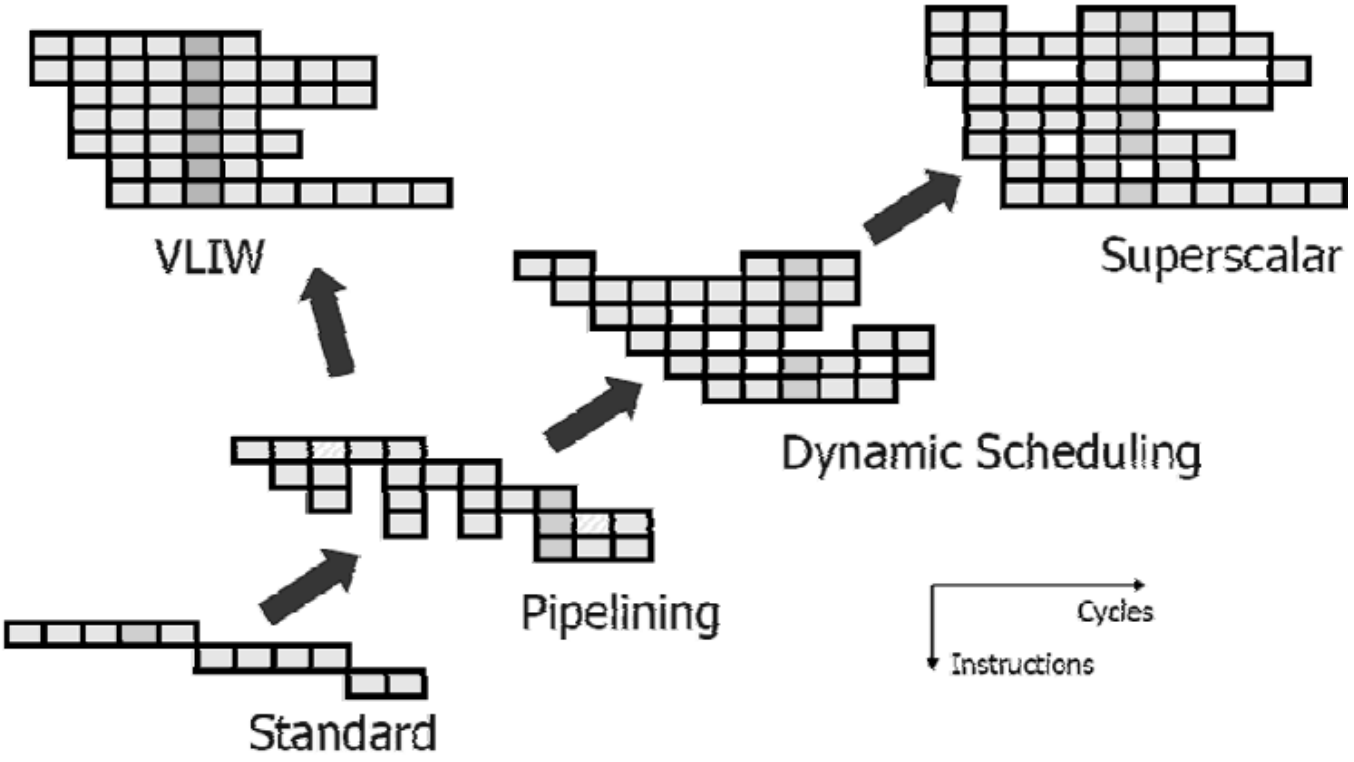
Instruction Level Parallelism & architetture VLIW



Davide Patti
davide.patti@dieei.unict.it

Topics

- **Instruction Level Parallelism (ILP)**
- **Superscalar vs VLIW**
- **Unità funzionali**
- **Exposed Latencies**
- **VLIW encodings**
- **VLIW in embedded multimedia**
- **ILP compiling: If-conversion**
- **ILP compiling: Hyperblock formation**

Trends

- **Legge di Moore**: La densità dei transistor raddoppia ogni 18 mesi.
- Crescente complessità dei sistemi embedded: requisiti di performance, power, etc..

Come sfruttare questa “abbondanza di transistor” per ottenere vantaggi in termini di performance ?

Sequential Program semantic

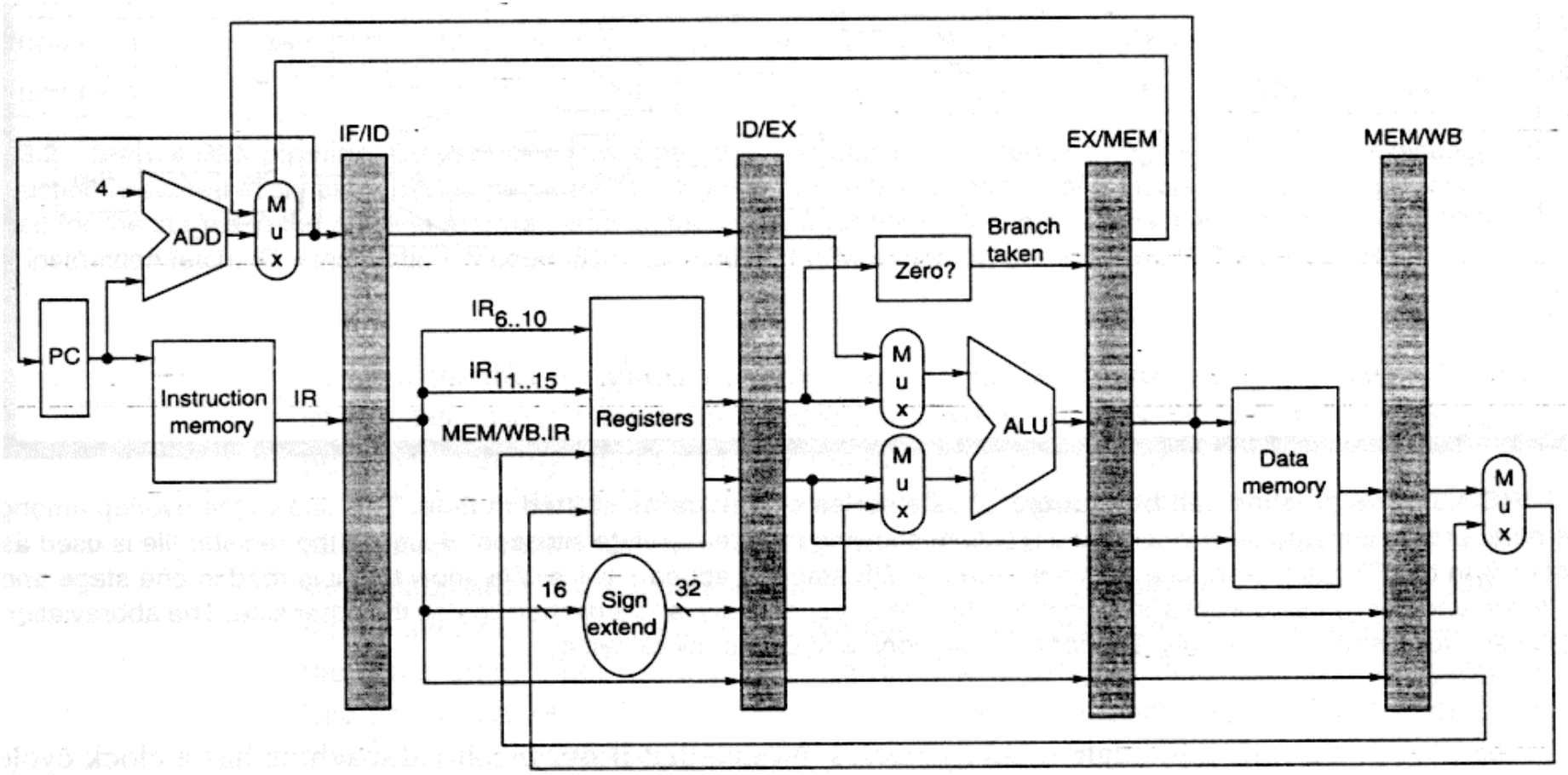
- Nessun parallelismo: Ogni istruzione può essere avviata solo dopo che la precedente è stata completata.
- Semantica di esecuzione sequenziale: visione classica del programmatore nell'immaginare l'esecuzione del proprio codice
- Dato che la latenza delle istruzioni e' > 1, e' praticamente impossibile avviare una nuova istruzione ad ogni ciclo di clock

ILP

- **Instruction Level Parallelism (ILP)**: L'idea è quella di aggiungere ulteriori risorse hardware all'interno del microprocessore al fine di permettere l'esecuzione simultanea di più istruzioni.
- Risorse hardware:
  - registri, unità funzionali (ALU, shifter, unità per float)
- Esempio classico: **pipelining** mediante aggiunta dei registri di pipeline e dell'hardware di controllo

ILP classico: pipelining

- A meno di stalli e conflitti, ad ogni ciclo di clock è completata una nuova istruzione (CPI =1).



ILP: Limitazioni Pipelining

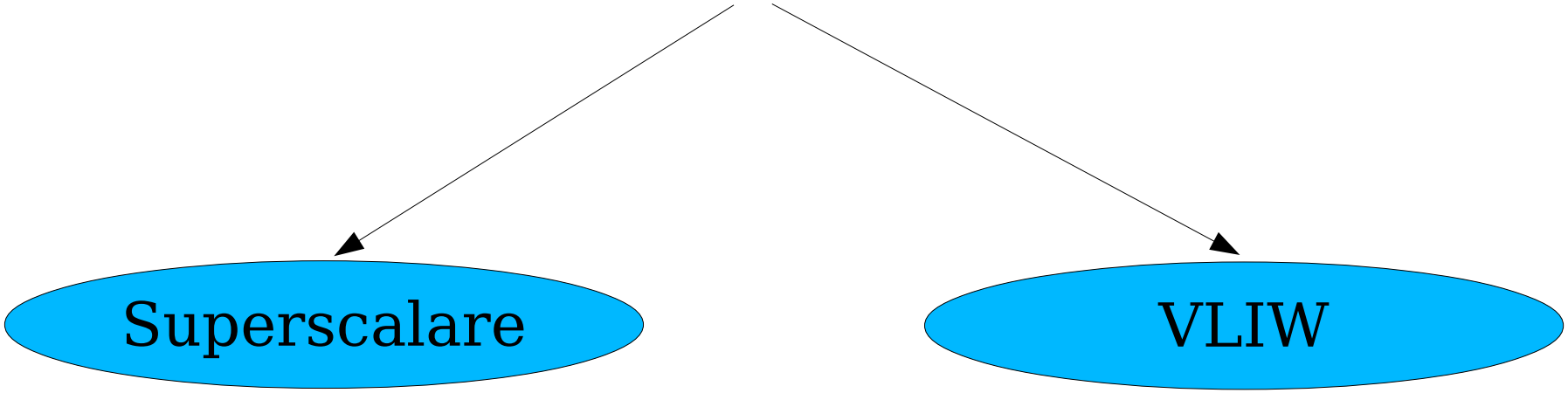
- **Stalli** dovuti ad conflitti sui dati, strutturali e di controllo
- **In-order execution**: se una istruzioni mette in stallo la pipeline quelle successive non possono essere avviate. Esempio:

```
DIVD f6,f2,f4    ;grande latenza....
ADDD f0,f6,f8    ;deve aspettare f6
SUBD f12,f8,f14 ;la si potrebbe avviare!
```

ILP: Multiple Issue

Idea:
Avviare più istruzioni in uno stesso ciclo di clock.

Bene, ma...
chi decide quali istruzioni possono essere avviate in un determinato ciclo di clock ?



NB: Entrambi i due approcci consentono un CPI<1
ossia “piu' istruzioni per ciclo”

Misurare l'ILP: CPI o IPC ?

- In una architettura classica, un CPI=1 è spesso definito come un *“Holy Grail ”*:

CPI = $\mathbf{CPI_{ideale}}$ + Cicli di stallo per istruzione

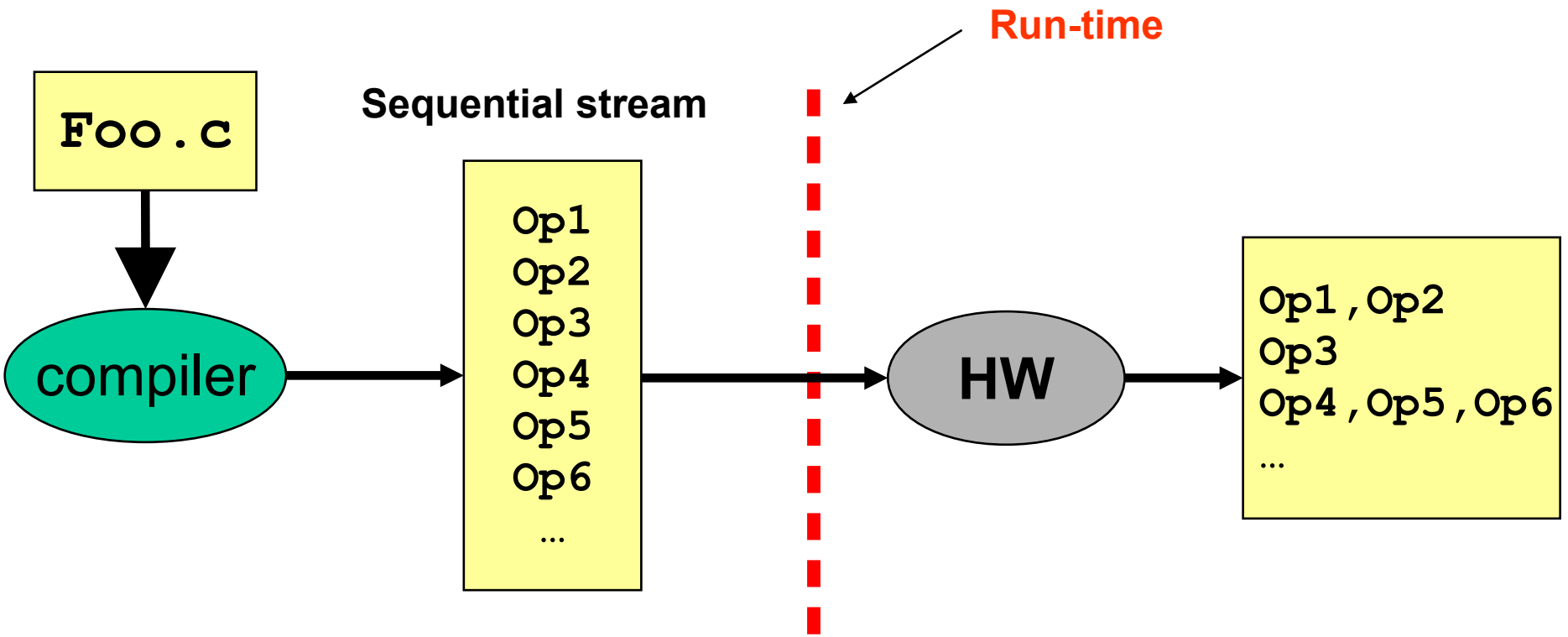
Domanda:

si può scendere oltre il limite teorico di un CPI = 1 ?

- Definiamo **IPC (Instructions per Cycle)** come: IPC = 1/CPI
- Se CPI >=1 , allora IPC <=1

ILP: Approccio Superscalare

- Ricerca dell'ILP nascosta a livello di Instruction-Set
- ILP scoperto *dinamicamente* dall'hardware di controllo durante l'esecuzione



•Super (oltre) scalare (unidimensionale), ossia più di una istruzione alla volta.

•Aggiunta di uno **schedulatore hardware** di controllo che individua, durante l'esecuzione, quali istruzioni possono essere avviate contemporaneamente.

•Trasparenza: dal punto di vista del programmatore /compilatore viene mantenuta l'illusione della esecuzione sequenziale.

•**Out-of-order execution:** Se gli operandi sono pronti, si può avviare una istruzione anche prima del previsto.

•NOTA BENE: Architetture RISC o CISC possono avere una “implementazione” superscalare, pur mantenendo immutato il set di istruzioni.

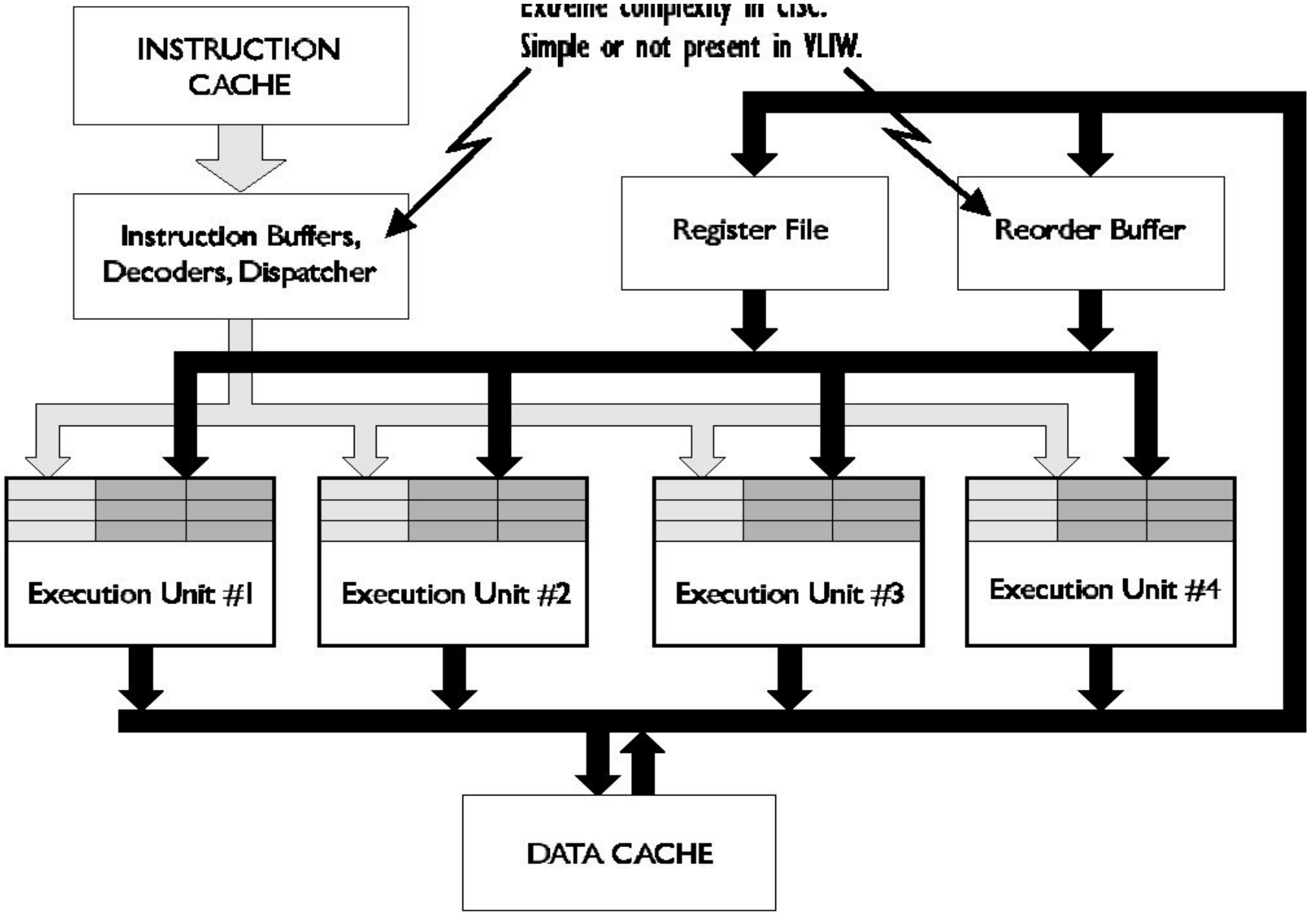
ILP: approccio Superscalare

- Compatibilità binaria con il codice preesistente

- Trasparenza dal punto di vista del compilatore e del programmatore assembly che continua a “vedere” codice sequenziale.

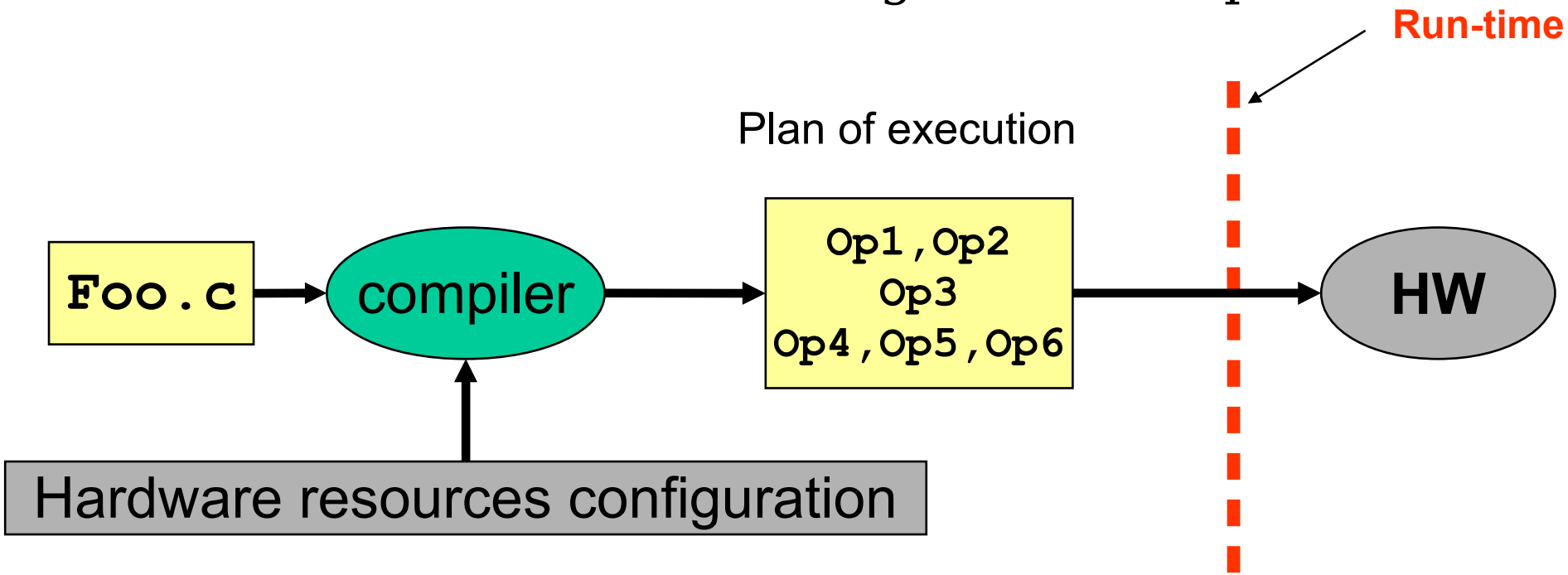
CONTRO

- Complessità dell'hardware di controllo
- Ciclo di clock più lungo
- Difficoltà nella modifica e il testing dello schedulatore


Extreme complexity in CISC.
Simple or not present in VLIW.
INSTRUCTION CACHE
Instruction Buffers, Decoders, Dispatcher
Register File
Reorder Buffer
Execution Unit #1
Execution Unit #2
Execution Unit #3
Execution Unit #4
DATA CACHE

ILP: Approccio VLIW

- Risorse hardware “architetturalmente visibili” al compilatore
- Il compilatore crea una sequenza di **Very Long Instructions** che definisce un **Plan Of Execution**
- L’hardware si limita ad eseguire il POE prestabilito



VLIW: terminologia

- Una **Very Long Instruction** è composta da un insieme di slot
- Ogni slot può contenere una operazione (di solito una normale istruzione RISC a 32 bit) oppure puo' essere vuoto (NOP)
- **Issue width**: numero di slot di ogni Very Long Instruction
- Si usa il termine “operazione” per evitare ambiguità. Quindi “istruzione” in ambito VLIW indica tutta la Very Long Instruction.
- Es: issue width=5, istruzione di 32 x 5 = 160bit

Esempio

Codice RISC classico :

```
lw r1, vet(r4)
lw r2, vet(r5)
add r3, r4, r5
divf f2, f4, f5
sub r6,r8,r9
sw vet(r7), r9
beqz r9, loop
```

Esempio

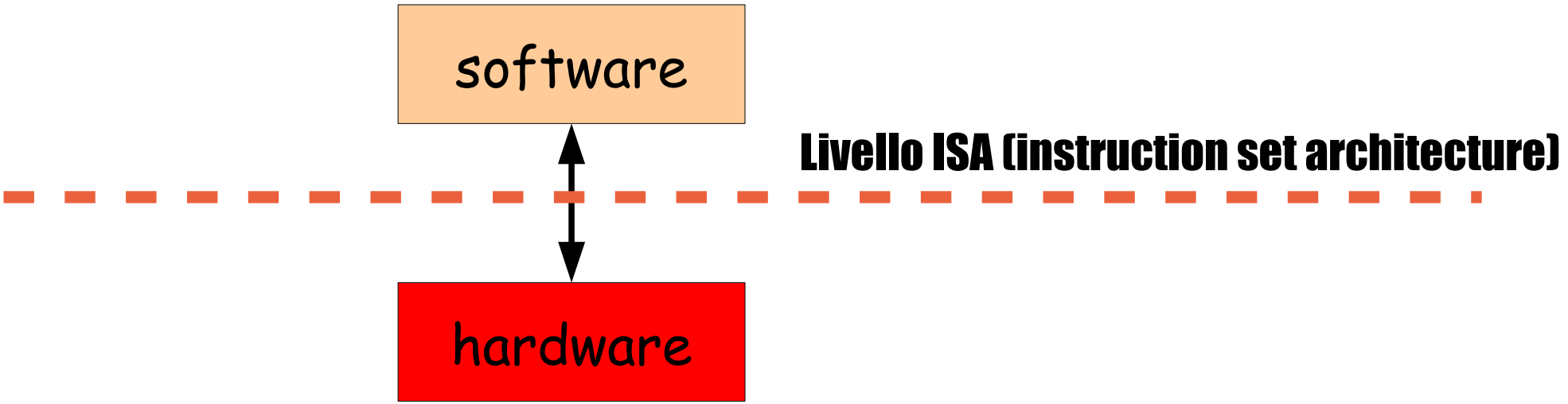
- Supponiamo una issue width=5 slot, che ci siano 2 unità funzionali per eseguire load/store, 2 per eseguire operazioni floating point, ed una per le operazioni intere.
- NB: alcuni slot rimangono vuoti (NOP = no operation)

lw r1,vet(r4)	**lw r2,vet(r5)**	**divf f2,f4,f5**	NOP	**Add r3,r4,r5**
sw vet(r7),r9	NOP	NOP	NOP	**Sub r6,r8,r9**
NOP	NOP	NOP	NOP	**Beqz r9,loop**

VLIW vs Superscalare

- **Superscalare** : ILP ottenuto dinamicamente a runtime.
- **VLIW**: ILP ottenuto staticamente, in fase di compilazione.

La distinzione sta nel decidere dove porre la complessità:
- a livello software (compilatore) o
- a livello di hardware (hardware di controllo)



VLIW: Ruolo del compilatore

- Ha la responsabilità della corretta schedulazione delle operazioni all'interno delle long instruction
- In particolare, deve conoscere:
  - **Numero e il tipo di unità funzionali**: serve a stabilire quante operazioni di un certo tipo posso avviare in parallelo;
  - **Latenza delle operazioni**: il numero di cicli per i quali ogni unità rimane occupata eseguendo un certo tipo di istruzione.

VLIW: Unità funzionali

- Ogni unità funzionale è associata ad un insieme di operazioni che essa può eseguire. Es:
  - **floating point ALU**:operazioni in virgola mobile
  - **integer ALU** per le operazioni su interi, calcolo delle condizioni nei salti
  - **load/store unit**: calcolano gli indirizzi, gestiscono l'I/O registri/memoriab
  - **branch unit**: gestione delle diramazioni

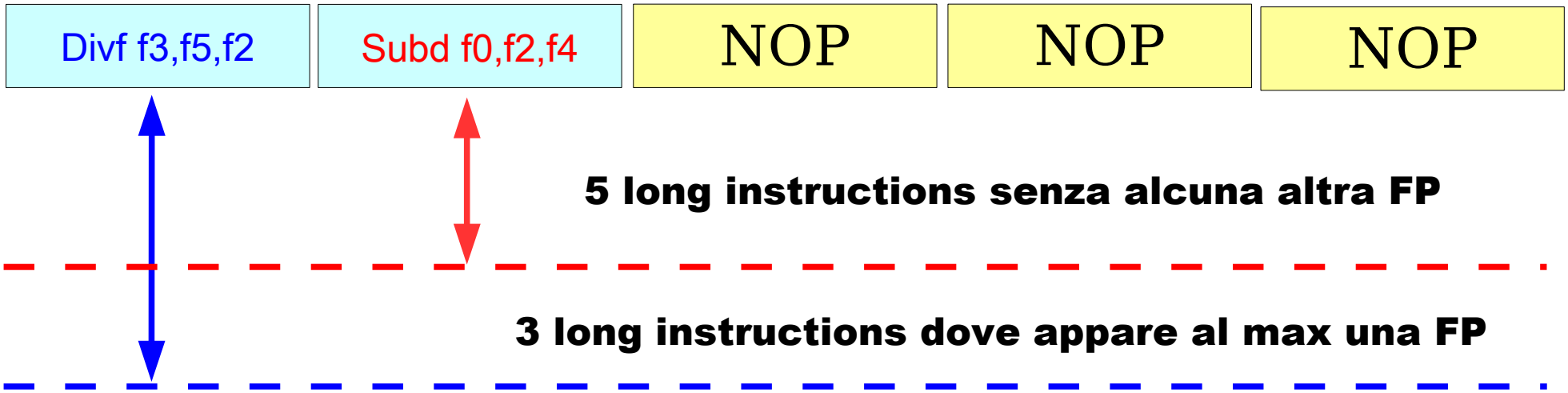
Esempio pratico: Se ci sono 2 unità per l'esecuzione di operazioni FP non posso schedulare 3 operazioni di tipo FP all'interno della stessa istruzione!

Divf f3,f5,f2	Subd f0,f2,f4	Addd f8,f6,f4	NOP	NOP

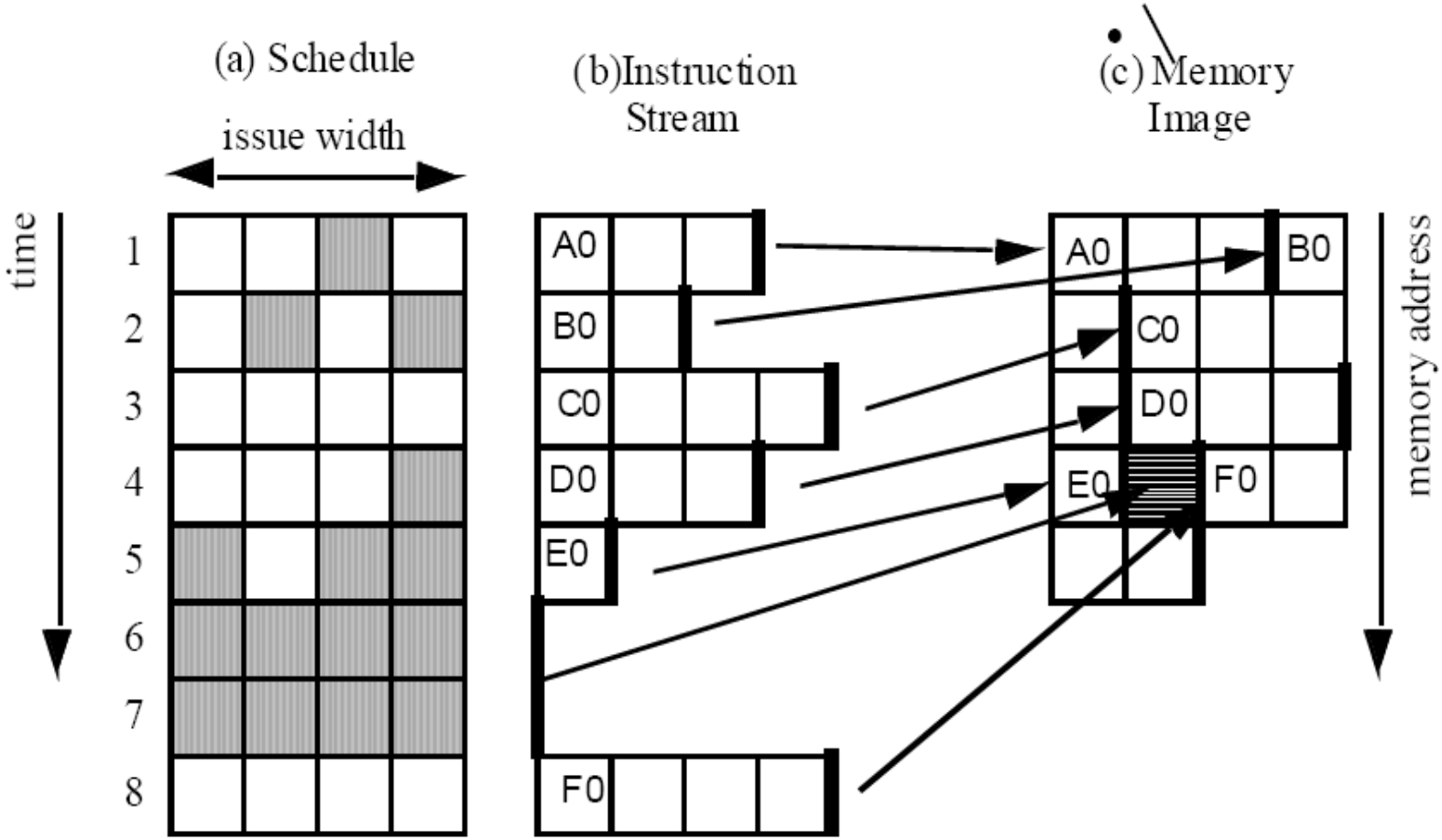
VLIW: Exposed latencies

- La latenza di ogni tipo di operazione è “esposta al compilatore” affinchè possa realizzare un “Plan of Execution” corretto.

  Esempio pratico: ci sono 2 unità per l'esecuzione di operazioni FP. La latenza di una DIVF è 8 cicli e quella di una SUBD è 5 cicli

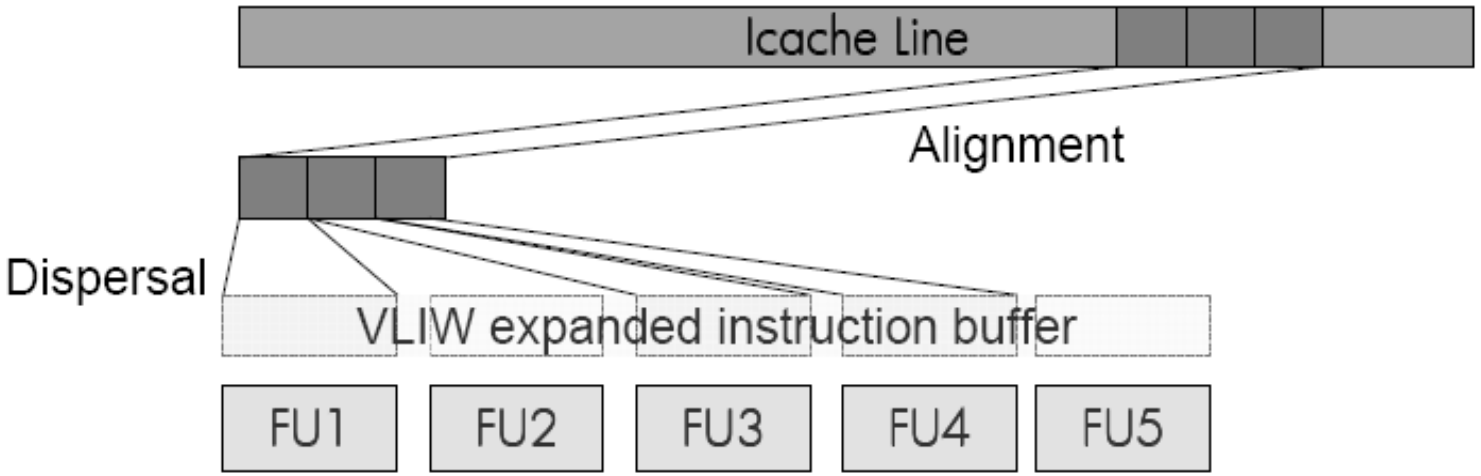


Horizontal and Vertical nops



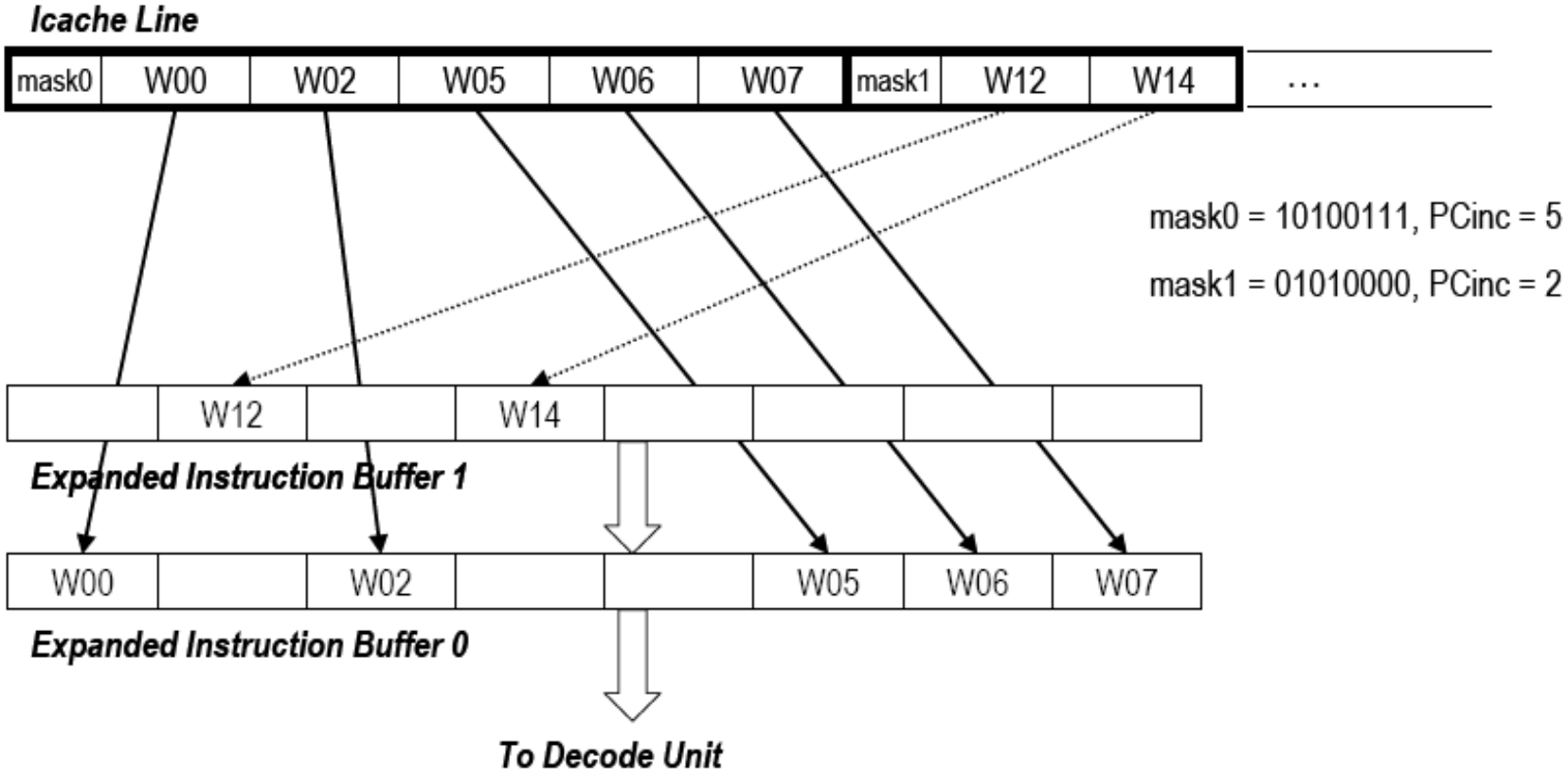
Very Long Instructions

- VLIW instructions are wide…
  - The compilers cannot always fill all the available issue slots
  - Variable length and compressed formats become necessary to avoid excessive code size and instruction cache pressure
- Fetching and issuing compressed long instructions
  - Need bandwidth from Icache to fetch the maximum number of operations supported
  - Need to *align* the fetch packet
  - Need to *disperse* the operations to the functional units

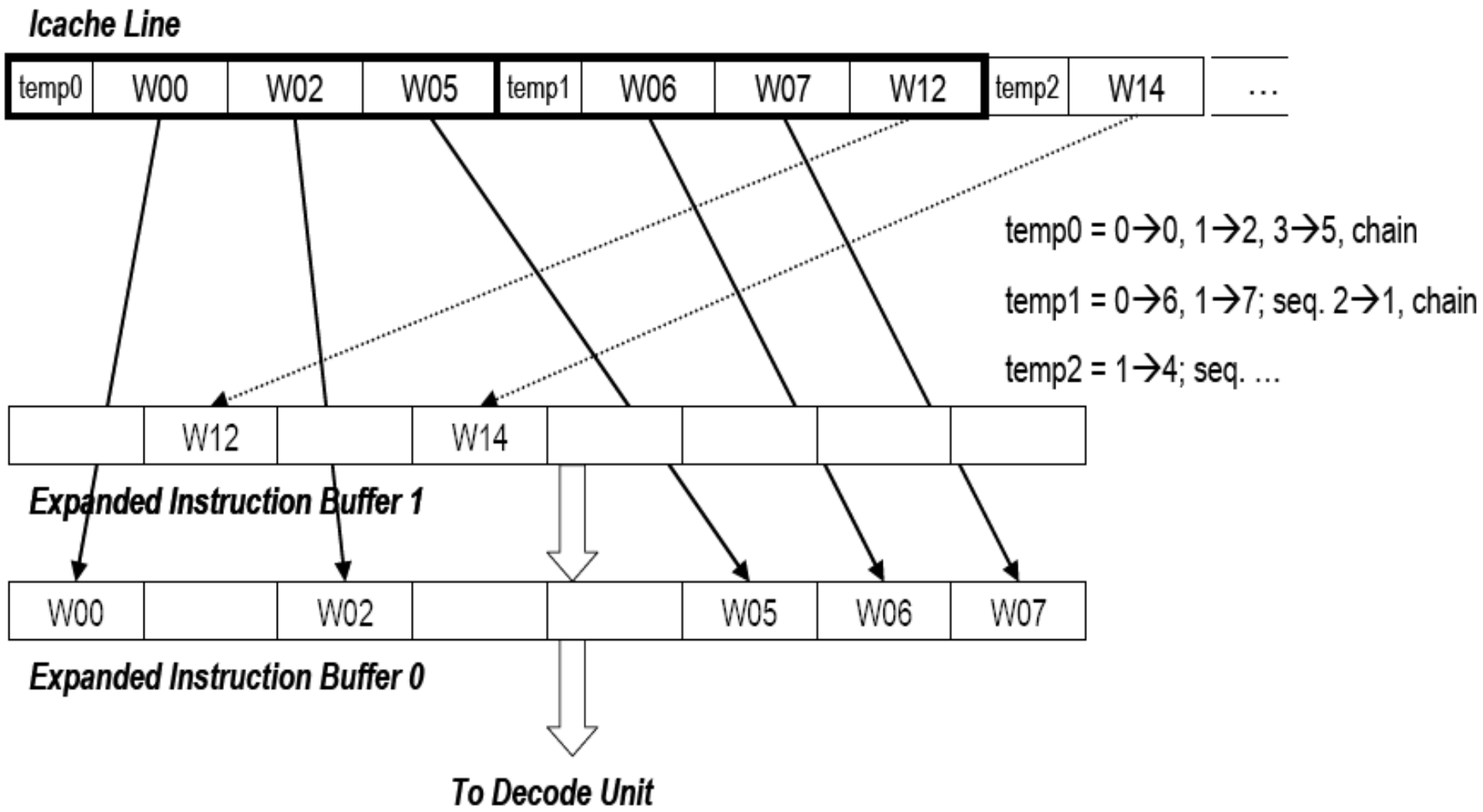


Mask-Based Encoding

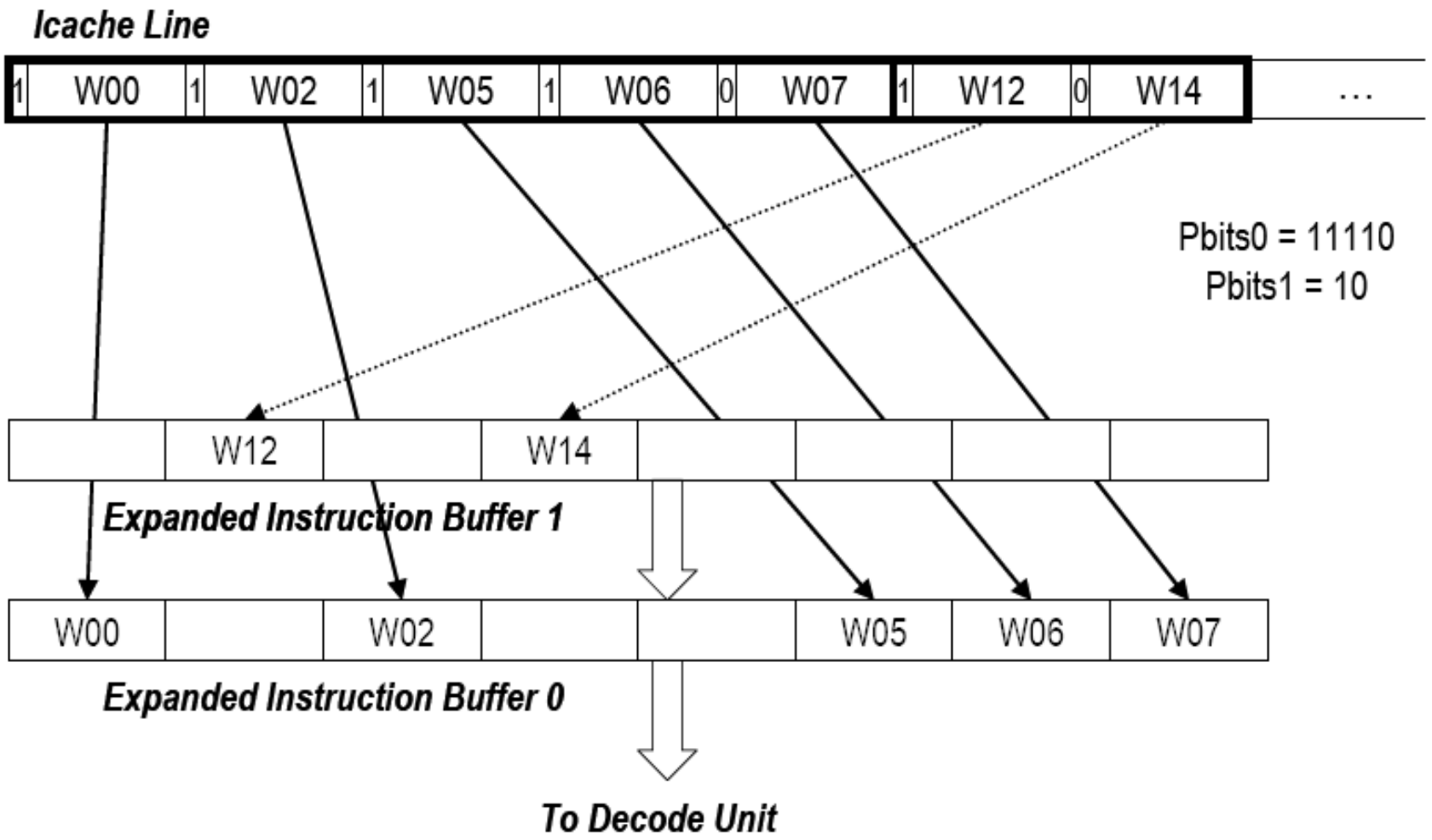
- Idea: add "mask-bits" (or templates) to VLIW instructions
  - Mask identifies "what goes where" in the instruction buffer
  - Mask identifies "next PC" address



Template-based Encoding



Distributed encoding with stop-bits

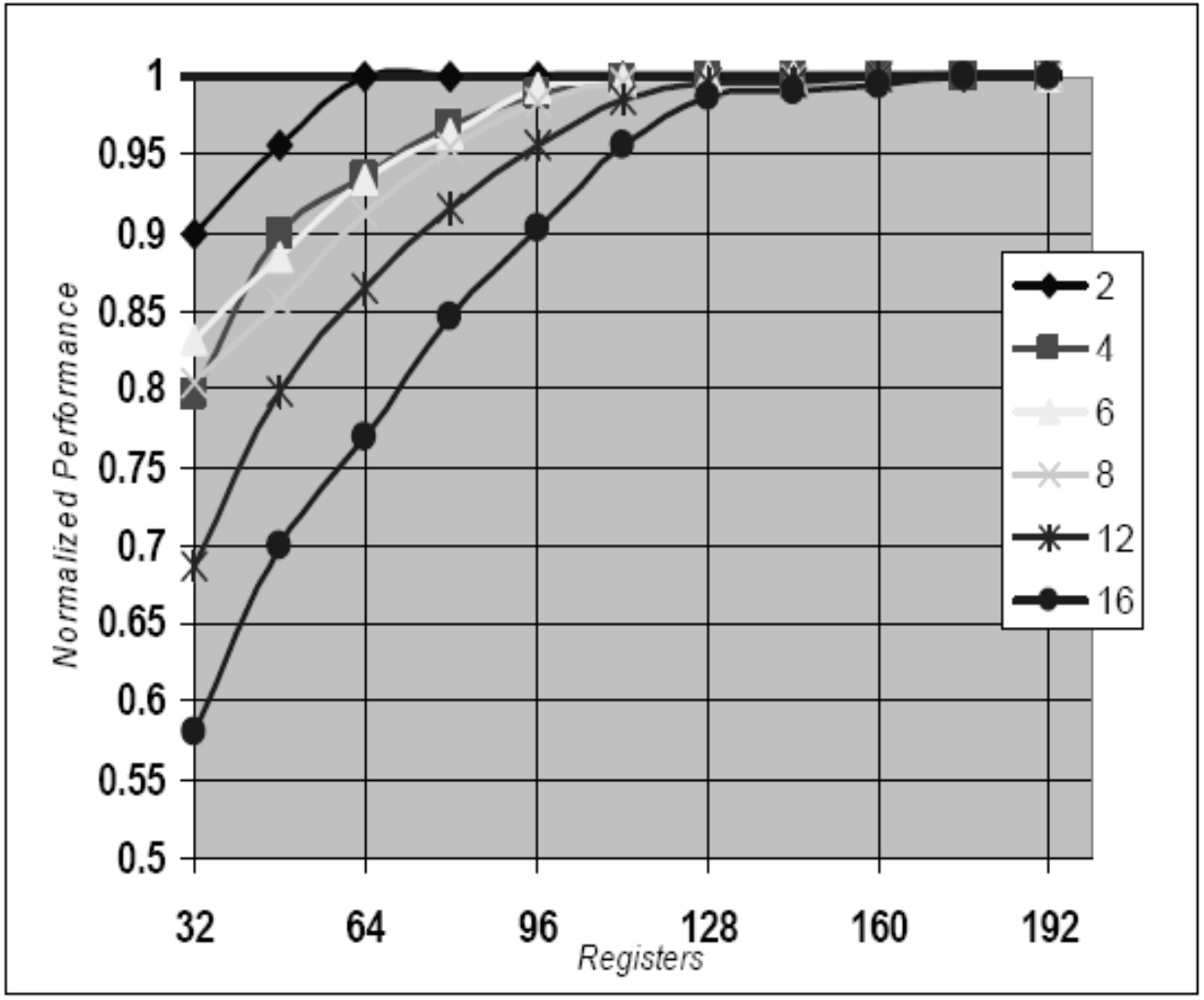


VLIW: Register pressure

Register Pressure: La latenza delle operazioni unita all'avvio multiplo delle stesse determina un aumento del numero di registri utilizzati.

- Es: Se in una istruzione ho 3 operazioni DIVF, supponendo 10 cicli di latenza, ho 3 registri inutilizzabili per un periodo di 10 long instruction, potenzialmente corrispondente a molte operazioni!

How Many Registers?



Experiment

- Media-oriented benchmarks
- Aggressive VLIW compiler
- VLIW architecture
  - 2-16 units
  - 1-8 mem ports
- Integer registers

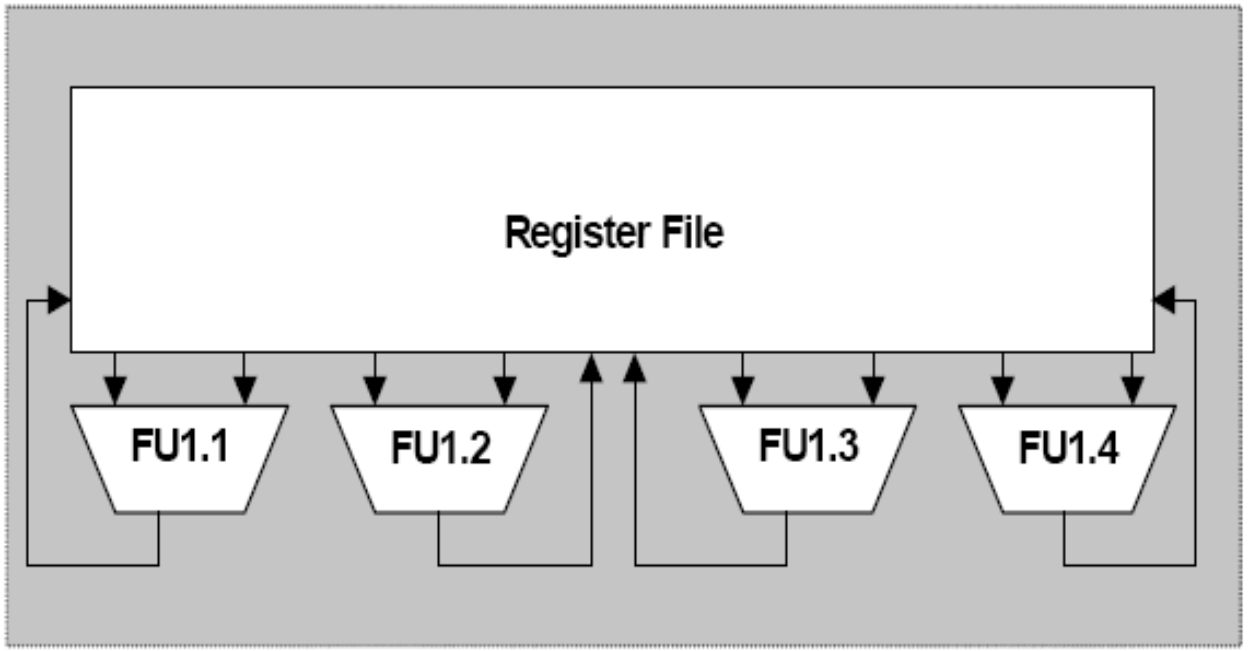
Register allocation overhead

- In mancanza di registri disponibili è necessario aggiungere delle operazioni di **spill/restore**, praticamente delle load/store per lo swapping tra registri/memoria.

- Evento non così raro poichè un registro di destinazione è inutilizzabile per tutti cicli di latenza dell'operazione che lo deve scrivere.

How Do We Keep Execution Units Busy?

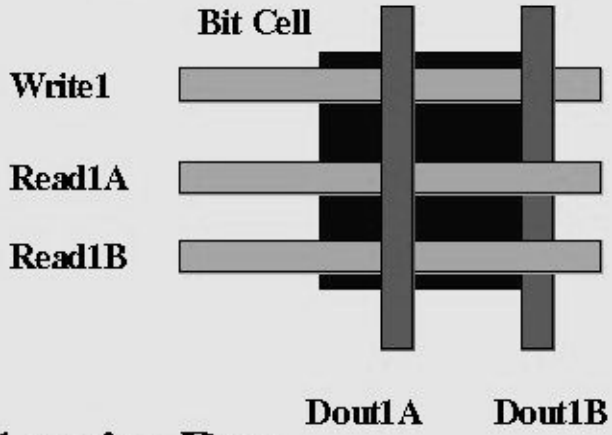
- Example:
  - A VLIW with 4 execution units (2-input, 1-output)
  - We need to read 8 and write 4 registers per cycle
  - For N execution units: 2N reads / N writes per cycle



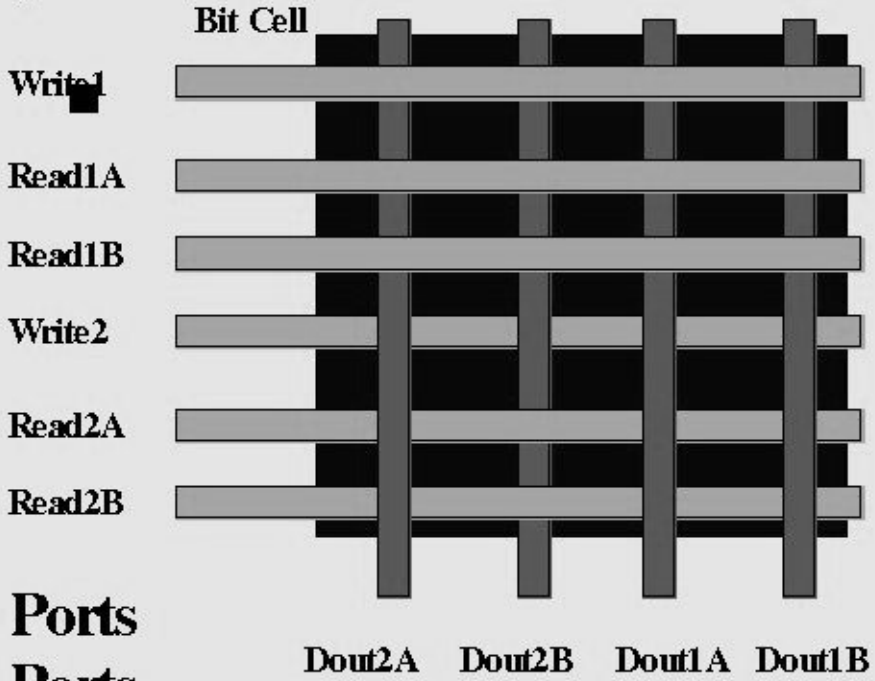
Unified Multi-Ported Register File

Register File Limits: Area

- Area of the register file grows with the square of the number of ports
  - Each port requires new routing in X and Y direction
  - Typically limited by routing



1 write Port
2 Read Ports



2 write Ports
4 Read Ports

Register Files Limits: Speed

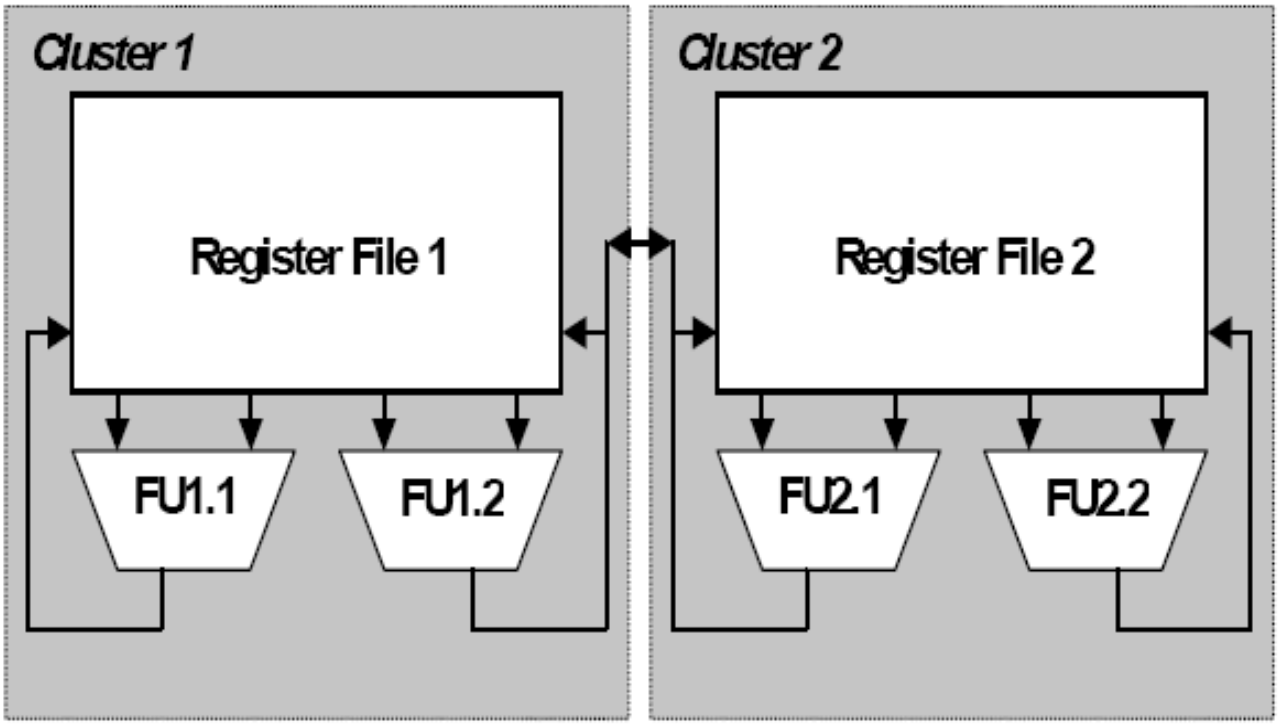
- Read access time grows linearly with the number of ports
  - Internal bit cell loading becomes larger
  - Larger area of register file causes longer wire delays
  - Longer wires and larger cells: more power-hungry

■

- What is reasonable today in terms of number of ports?
  - It changes with technology, for example in 0.18µ:
    - Max: around 15-20 ports in (read and write)
    - Sweet spot: around 12 ports (8 read, 4 write)
  - 15-20 ports can support 5-7 execution units
    - With simultaneous operand accesses

Solving the Register File Bottleneck

- Partition the register file
  - Each register file is connected to a few executions units
    - More than one unit per register file, in today's technology



Clustered VLIW

Architectural & Physical registers

- **Physical Registers** : l'insieme dei registri effettivamente presenti a livello microarchitettura, ossia nell'implementazione hardware dell'architettura.

- **Architectural Registers** : registri che sono visibili a livello di ISA, ossia dall'utilizzatore dal programmatore assembly e/o dal compilatore.

Es. 64 GPR physical registers di cui solo 32 GPR architectural registers. Sebbene ci siano fisicamente 64 registri GPR , il compilatore non può schedulare una istruzione del tipo `LW r35,vett(r4)`.

Architectural & Physical Registers

- Nelle architetture superscalari, al fine di mantenere la compatibilità del codice, il numero di architectural register è mantenuto immutato (es. 32 gpr), mentre il numeri di registri effettivamente presenti è maggiore (es. 64).

- I registri supplementari, non visibili al compilatore/programmatore, sono sfruttati dall'hardware di controllo mediante il **register renaming**, che dinamicamente permette la schedulazione di più istruzioni usando tutti i registri effettivamente presenti.

VLIW: Architectural & Physical registers

- Il compilatore VLIW, nella determinazione della sequenza di long instruction che verranno eseguite, deve conoscere il numero di registri effettivamente presenti.

- Infatti successivamente l'hardware dovrà semplicemente eseguire il plan of execution (POE), non operando alcuna modifica del codice come register renaming etc.

- **Architecturally visible registers:** in una architettura di tipo VLIW tutti i registri fisicamente presenti devono essere visibili a livello ISA. In caso contrario, quelli “invisibili” resterebbero inutilizzati!

VLIW: pro

- Semplificazione dell'hardware di controllo: minore consumo di potenza, diminuizione del critical path con conseguente possibilità di un clock più veloce
- Modificare lo schedulatore software di un compilatore è più semplice che modificare hardware di controllo.
- Ammortizzamento dei costi: se si aggiungono 10 righe nel compilatore il costo di questa modifica lo sfrutto in ogni chip prodotto. Se aggiungo 10 transistor, li devo pagare per ogni unità prodotta.
- Scalabilità: è possibile supportare alti livelli di ILP aumentando la iussue-width, senza che ciò comporti un aumento di complessità hardware.

Power Dissipation Fundamentals

- Power / Energy have become *very* important
  - Fundamental in embedded and battery-operated appliances
  - Power often limits even high-end processors speeds
- Two components of power dissipation in a VLSI circuit
  - Leakage (static, ∝ number of transistors)
    - Not much to do, except powering off
  - Switching (dynamic, ∝ number of transitions)

$$\frac{V_{dd}^{\,2}}{2}\sum_{i}^{nets} f_s \cdot C_i \approx k \cdot f_s \cdot V_{dd}^{\,2}$$

- Reducing frequency and voltage saves energy

$$\frac{E_1}{E_2} \approx \frac{k \cdot t_1 \cdot f_1 \cdot V_1^2}{k \cdot t_2 \cdot f_2 \cdot V_2^2} \approx \frac{t_1 \cdot f_1 \cdot V_1^2}{t_1 \cdot \frac{f_1}{f_2} f_2 \cdot V_2^2} = \left(\frac{V_1}{V_2}\right)^2$$

VLIWs Are Energy Efficient

- Two strategies to meet a real-time deadline
  - 1. Increase clock speed
  - 2. Increase ILP
- Increasing ILP is more energy efficient
  - Adding resources increases energy requirements linearly
  - However, we can run at lower frequency and voltage
    - Lower voltage has a quadratic effect on power!
- Simple Example
  - M1: Scalar (1 unit), f = 500MHz, V = 1.8V
  - M2: VLIW (4 units), f = 250MHz, V = 1.3V
    - Assume same performance, i.e. a sustained average IPC of 2
  - At same *f* and *V,* M2 uses 2x the power of M1 (very conservative)
  - M2 (VLIW) requires ~half the energy of M1 for the same task!

$$\frac{E_1}{E_2} \approx \frac{t \cdot k_1 \cdot f_1 \cdot V_1^2}{t \cdot k_2 \cdot f_2 \cdot V_2^2} = \frac{k_1 \cdot 500 \cdot (1.8)^2}{2 \cdot k_1 \cdot 250 \cdot (1.3)^2} = 1.9$$

VLIW: contro

Binary incompatibility: il codice assembly e quindi il binario prodotto dipende fortemente dal numero ed il tipo di unità funzionali e dalle latenze operazionali. Es: Il codice binario prodotto per una implementazione VLIW con 3 unità FP non potrà andare in esecuzione in un processore identico ma con 2 unità FP.

- Il parallelismo intrinseco dell'applicazione da eseguire potrebbe non consentire l'ottenimento di ILP elevati e quindi non giustificare l'utilizzo di un approccio VLIW based. (es. Applicativi desktop...)

VLIW: prospettiva storica

- Nei primi anni 80 molte startup companies cercavano di rimpiazzare i costosi minicomputer per elaborazioni scientifiche con soluzioni VLIW based (Multiflow, Cydrome)
- Tuttavia non vi era ancora una densità di transistor sufficiente per implementare un VLIW con molte risorse hardware in un unico chip
- Inoltre la ricerca nell'ambito di compilatori capaci di ottenere un alto ILP era in una fase iniziale
- Di fatto, nessun prodotto è sopravvissuto alla rivoluzione dei microprocessori degli anni 90

Riscoperta VLIW nell'embedded multimedia(1/2)

- Dati fortemente parallelizzabili:codifica di stream audio/video parti diverse di una immagine possono essere trattate in parallelo da un codec jpeg.

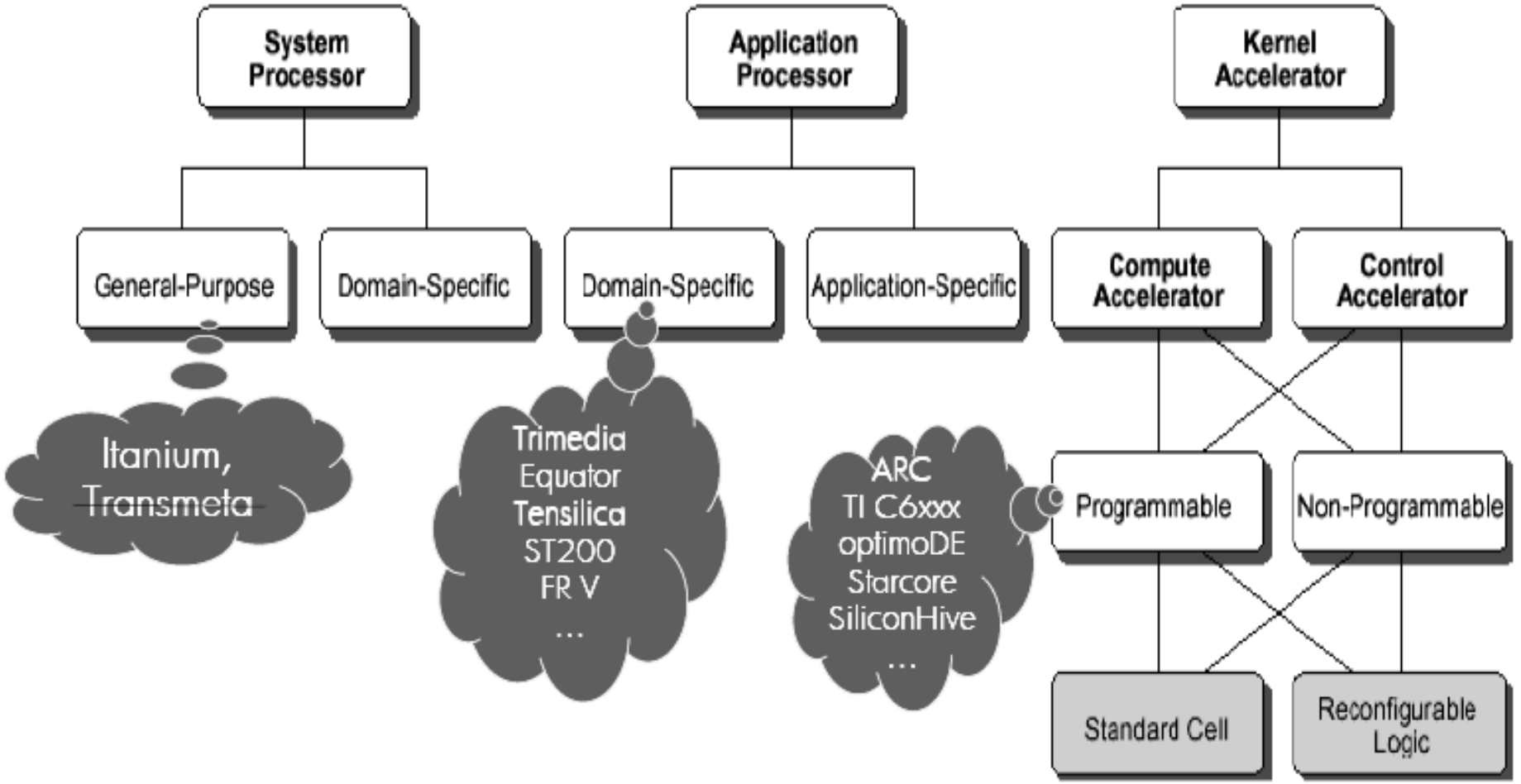
- Non necessità di compatibilità del codice binario tra i vari prodotti: Il software è molto specifico e fornito dal produttore direttamente con l'hardware. Es: le persone non si scambiano il software che sta dentro le fotocamere digitali, un modello successivo di fotocamera non deve essere compatibile col precedente software: il produttore compilerà ed installerà quello nuovo.

Riscoperta VLIW nell'embedded (2/2)

- Semplificazione hardware: meno transistor necessari, minor consumo di potenza.
- **ASIP** (*application-specific instruction set processors*): posso tarare su misura la quantità di risorse hardware (unità funzionali, registri etc..) per raggiungere il livello di prestazioni richiesto da una certa classe di applicazioni.

- Si potrebbe sfruttare la “sovrabbondanza di prestazioni”. Es. supponiamo che con un processore classico si ha un tempo X con un clock a 100 Mhz. Se sfruttando l'ILP con lo stesso clock ha un tempo X/2, potrei portare il clock a 50 Mhz risparmiando potenza su tutto il circuito e ottenendo un tempo X.

Where is VLIW today?



VLIW: limitazioni sull'ILP

Il numero di istruzioni che possono essere schedulate in parallelo, ossia il livello di ILP raggiungibile dipende da:

- Disponibilità di risorse hardware disponibili (unità funzionali, registri etc..)
- Parallelismo intrinseco presente nell'applicazione
- Tecniche di compilazione (es. Loop unrolling , Predicated execution , speculative execution)

Basic Block

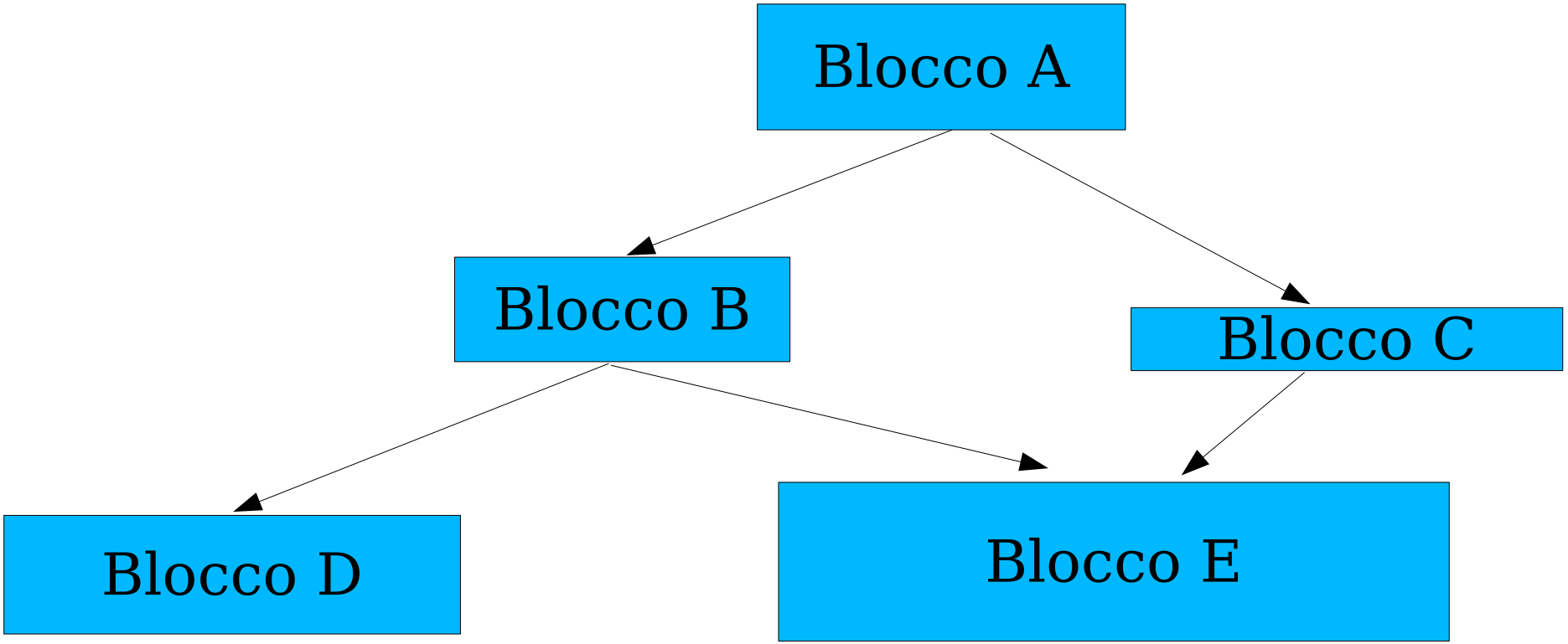
E' una sequenza di codice lineare avente un punto di ingresso e un punto di uscita. Esempio :

```
loop: lw r1,vett(r2) <------ punto di ingresso
      addi r2,r2,4
      sw vett2(r2),r1
      subi r3,r3,1
      bnez r3,loop <-------- punto di uscita
```

- Si accede al punto di ingresso quando il Program Counter arriva all'indirizzo dell'etichetta `loop`.
- Si esce solamente da un punto, corrispondente ad una istruzione di salto o diramazione.

Control Flow Graph

- Può essere utile rappresentare la struttura del codice mediante un **Control Flow Graph (CFG)** dove ogni basic block è un nodo e le connessioni tra i nodi sono le diramazioni che possono avvenire durante l'esecuzione



VLIW: limitazioni sull'ILP

- Uno dei fattori che limita maggiormente il livello di ILP ottenibile è rappresentato dalla presenza di **conditional branches**.

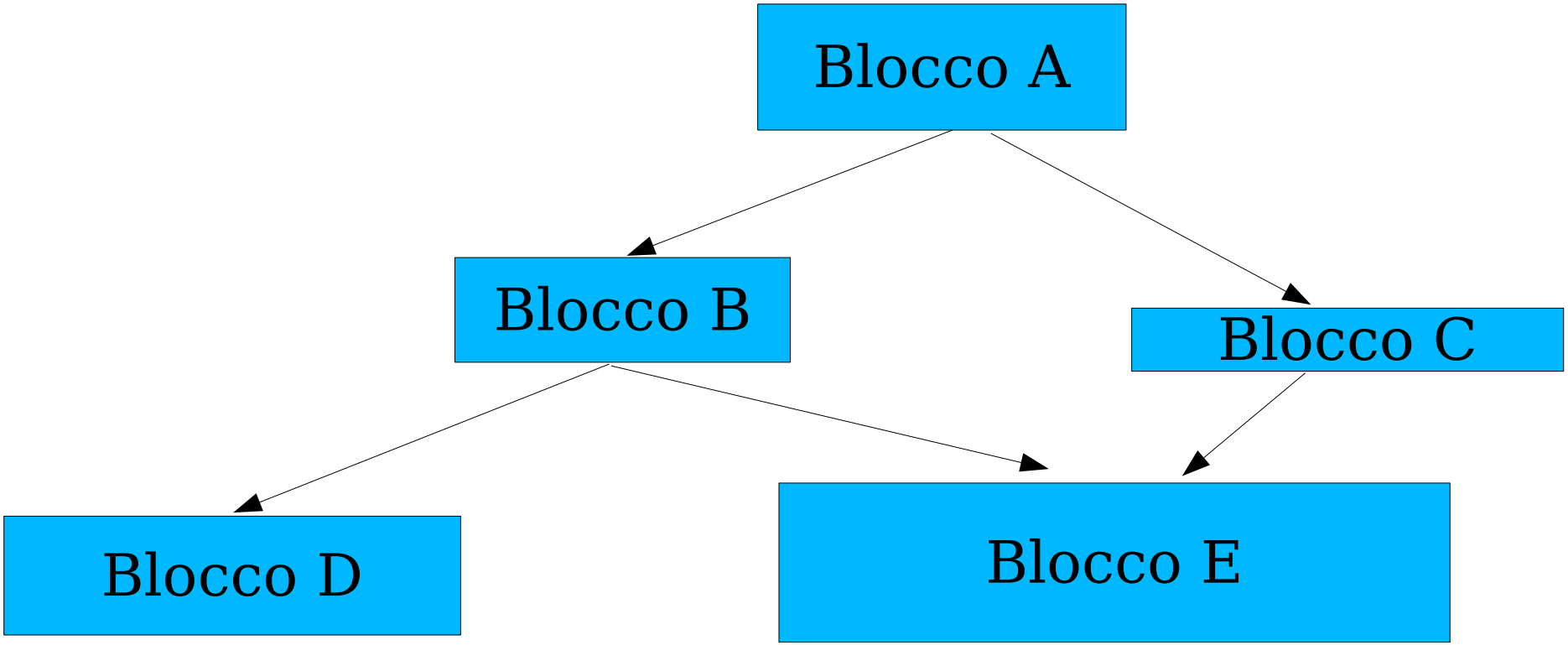
- Tali branch frammentano le sequenza di istruzioni in più **basic block**

- Il “raggio di azione” dello schedulatore è confinato all'interno dei basic block il compilatore agire su porzioni limitate di codice.

- Infatti le operazioni che risiedono su basic block differenti non possono essere messe in parallelo poiché non si sa, al momento della compilazione, quale direzione verrà presa nei branch!

VLIW: limitazioni sull'ILP

Non posso schedulare in parallelo istruzioni appartenenti a basic blocks diversi, poichè la loro esecuzione dipende dalle diramazioni prese durante l'esecuzione.



Predicated Execution (1/3)

Per superare la limitazione introdotta dai conditional branch, i compilatori VLIW possono usare delle **istruzioni predicate** per implementare la tecnica detta **if-conversion**.

Predicate registers: registri di dimensione 1 bit, contenenti valori booleani.

Una **predicated instruction**, oltre ai soliti operandi, un campo aggiuntivo che specifica un predicate register. Esempio:

sw vett(r1),r7,p2

oppure: addi r2,r2,4,p7

Predicated Execution (2/3)

La semantica delle istruzioni predicate è:

- Se il predicate register = '1' allora esegui normalmente
- Se il predicate register è = '0' allora tratta come NOP

Per questo motivo è uso comune indicare per chiarezza nella notazione di una istruzione predicata l'operando aggiuntivo preceduto da 'if' :

```
ADDI r2,r2,4 if p5
```

NB. E' solo una simbologia! abbiamo una singola istruzione, una ADD con 4 operandi: 2 registri GPR, un immediato, e un predicate register.

Predicate Execution (3/3)

Il valore dei registri predicate viene settato tramite speciali istruzioni dette **compare to predicate.**

Esse verificano una certa condizione <cond> e settano un opportuna coppia di predicate register con valori complementari. Es:

```
CMPP.<cond> p1,p2,r4,r5
```

Ci sono svariate istruzioni, a seconda di qual'è la condizione da verificare. Es. <cond>=LE (less equal)

`CMPP.LE p1,p2,r4,r5` significa:

se r4<=r5 setta p1 = 1, p2 =0
atrimenti setta p1 = 0, p2 =1

ILP oriented compiling: if-conversion (1/3)

```
If (a<b) c=a;
else
          c=b;
if (d<e) f=d;
else
          f=e;
```

Supponendo una issue width di 4 slot, il codice può essere schedulato in due sole long instruction. NB: per semplicità nell'esempio in basso usiamo a,b,c,d ma in realta' andrebbero usati i registri in cui i rispettivi valori sono stati caricati.

`CMPP p1,p2,a,b`	`CMPP p3,p4,d,e`	NOP	NOP
`c=a if p1`	`c=b if p2`	`f=d if p3`	`f=e if p4`

ILP oriented compiling: if conversion (2/3)

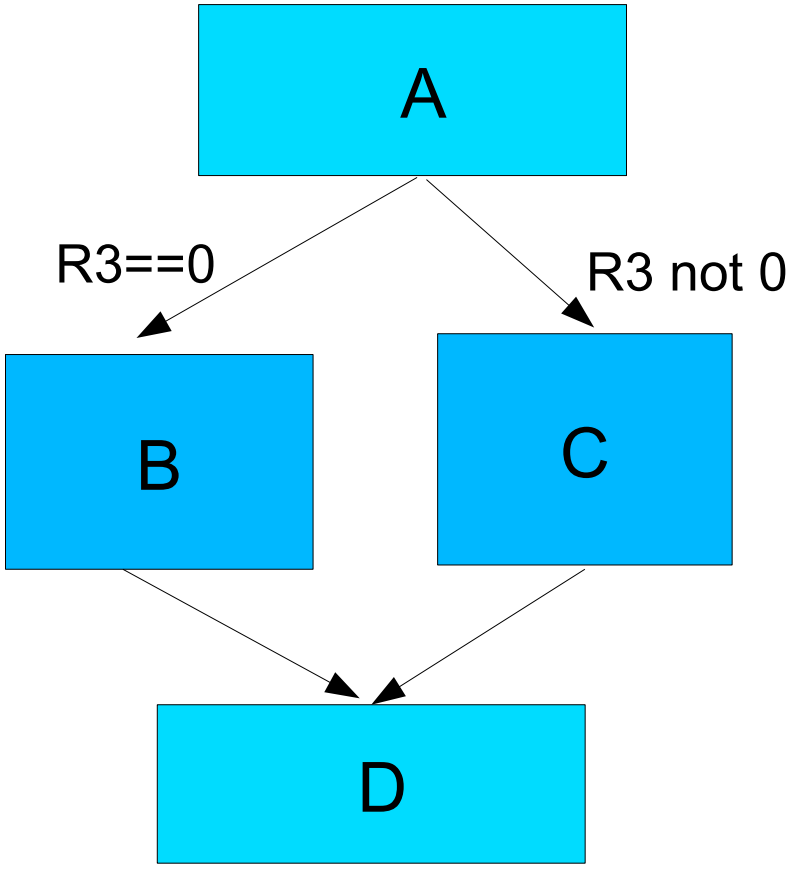
•Tramite la **if-conversion** un branch condizionale viene dunque rimpiazzato da una istruzione che setta una coppia di predicate register con valori complementari.

•Le istruzioni che appartenevano ad ognuna delle due diramazioni possono tranquillamente essere schedulate in parallelo, poichè la loro effettiva esecuzione è vincolata al valore booleano contenuto in appositi predicate registers.

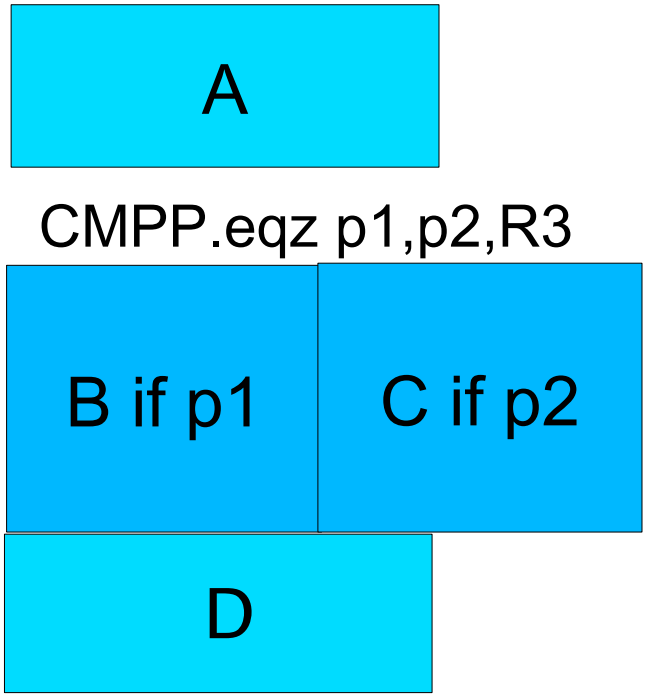
Si è trasformata una dipendenza di controllo in una dipendenza sui dai

ILP oriented compiling: if-conversion (3/3)

Senza if-conversion



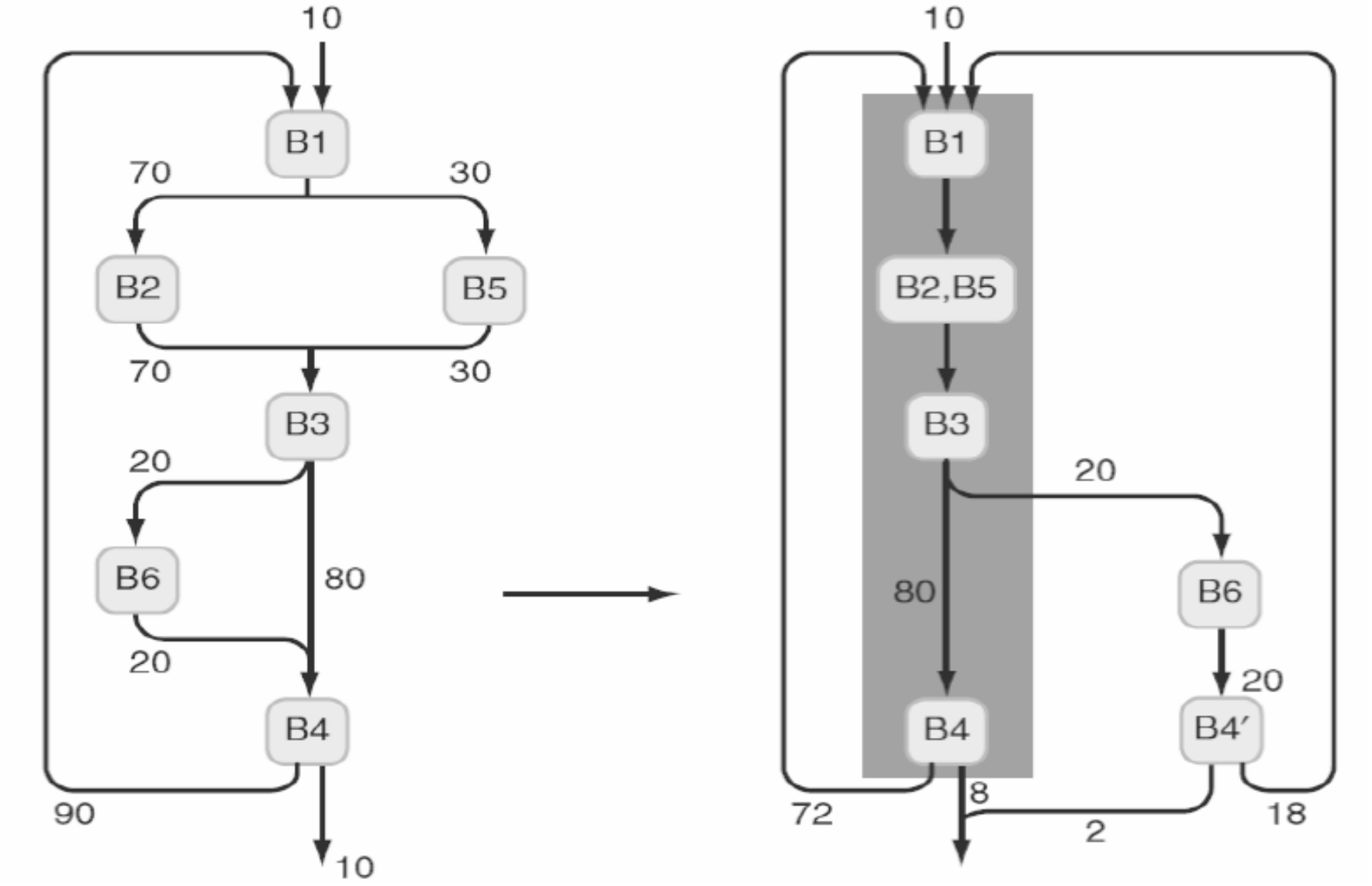
con if-conversion



ILP oriented compiling: region formation

- Una **region** è formata combinando più basic block appartenenti a differenti path di esecuzione, magari scelti in base ad una fase preliminare di profiling per determinare i percorsi piu' probabili.
- Esempio di region:
  - **Hyperblock**: singolo punto di ingresso, molteplici punti di uscita
- La formazione di un hyperblock avviene in due fasi:
  - Eliminazione dei branch interni mediante **if-conversion**
  - Eliminazione degli ingressi laterali mediante **tail duplication**

Hyperblock


10
B1
70
30
B2
B5
70
30
B3
20
B6
80
20
B4
90
10
10
B1
B2,B5
B3
20
80
B6
20
B4
B4′
8
72
2
18