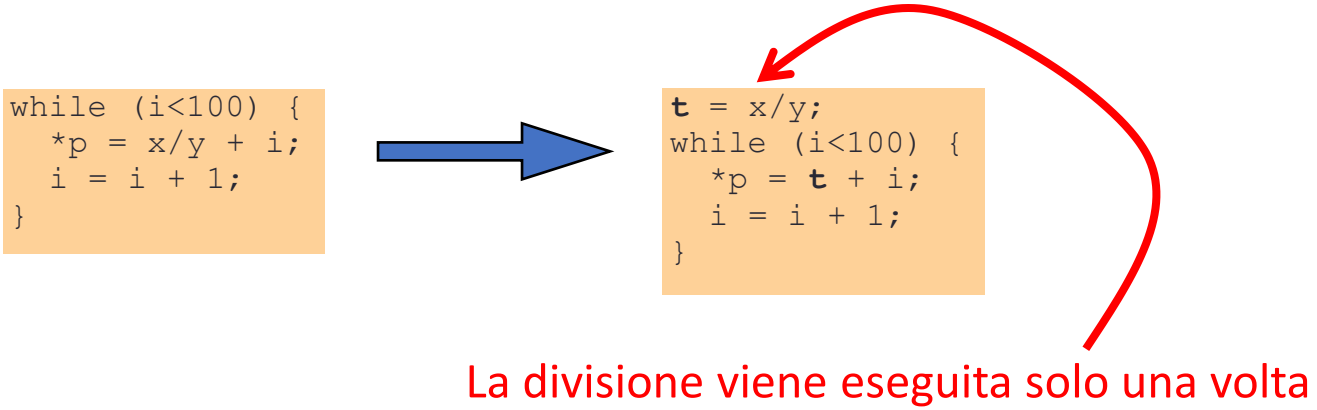
Esempi di ottimizzazione: **Constant Propagation (CP) + altri**

- 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



Esempi di ottimizzazione: **Loop Invariant Code Motion (LICM)**

- Sposta le istruzioni indipendenti dal *loop* fuori dal *loop* stesso
  - Viene anche chiamata *Code Hoisting*
  - Evita i calcoli ridondanti



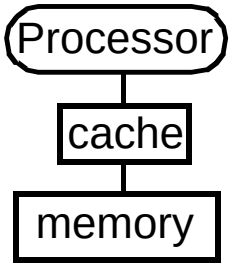
Ottimizzazioni sui loop

- La maggior parte dei programmi spende il grosso del suo tempo di esecuzioni dentro uno o più loop
  - Ottimizzare il loop ha quindi un grande impatto sulla performance dell'intero programma
- Le ottimizzazioni sui loop sono spesso propedeutiche a ottimizzazioni *machine-specific* (effettuate nel *backend*)
  - Register allocation
  - Instruction-level parallelism
  - Data parallelism (multi-core, SIMD) //tante CPU
  - Data-cache locality
- I loop sono in generale un target per le ottimizzazioni
  - Centrali nel parallelismo

Come i compilatori migliorano la performance?

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instruction\ Count\ \times CPI}{Frequency}$$



- Minimizzano le *cache miss*
  - Sia su istruzioni che su dati
- Sfruttano il parallelismo
  - Scheduling delle istruzioni nel singolo thread (ILP)
  - Esecuzione parallela su multipli threads
    - Single program, multiple data (SPMD)
    - Multiple program, multiple data (MPMD)

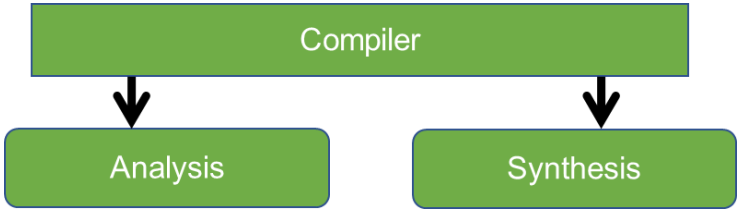
UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Anatomia di un compilatore

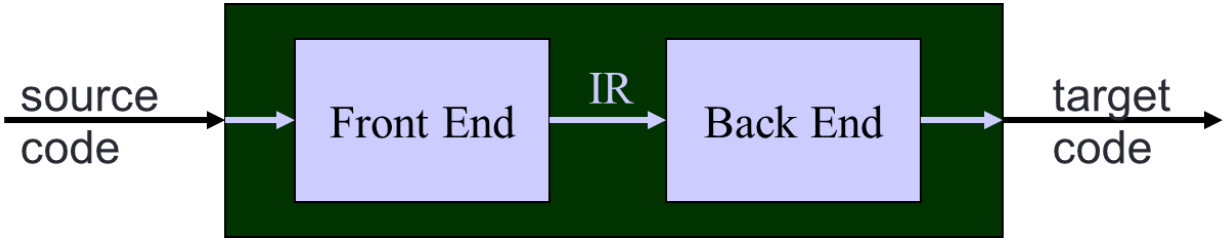
Anatomia di un compilatore

- Un compilatore deve svolgere almeno due compiti:

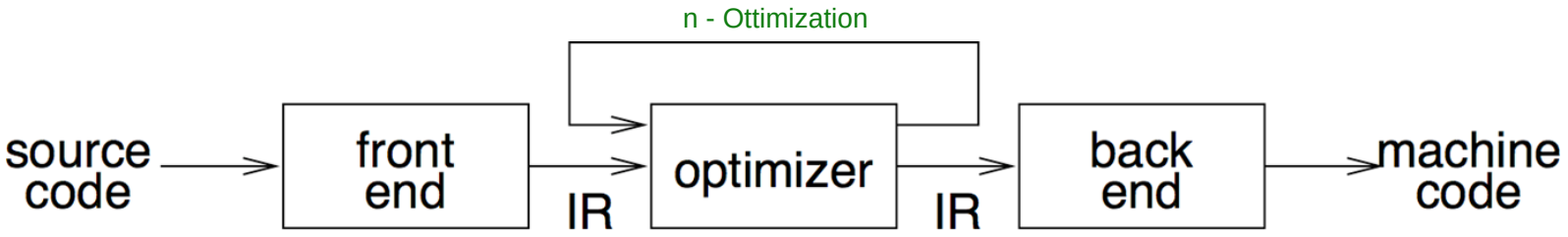


1. Analisi del codice sorgente (source) → Analizzo IR
2. Sintesi di un programma in linguaggio macchina (target)

- Opera su una Rappresentazione Intermedia (IR)

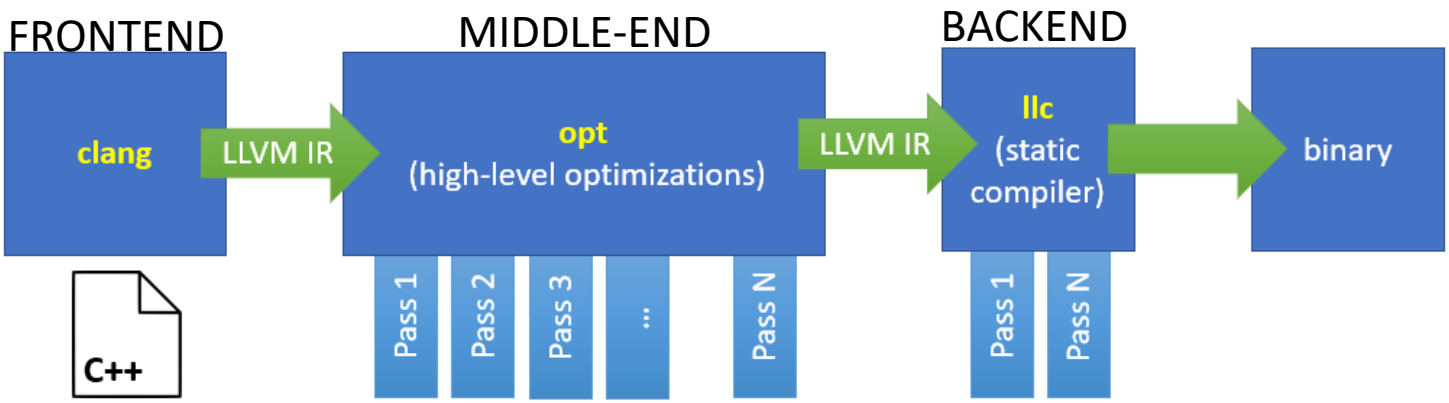


Rappresentazione Intermedia (IR)



- Il blocco **Front-end** produce la IR
  - Oggetto della prima parte del corso
- Il blocco **Middle-end** (optimizer) trasforma la IR in vari *passi* in una versione più efficiente
  - Oggetto di questa parte del corso
- Il blocco **Back-end** trasforma la IR nel codice *target*
  - Oggetto (brevemente) di questa parte del corso

Rappresentazione Intermedia (IR)



- L'ottimizzatore LLVM (opt)
  - È organizzato in una serie di passi di analisi/trasformazione.
  - Il *pass manager* stabilisce in che ordine applicare i passi per un dato obiettivo
- NOTA: Esistono passi di ottimizzazione anche nel *backend* (llc)

Flag di ottimizzazione

- I tipici flag che si possono passare al compilatore (cioè, al *pass manager*) per influenzare il numero e l'ordine dei passi di ottimizzazione sono:
  - **-g**
    - Solo per debugging, nessuna ottimizzazione.
  - **-O0**
    - Nessuna ottimizzazione
  - **-O1**
    - Esegue ottimizzazioni che non impiegano molto tempo
    - CP, CF, CSE, DCE, LICM, inlining…
  - **-O2**
    - Impiega molto tempo
    - Abilita passi di ottimizzazione più aggressivi

    Default
  - **-O3**
    - Esegue i passi in un ordine che sfrutta compromessi tra velocità e spazio occupato (sia del compilato che del processo di compilazione): loop unrolling, inlining spinto, …

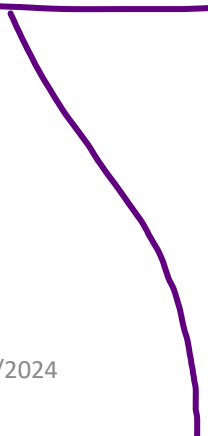
    Bisogna che il programmatore, ci faccia attenzione.
    Perchè altrimenti potresti non avere il risultato previsto.
  - **-Os**
    - Ottimizza per dimensione del compilato

    Fa eseguibili più piccoli.

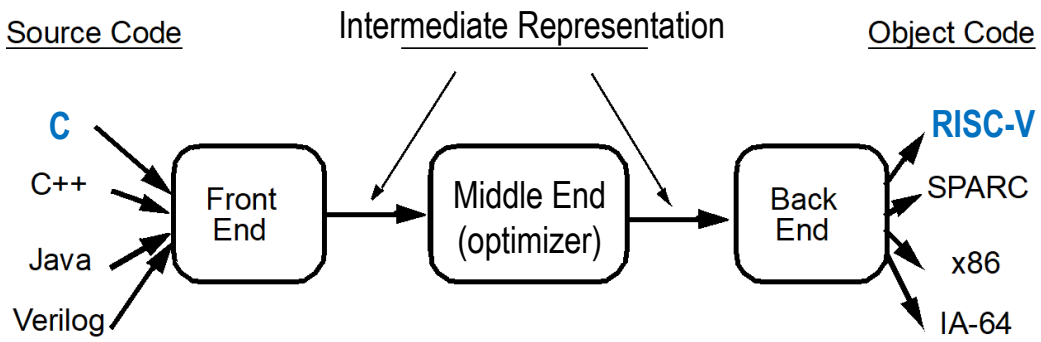
Per esempio in O2 esegue inlining di funzioni. Mentre qua no.
Inlining -> dove c'è la chiamata, inserisco la funzione così non c'è overhead per la call.

Perché usare una IR?

1. Principio di Ingegneria del Software
   - Spezza il compilatore in parti più gestibili
2. Semplifica il *retargeting* ad un nuovo ISA
   - Isola il Back-end dal Front-end
3. Semplifica il supporto a molti linguaggi
   - Diversi linguaggi condividono Middle- e Back-end
4. Abilita ottimizzazioni *machine-independent*
   - Tecniche generali, multipli *passi*



Anatomia di un compilatore

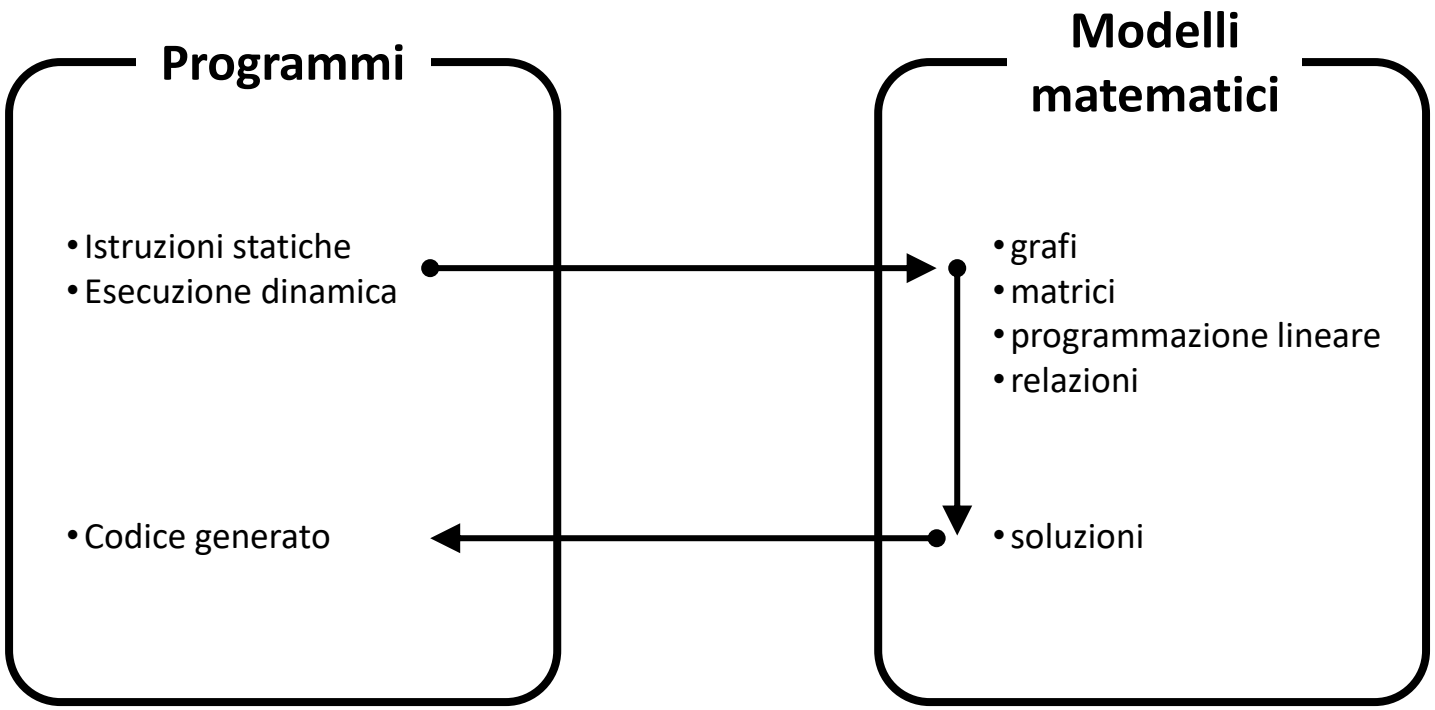


- Il blocco Middle-end (ottimizzatore) opera sulla stessa IR prodotta da ogni Front-end e ricevuta in input da ogni back-end
- Per supportare un nuovo linguaggio occorre solo scrivere un nuovo Front-end
- Per supportare un nuovo target (ISA) occorre solo scrivere un nuovo Back-end

Ingredienti dell'Ottimizzazione

- **Formulare un problema di ottimizzazione**
  - Identificare opportunità di ottimizzazione
    - Applicabili a molti programmi
    - Che impattino su parti significative del programma (loop/ricorsione)
    - Sufficientemente efficienti
- **Rappresentazione**
  - Deve astrarre i dettagli rilevanti per l'ottimizzazione

Ingredienti dell'Ottimizzazione



Ingredienti dell'Ottimizzazione

- **Formulare un problema di ottimizzazione**
  - Identificare opportunità di ottimizzazione
    - Applicabili a molti programmi
    - Che impattino su parti significative del programma (loop/ricorsione)
    - Sufficientemente efficienti
- **Rappresentazione**
  - Deve astrarre i dettagli rilevanti per l'ottimizzazione
- **Analisi**
  - Capire se è sicuro e desiderabile applicare una trasformazione
- **Trasformazione del codice**
- **Validazione sperimentale** (e si ripete il processo)

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

1. Introduzione al corso

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