Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

Università di Pisa

Master Program (Laurea Magistrale) in Computer Science and Networking

High Performance Computing Systems and Enabling Platforms

Marco Vanneschi

2. Instruction Level Parallelism

2.1. Pipelined CPU

2.2. More parallelism, data-flow model

Improving performance through parallelism

Sequential Computer

- The sequential programming assumption
  - User's applications exploit the sequential paradigm: technology pull or technology push?
- Sequential cooperation P – MMU – Cache – Main Memory – …
  - Request-response cooperation: low efficiency, limited performance
  - *Caching*: performance improvement through ***latency reduction*** (memory access time), even with sequential cooperation
- Further, big improvement: *increasing CPU bandwidth*, through CPU-internal parallelism
  - **Instruction Level Parallelism (ILP)**: several instructions executed in parallel
  - *Parallelism is hidden to the programmer*
    - The sequential programming assumption still holds
  - Parallelism is **exploited at the firmware level** *(parallelization of the firmware interpreter of assembler machine)*
  - **Compiler optimizations** of the sequential program, exploiting the firmware parallelism at best

ILP: which parallelism paradigm?

Parallelism paradigms / parallelism forms

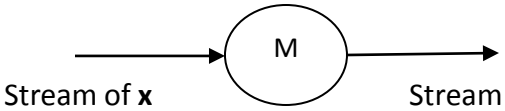
- Stream parallelism
  - Pipeline
  - Farm
  - Data-flow
  - …
- Data parallelism
  - Map
  - Stencil
  - …
- Stream + data parallelism
  - Map / stencil operating on streams
  - …

All forms are feasible:

- Pipeline → Pipelined implementation of the firmware interpreter
- Farm → Replication of some parts of the firmware interpreter
- Data-flow → Data-flow ordering of instructions
- Data parallelism → Vectorized instructions
- Stream + data parallelism → Vectorized implementation of some parts of the firmware interpreter

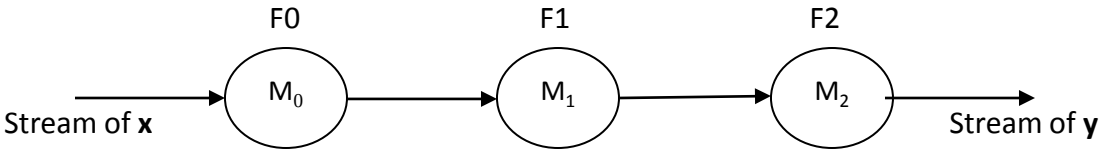
Pipeline paradigm in general

Module operating on stream



$\forall x : y = F(x) = F2(F1(F0(x)))$

$T_{seq} = T_F = T_{F0} + T_{F1} + T_{F2}$



Parallelism degree: n = 3

$T_0 = T_{F0} + T_{com}$ $T_1 = T_{F1} + T_{com}$ $T_2 = T_{F2} + T_{com}$

Service Time: $T_{pipe} = max(T_0, T_1, \ldots, T_{n-1})$ *parallelism degree n* $T_{pipe\text{-}id} = T_{seq} / n$

Efficiency: $\varepsilon = T_{pipe\text{-}id}/T_{pipe} = T_{seq} / (n \max(T_0, T_1, \ldots, T_{n-1}))$

if $T_{F0} = T_{F1} = \ldots = T_{Fn-1} = t$: $T_{pipe} = t + T_{com}$ balanced pipeline

if $T_{com} = 0$: $T_{pipe} = T_{pipe\text{-}id} = T_{seq} / n$ fully overlapped communications

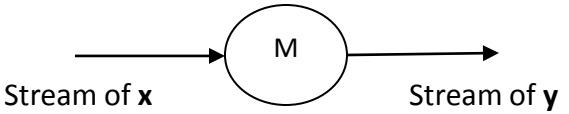
For balanced, fully overlapped pipeline structures:

processing bandwidth: $B_{pipe} = B_{pipe\text{-}id} = n\, B_{seq}$

efficiency: $\varepsilon = 1$

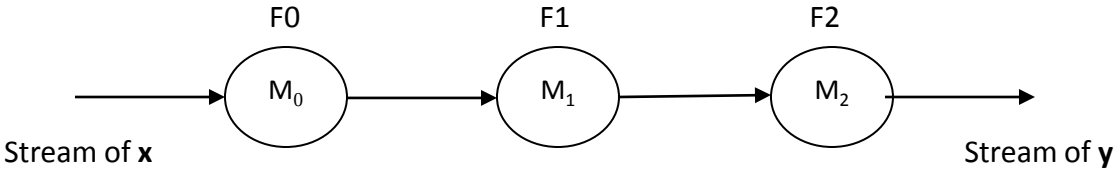
Pipeline latency

Module operating on stream



$\forall x : y = F(x) = F2(F1(F0(x)))$

latency $L_{seq} = T_{seq} = T_F = T_{F0} + T_{F1} + T_{F2}$

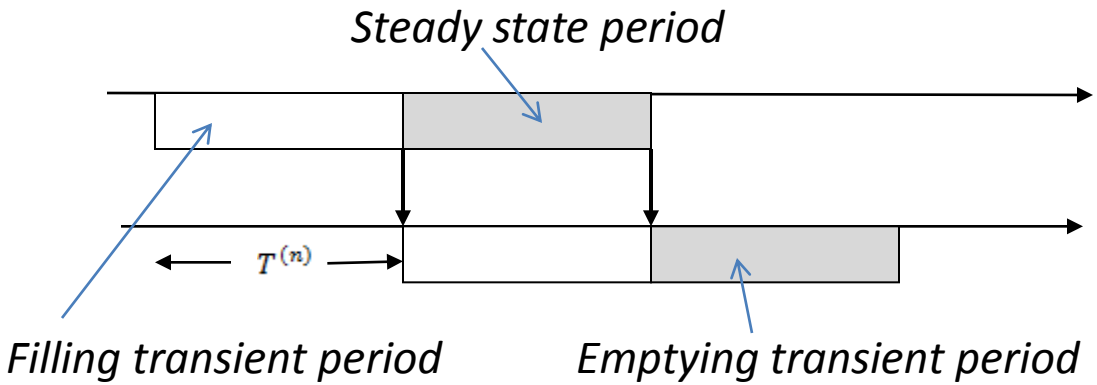


$L_0 = T_{F0} + L_{com}$ $L_1 = T_{F1} + L_{com}$ $L_2 = T_{F2} + L_{com}$

Latency: $L^{(n)} = \sum_{i=1}^{n} (L_i + L_{com}) \geq L_{seq}$

Bandwidth improvement
at the expense of **increased latency** wrt the sequential implementation.

Pipeline completion time



m = stream length (= 2)
n = parallelism degree (= 2)
$T^{(n)}$ = service time with parallelism degree n

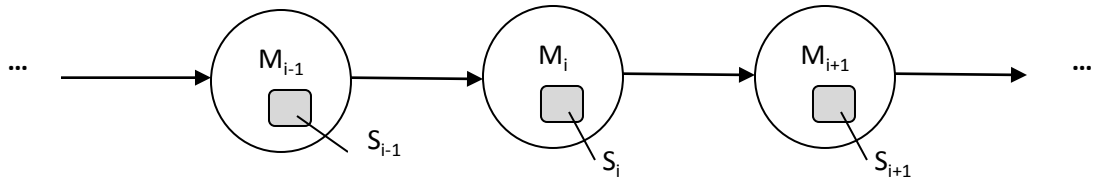
$$T_c = (m + n - 1)\, T^{(n)}$$

If $m >> n$:

$$T_c \sim m\, T^{(n)}$$

The *fundamental relation* between completion time and service time, valid for "**long streams**" (e.g. instruction sequence of a program …)

Pipeline with internal state



- Computation with internal state: for each stream element, the output value depends also on an internal state variable, and the internal state is updated.
- **Partitioned internal state:**

  if disjoint partitions of the state variable can be recognized

  and they are encapsulated into distinct stages

  and each stage operates on its own state partition only,

  then the presence of internal state has no impact on service time.

- However, if state partitions are related each other, i.e. if there is a **STATE CONSISTENCY PROBLEM**, then some degradation of the service time exists.

Pipelining and loop unfolding

```
M ::    int A[m]; int s, s0;
        { < receive s0 from input_stream >;
          s = s0 ;
          for (i = 0; i < m; i++)
                    s = F (s, A[i]);
        < send s onto output_stream >
        }
```

k-unfolding

```
M ::    int A[m]; int s;
        { < receive s0 from input_stream >;
          s = s0;
          for (i = 0; i < m/k; i++)
                    s = F (s, A[i]);
          for (i = m/k; i < 2m/k; i++)
                    s = F (s, A[i]);
          …
          for (i = (k-1)m/k ; i < m; i++)
                    s = F (s, A[i]);

          < send s onto output_stream >
        }
```

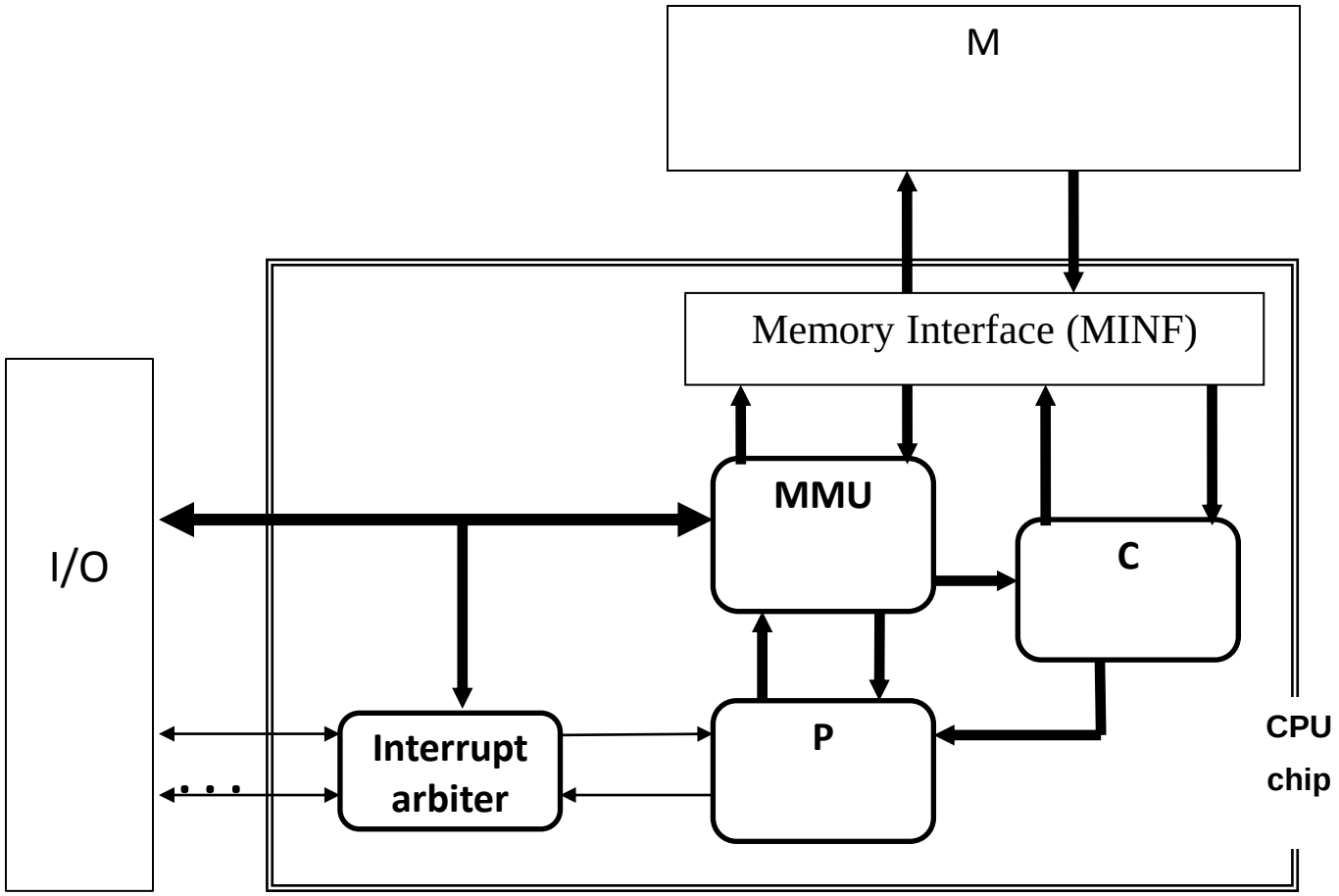
k times

k-stage balanced pipeline with A partitioned (k partitions)

Formally, this transformation implements a form of stream data-parallelism with stencil:

- simple stencil (linear chain),
- optimal service time,
- but increased latency compared to non-pipelined data-parallel structures.

From sequential CPU …



… to pipelined CPU

Parallelization of the firmware interpreter

while (true) *do*

{ instruction fetch (IC);

instruction decoding;

(possible) operand fetch;

execution;

(possible) result writing;

IC update;

< interrupt handling: firmware phase >

}

Generate stream

- instruction stream

Recognize pipeline stages

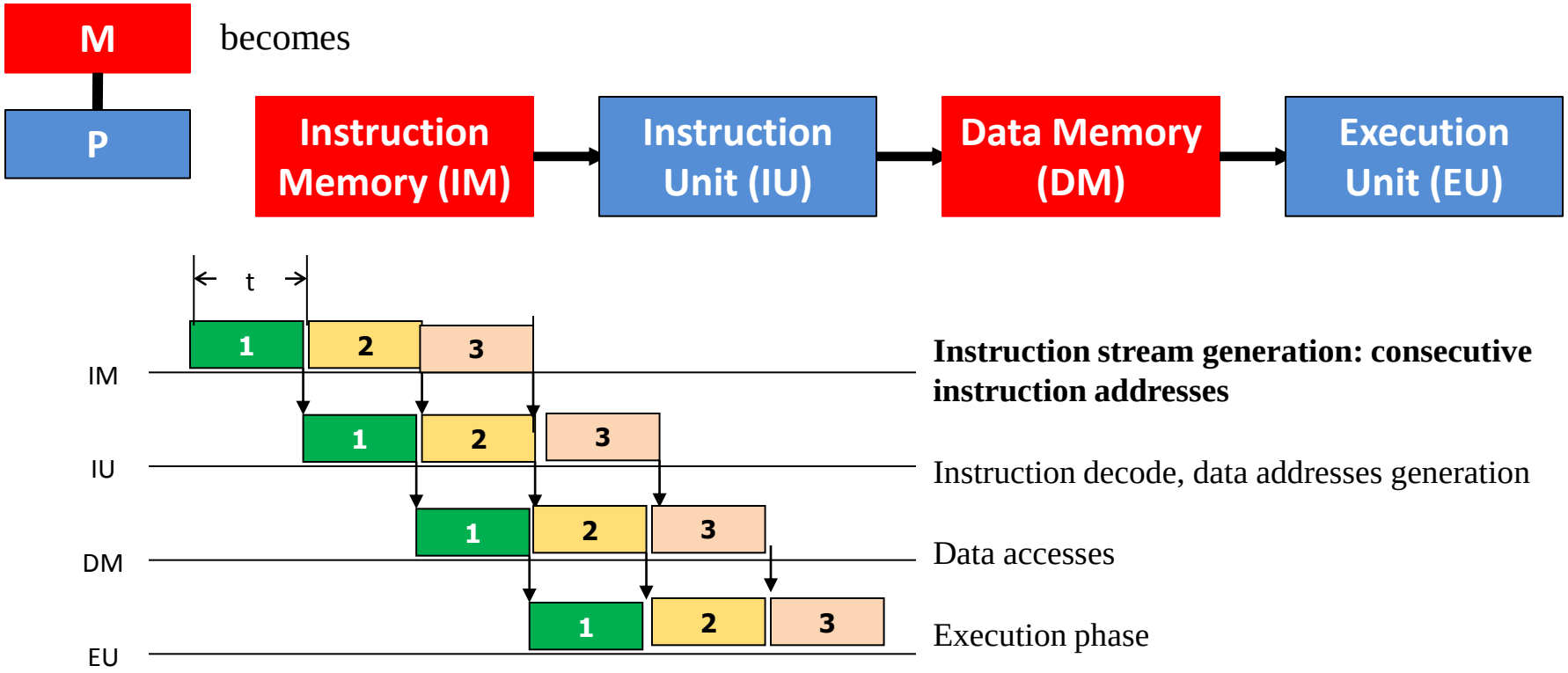
- corresponding to interpreter phases

Load balance of stages

Pipeline with internal state

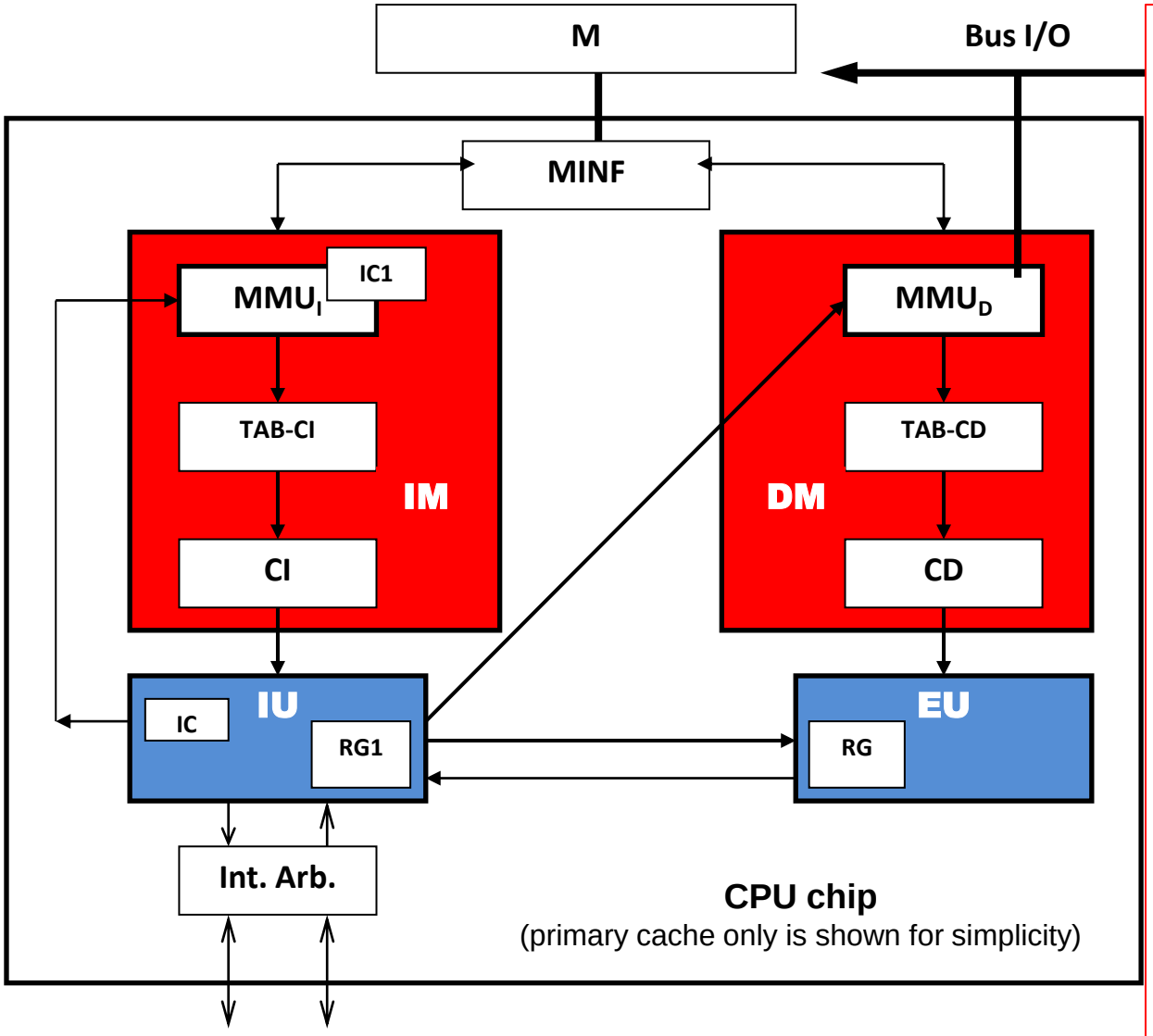
- problems of state consistency
- *"impure" pipeline*
- impact on performance

Basic principle of Pipelined CPU: simplified view



In principle, a continuous stream of instructions feeds the pipelined execution of the firmware interpreter;
each pipeline stage corresponds to an interpreter phase.

Pipelined CPU: an example



- IM: Instruction Memory
- MMU_I: Instruction Memory MMU
- CI: Instruction Cache
- TAB-CI: Instruction Cache Relocation

- DM: DataMemory
- MMU_D: Data Memory MMU
- CD: Data Cache
- TAB-CD: Data Cache Relocation

- IU: Instruction prepation Unit; possibly parallel
- EU: instruction Execution Unit ; possibly parallel/pipelined
- RG: General Registers, primary copy
- RG1: General Registers, secondary copy
- IC: program counter, primary copy
- IC1: program counter, secondary copy

Main tasks of the CPU units (simplified)

- IM : generates the instruction stream at consecutive addresses (IC1); if a new value IC1 is received by IU, re-starts the stream generation starting from this value
  - IM service time = 1τ + Tcom = 2τ ($T_{tr} = 0$)

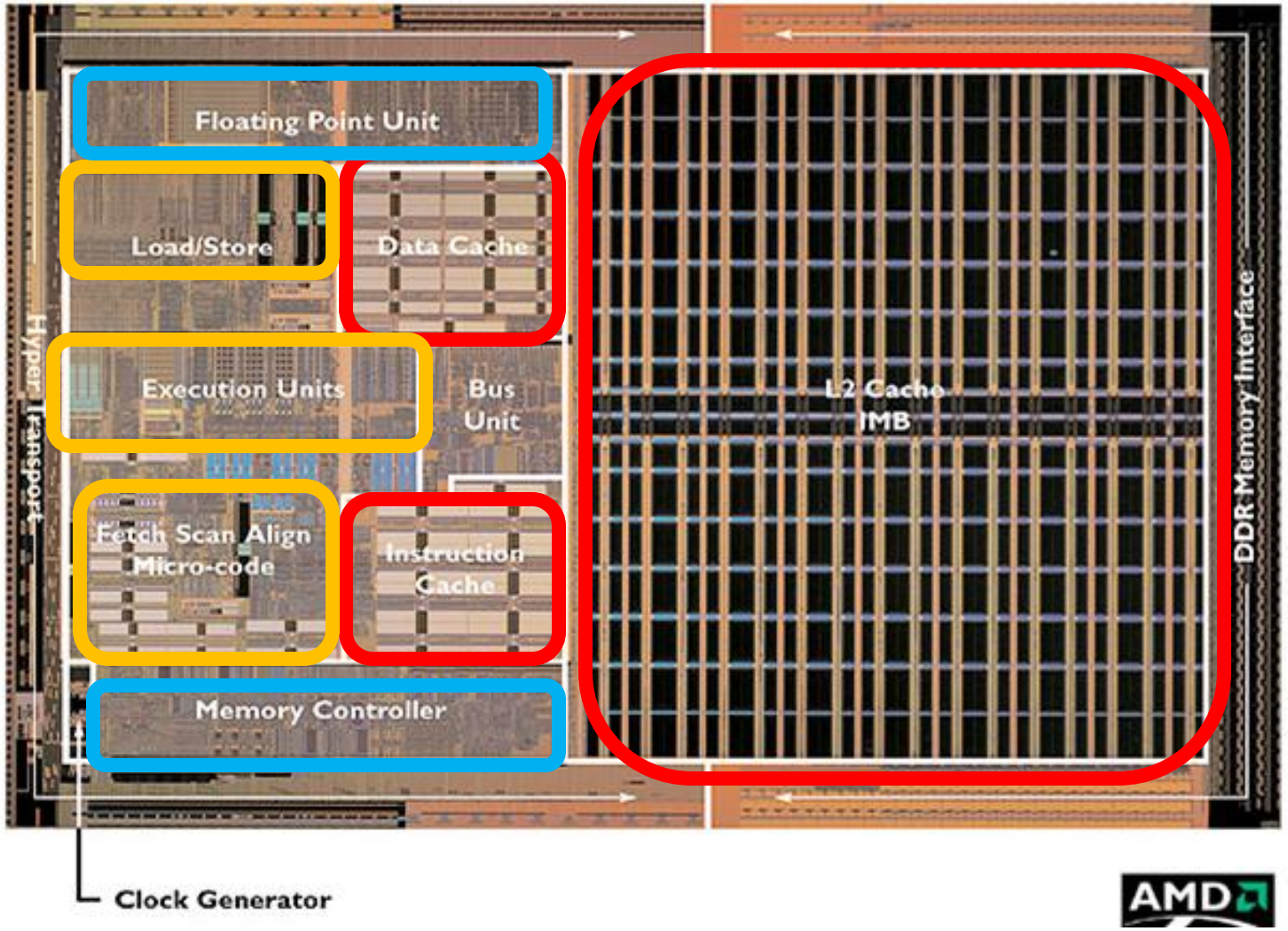
- IU : receives instruction from IM; *if valid* then, according to instruction class:
  - if arithmetic instruction: sends instruction to EU
  - if LOAD: sends instruction to EU, computes operand address *on updated registers*, sends reading request to DM
  - if STORE: sends instruction to EU, computes operand address *on updated registers*, sends writing request to DM (address and data)
  - if IF: verifies branch condition on general registers; if true then updates IC with the target address and sends IC value to IM, otherwise increments IC
  - if GOTO: updates IC with the target address and sends IC value to IM
- IU service time = 1τ + Tcom = 2τ ($T_{tr} = 0$)

Main tasks of the CPU units (simplified)

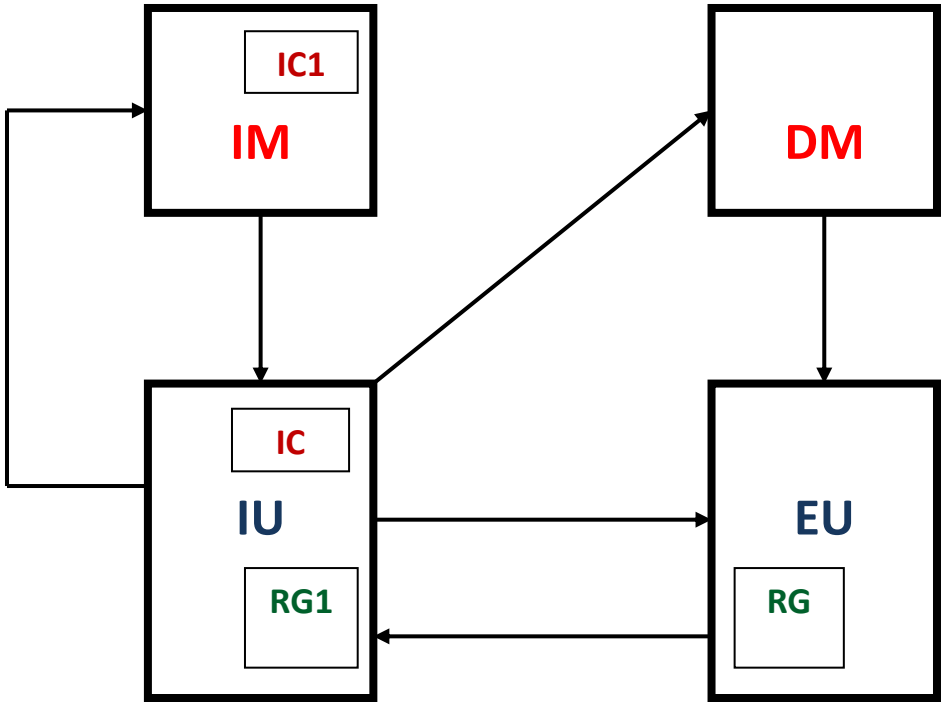
- DM : receives request from IU and executes it; if a reading operation then sends the read value to EU
  - DM service time = 1τ + Tcom = 2τ ($T_{tr} = 0$)

- EU : receives request from IU and executes it:
  - if arithmetic instruction: executes the corresponding operation, stores the results into the destination register, and sends it to IU
  - if LOAD: wait for operand from DM, stores it into the destination register, and sends it to IU.
- EU service time = 1τ + Tcom = 2τ ($T_{tr} = 0$); latency may be 1τ or greater

AMD Opteron X4 (Barcelona)



Pipeline abstract architecture



The compiler can easily *simulate* the program execution on this abstract architecture, or an *analytical cost model* can be derived.

The CPU architecture "as seen" by the compiler

It captures the essential features for defining an effective **cost model** of the CPU

First assumption: ***Pure Risc*** *(e.g. D-RISC)*

- all stages have the same service time: *well balanced*:

$$t = T_{id}$$

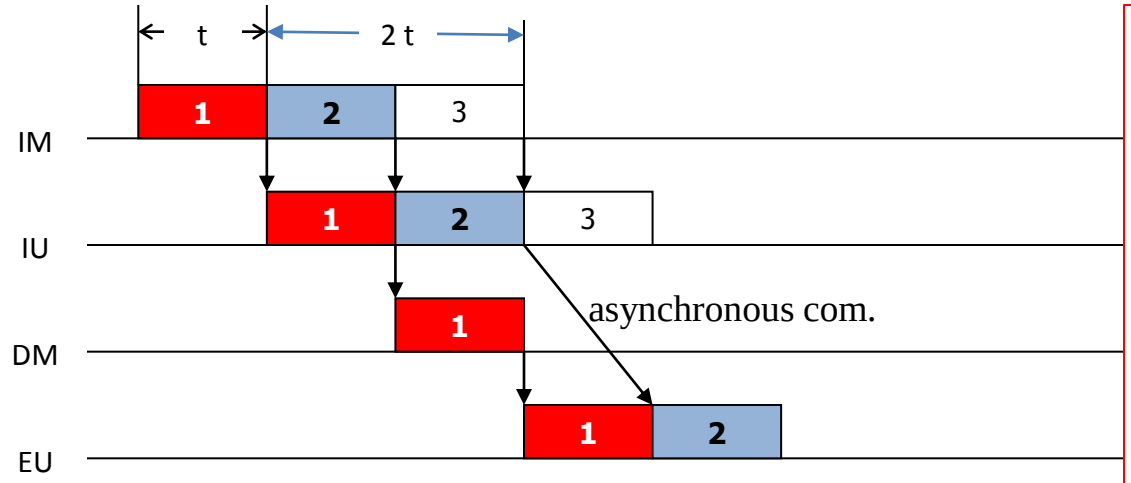
- $T_{tr} = 0$: if $t \geq 2\tau \Rightarrow Tcom = 0$

for very fine grain operations, all communications are fully overlapped to internal calculations

Example 1

1.	LOAD	R1, R0, R4
2.	ADD	R4, R5, R6
3.	…	

Execution *simulation:*
to determine ***service time*** *T and* ***efficiency*** ε



$$T = \frac{2t}{2} = t = T_{id}$$

$$\varepsilon = \frac{T_{id}}{T} = 1$$

Instruction sequences of this kind are able to fully exploit the pipelined CPU.

In fact, in this case CPU behaves as a "pure" pipeline.

No performance degradation.

"Skipping" DM stage has no impact on service time, in this example.

Peak performance

- For $t \geq 2\tau \Rightarrow Tcom = 0$
- Assume: **$t = 2\tau$** for all processing units
  - including EU: arithmetic operations have **service time ≤ 2τ**
  - **floating point arithmetic**: **pipeline** implementation (e.g., 4 or 8 stages …)
  - for integer arithmetic: **latency ≤ 2τ** (e.g., hardware multiply/divide)
  - however, this latency does not hold for **floating point** arithmetic, e.g. 4 or 8 times greater
- **CPU Service time**: $T = T_{id} = t$
- **Performance:**

$$\mathcal{P}_{id} = 1/T_{id} = 1/2\tau = f_{clock}/2 \quad \text{instructions/sec}$$

  - e.g. f_{clock} = 4 GH2, $\mathcal{P}_{id}$ = 2 GIPS
- Meaning: an instruction is fetched every 2τ
  - i.e., a new instruction is processed **every 2τ**
  - from the evaluation viewpoint***, the burden between assembler and firmware is vanishing!***
    - *marginal advantage of firmware implementation: microinstruction parallelism*
- Further improvements (e.g., superscalar) will lead to

$$\mathcal{P}_{id} = w/\tau = w\, f_{clock} \quad , \quad w \geq 1$$

Performance degradations

- For some programs, the CPU is not able to operate as a “pure” pipeline
- Consistency problems on replicated copies:
  - **RG** in EU, **RG1** in IU associated synchronization mechanism
  - **IC** in IU, **IC1** in IM associated synchronization mechanism
- In order to guarantee consistency through synchronization, some “*bubbles*” are introduced in the pipeline flow
  - **increased service time**
- Consistency of RG, RG1: **LOGICAL DEPENDENCIES ON GENERAL REGISTERS** (“data hazards”)
- Consistency of IC, IC1: **BRANCH DEGRADATIONS** (“control hazards or branch hazards”)

Example 2: branch degradation

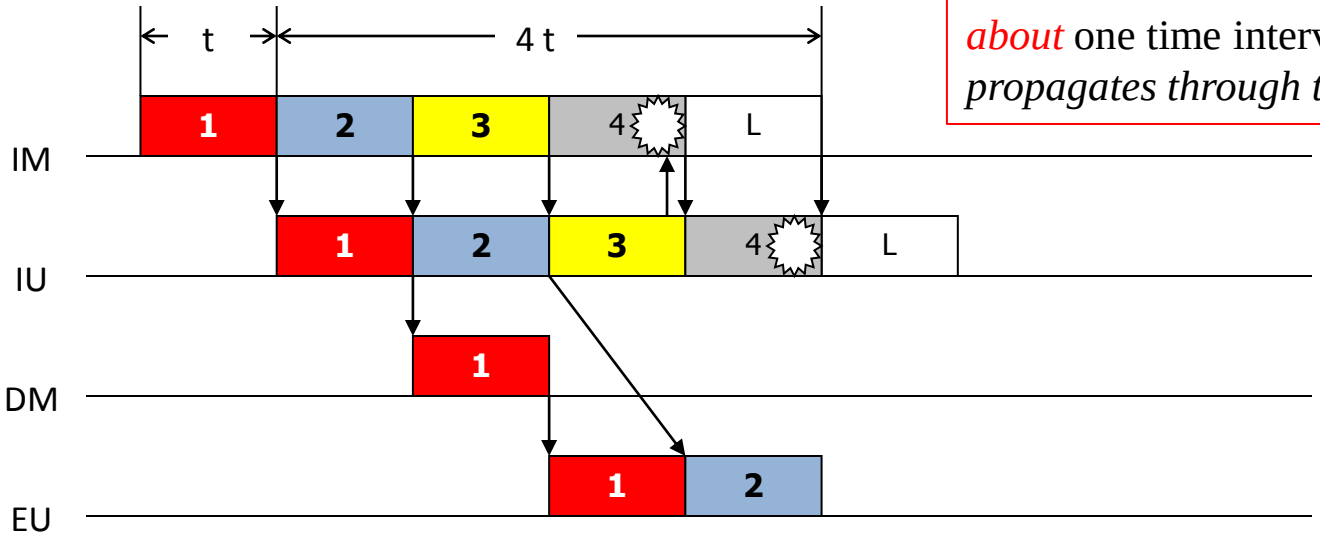
```
1.    LOAD    R1, R0, R4
2.    ADD     R4, R5, R6
3.    GOTO    L
4.    ...
```


"bubble":

IM performs a *wasted* fetch (instruction 4),

instruction 4 must be **discarded** by IU,

about one time interval *t* is wasted and it *propagates through the pipeline.*



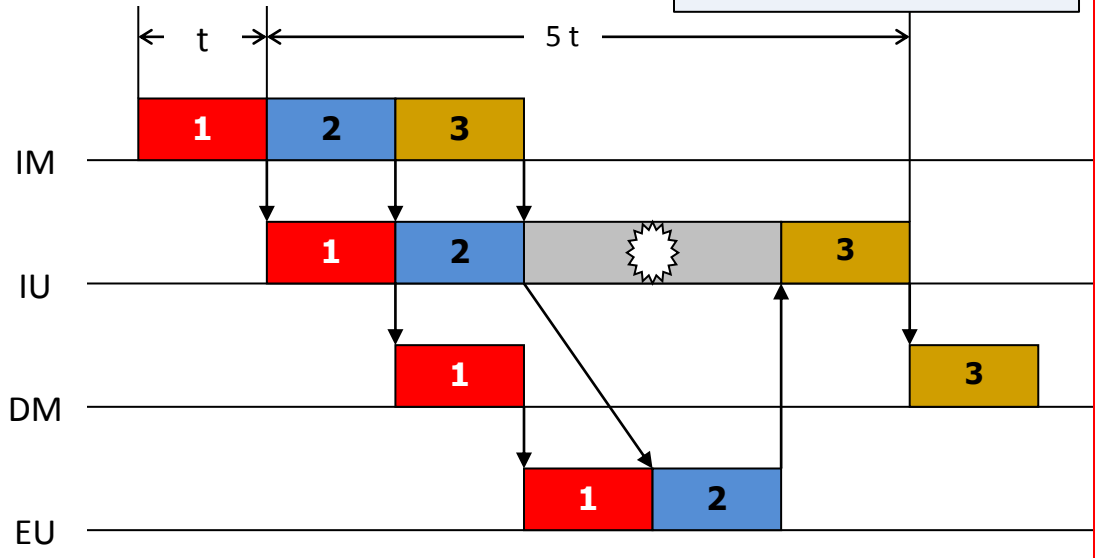
$$T = \frac{4}{3} \cdot t \qquad \varepsilon = \frac{T_{id}}{T} = \frac{3}{4}$$

Abstract architecture: *exactly* one time interval *t* is wasted (approximate evaluation of the concrete architecture)

Example 3: logical dependency

1.	LOAD	R1, R0, R4
2.	ADD	R4, R5, R6
3.	STORE	R2, R0, R6

Logical dependency, *induced by* instruction 2 *on* instruction 3; *distance* k =1



$$T = \frac{5}{3} \cdot t \qquad \varepsilon = \frac{3}{5}$$

Instruction 3: when processed by IU, it *needs the updated value of RG[6] contents*.

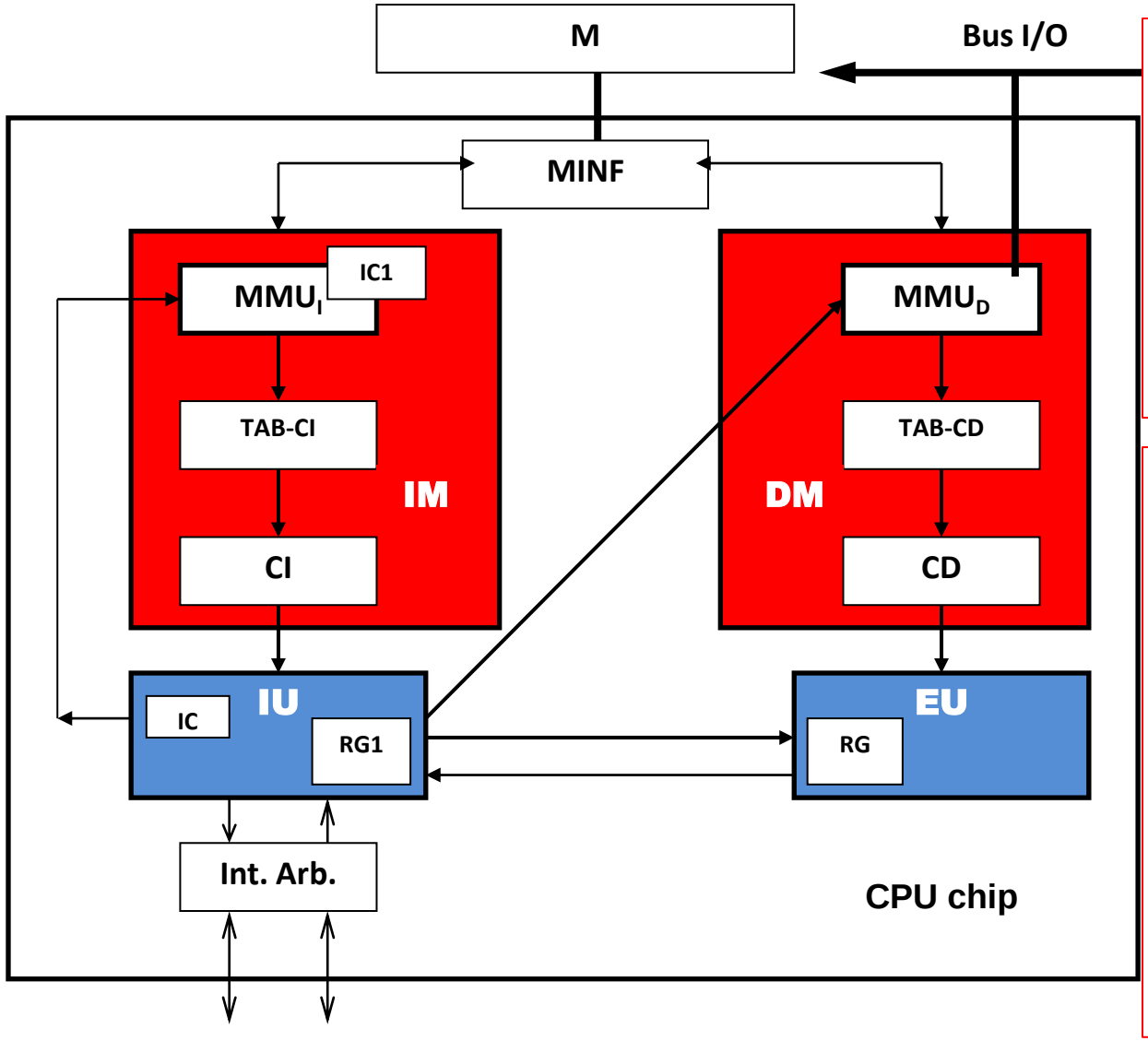
Thus, IU *must wait* for the completion of execution of instruction 2 in EU.

In this case, *two* time intervals *t* are wasted (approximate view of the abstract architecture).

The impact of *execution latency* is significant (in this case, DM is involved because of the LOAD instruction) .

If an arithmetic instruction were in place of LOAD: the bubble skinks to *one* time interval *t*.

Pipelined D-RISC CPU: implementation issues



Assume a **D-RISC** machine

Verify the feasibility of the abstract architecture characteristic:
all stages have the same service time = 2τ.

RG-RG1 consistency:
integer *semaphore* mechanism associated to RG1.

IC-IC1 consistency:
unique identifier associated to instructions from IM,
in order to accept or to discard them in IU.

RG1 consistency

- Each General Register RG1[i] has associated a non-negative integer semaphore S[i]
- For each arithmetic/LOAD instruction having RG[i] as destination, IU increments S[i]
- Each time RG1[i] is updated with a value sent from EU, IU decrements S[i]
- For each instruction that needs RG[i] to be read by IU (address components for LOAD, address components and source value for STORE, condition applied to registers for IF, address in register for GOTO), IU waits until condition (S[i] = 0) holds

IC1 consistency

- Each time IU executes a branch or jump, IC is updated with the new target address, and this value is sent to IM
- As soon as it is received by IU, IM writes the new value into IC1
- Every instruction sent to IU is *accompanied by* the IC1 value
- For each received instruction, *IU compares the value of IC with the received value of IC1*: the instruction is valid if they are equal, *otherwise the instruction is discarded.*

- Notice that IC (IC1) acts as a *unique identifier*. Alternatively, a shorter identifier can be generated by IU and sent to IM.

Detailed tasks of the CPU units

- **IM**
  - **MMU_I** : translates IC1 into physical address IA of instruction, sends (IA, IC1) to TAB-CI, and increments IC1
    - when a new message is received from IU, MMU_I updates IC1
  - **TAB-CI** : translates IA into cache address CIA of instruction, or generates a cache fault condition MISS), and sends (CIA, IC1, MISS) to CI
  - **CI** : if MISS is false, reads instruction INSTR at address CIA and sends (INSTR, IC1) to IU; otherwise performs the block transfer, then reads instruction INSTR at address CIA and sends (INSTR, IC1) to IU

Detailed tasks of the CPU units

- **IU**
  - when receiving *(regaddr, val)* from EU writes *val* into RG[regaddr] and decrements S[regaddr]

  or

  - when receiving (INSTR, IC1), if IC = IC1 the instruction is valid and IU goes on, otherwise discards the instruction
  - if type = arithmetic (COP, Ra, Rb, Rc), sends instruction to EU, increments S[c]
  - if type = LOAD (COP, Ra, Rb, Rc), waits for updated values of RG[a] and RG[b], sends request (read, RG[a] + RG[b]) to DM, increments S[c]
  - if type = STORE (COP, Ra, Rb, Rc), waits for updated values of RG[a], RG[b] and RG[c], sends request (read, RG[a] + RG[b], RG[c]) to DM
  - if type = IF (COP, Ra, Rb, OFFSET), waits for updated values of RG[a] and RG[b], if branch condition is true then IC = IC + OFFSET and sends the new IC to IM, otherwise IC = IC +1
  - if type = GOTO (COP, OFFSET), IC = IC + OFFSET and sends the new IC to IM
  - if type = GOTO (COP, Ra), waits for updated value of RG[a], IC = RG[a] and sends the new IC to IM

Detailed tasks of the CPU units

- **DM**
  - **MMU_D** : receives (op, log_addr, data) from IU, translates *log_addr* into *phys_addr*, sends (op, phys_addr, data) to TAB_CD
  - **TAB-CD** : translates *phys_addr* into cache address CDA, or generates a cache fault condition MISS), and sends (op, CDA, data, MISS) to CD
  - **CD** : if MISS is false, then according to *op*:
    - reads VAL at address CDA and sends VAL to EU;
    - otherwise writes *data* at address CDA;

    if MISS is true: performs block transfer; completes the requested operation as before
- **EU**
  - receives instruction (COP, Ra, Rb, Rc) from IU
  - if type = arithmetic (COP, Ra, Rb, Rc), executes RG[c] = COP(RG[a], RG[b]), sends the new RG[c] to IU
  - if type = LOAD (COP, Ra, Rb, Rc), waits VAL from DM, RG[c] = VAL, sends VAL to IU

Cost model

- To be evaluated on the abstract architecture: good approximation
- Branch degradation:

λ = branch probability

$$T = \lambda\ 2t + (1 - \lambda)\ t = (1 + \lambda)\ t$$

- Logical dependencies degradation:

$$T = (1 + \lambda)\ t + \Delta$$

Δ = average delay time in IU processing

- Efficiency:

$$\varepsilon = \frac{T_{id}}{T} = \frac{t}{(1+\lambda)\cdot t + \Delta} = \frac{1}{1 + \lambda + \frac{\Delta}{t}}$$

Example

1. LOAD R1, R0, R4
2. ADD R4, R5, R6
3. GOTO L
4. …

$\Delta = 0$

$\lambda = 1/3$

$$T = (1+\lambda)\cdot t = \frac{4}{3}\cdot t \qquad \varepsilon = \frac{3}{4}$$

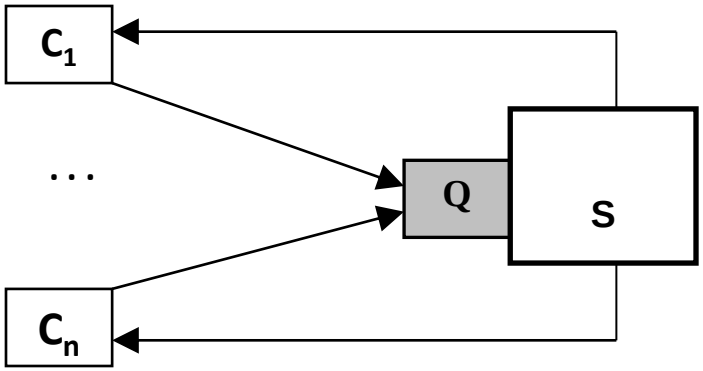
Logical dependencies degradation

- Formally, Δ is the response time R_Q in a **client-server system**
- *Server*: DM-EU subsystem
- A *logical queue of instructions* is formed at the input of DM-EU server
- *Clients*: *logical* instances of the IM-IU subsystem, corresponding to queued instructions sent to DM-EU
- Let d = average probability of logical dependencies

$$\Delta = d\, R_Q$$

- How to evaluate R_Q?

Client – server systems in general (at any level)



Assume a ***request-response behaviour***:

a client sends a request to the server,

then waits for the response from the server.

The service time of each client is delayed by the **RESPONSE TIME** of the server.

Modeling and evaluation as a Queueing System.

C_i::
initialize *x* …
while (true**) do**
{ < send *x* to S >;
< receive *y* from S >;
x = *G (y, ...)*
}

S::
while (true**) do**
{ < receive *x* from C $\in \{C_1, \ldots, C_n\}$ >;
y = *F (x, ...)* ;
< send *y* to *C* >
}

Client-server queueing modeling

$$\begin{cases} T_{cl} = T_G + R_Q \\ R_Q = R_Q(\rho, T_S, Ls, \sigma_S) \\ \rho = \dfrac{T_S}{T_A} \\ T_A = \dfrac{T_{cl}}{n} \end{cases}$$

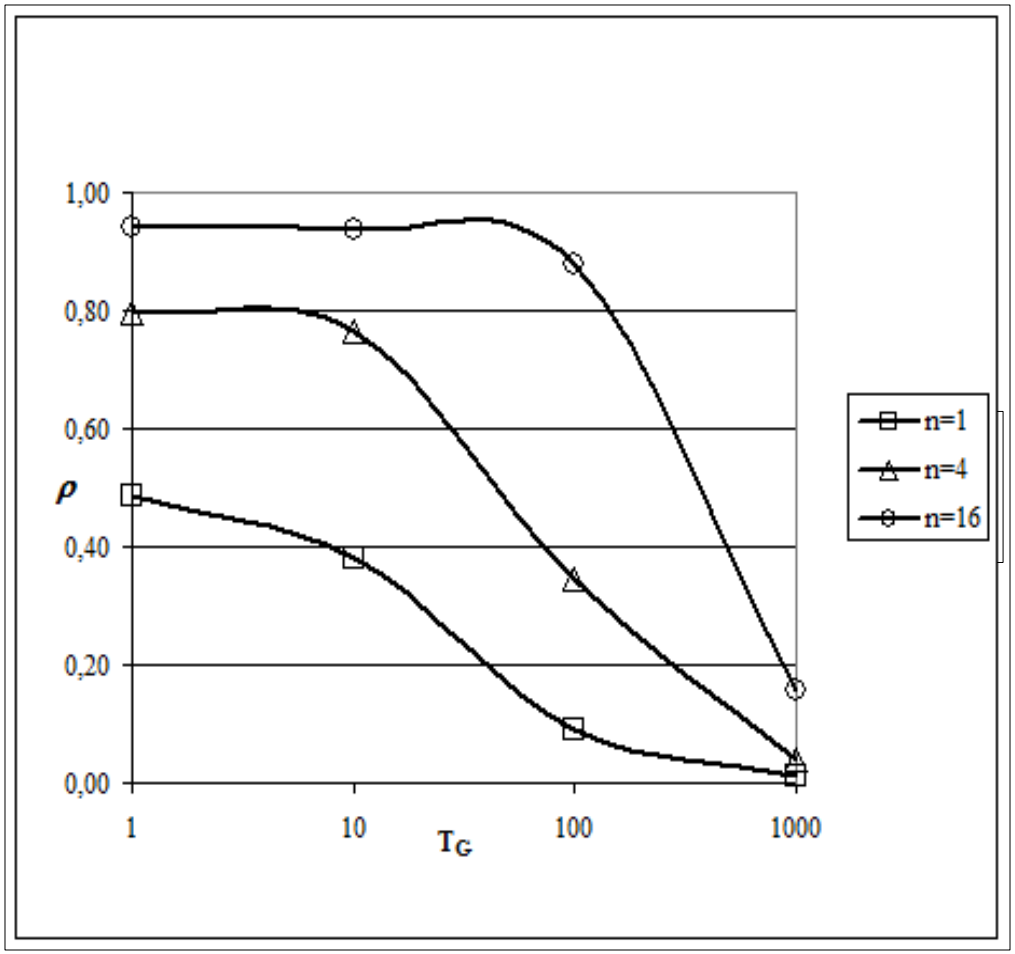
- T_{cl}: average service time of generic client
- T_G: ideal average service time of generic client
- R_Q: average response time of server
- ρ: server queue utilization factor
- T_S: average service time of server
- L_S: average latency time of server
- σ_S: other service time distribution parameters (e.g., variance of the service time of the server)
- T_A: average interarrival time to server
- n: number of clients (assumed identical)

System of equations to be solved through analytical or numerical methods.

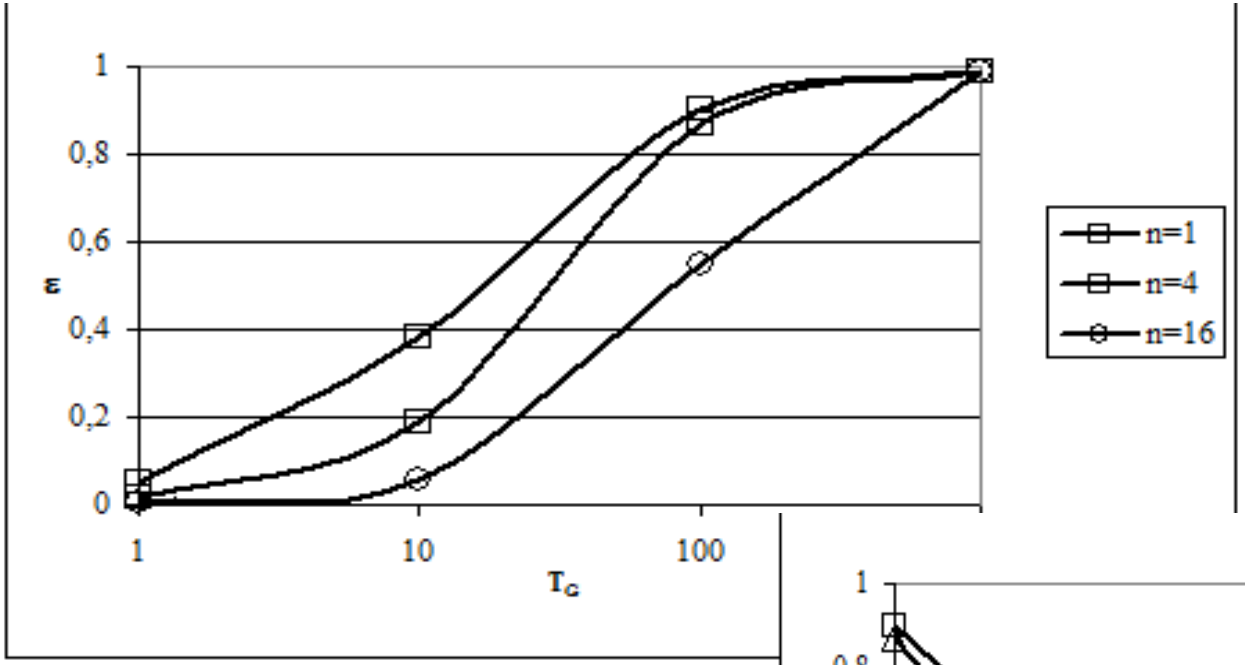
One and only one real solution exists satisfying the constraint: $\rho < 1$.

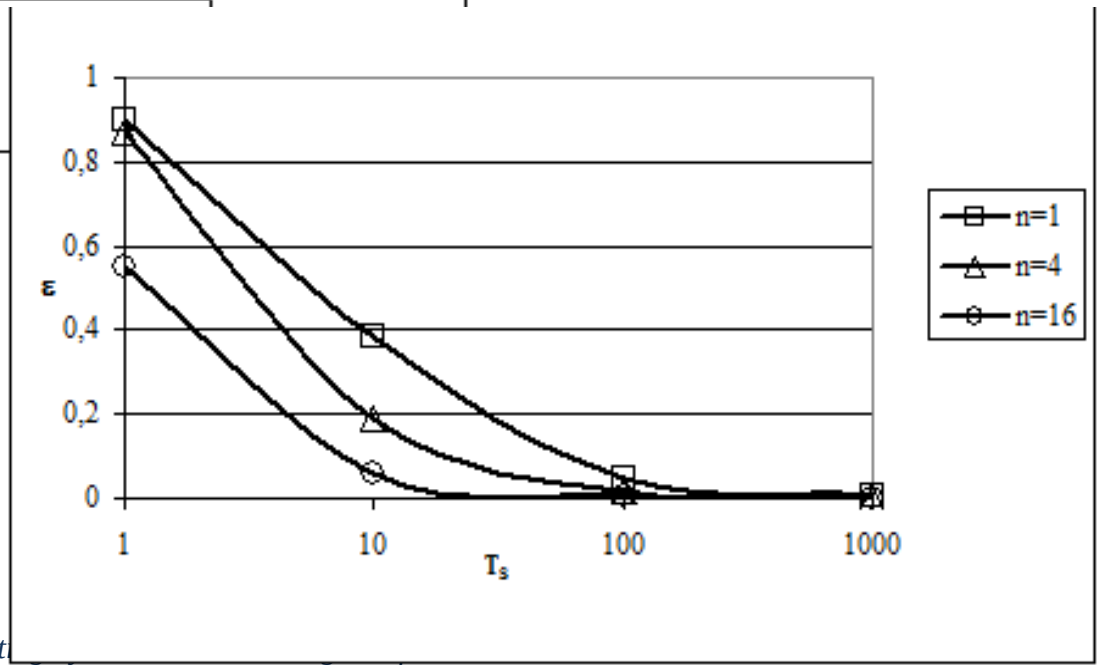
This methodology will be applied several times during the course.

Typical results: server utilization degree (= ρ)



Typical results: client efficiency (= T_G/T_{cl})





Parallel server: the impact of service time and of latency

- A more significant way to express the Response Time, especially for a parallel server:

$$R_Q = W_Q(\rho, \ldots) + L_S$$

- W_Q: average **Waiting Time** in queue (queue of clients requests): W_Q depends on the utilization factor ρ, thus on the **service time** only (*not* on latency)
- L_S is the **latency** of server
- Separation of the impact of service time and of latency
- *All the parallelism paradigms aim to reduce the service time*
- Some parallelism paradigms are able to reduce the latency too
  - Data-parallel, data-flow, …

  while other paradigms increase the latency
  - Pipeline, farm, …
- **Overall effect**: in which cases does the service time (latency) impact dominates on the latency (service time) impact ?
- For each server-parallelization problem, the relative impact of service time and latency must be carefully studied in order to find a satisfactory trade-off.

Pipeline CPU: approximate evaluation of Δ

- For a D-RISC machine
- Evaluation on the abstract architecture
- Let:
  - k = **distance** of a logical dependency
  - d_k = *probability* of a logical dependency with distance k
  - N_Q = *average number of instructions waiting in DM-EU subsystem*
- It can be shown that:

$$\Delta \approx t \cdot \sum_{k=1}^{\bar{k}} d_k \cdot \left(N_{Q_k} + 1 - k \right)$$

where summation is extended to non-negative terms only.

$N_Q = 2$ if the instruction inducing the logical dependency is the last instruction of a sequence containing all *EU-executable instructions* and *at least one LOAD*, otherwise $N_Q = 1$.

Δ formula for D-RISC: proof

- $\Delta = d_1\Delta_1 + d_2\Delta_2 + \ldots$ *extended to non-negative terms only*
- d_k = probability of logical dependency with distance *k*
- $\Delta_k = R_{Qk} = W_{Qk} + L_S$
- W_{Qk} = average number of queued instructions that (with probability d_k) precede the instruction inducing the logical dependency
- *L_S = Latency of EU in D-RISC; with hardware-implemented arithmetic operations on integers $L_S = t$*
- $\Delta_k = W_{Qk} + t$

- ***Problem: find a general expression for W_{Qk}***
- The number of *distinct* situations that can occur in a D-RISC *abstract* architecture is very limited
- Thus, the proof can be done *by enumeration*.
- Let us individuate the distinct situations.

Δ formula for D-RISC: proof

Case A: 1. ADD …, …, R1 2. STORE …, …, R1	distance **k = 1** probability $\mathbf{d_1}$	$\mathbf{W_Q = 0}$: when **2** is in IU, **1** is in EU (current service)	$\Delta = d_1 (W_Q + L_s) = d_1\, t$
Case B: 1. ADD …, …, R1 2. SUB …, …, R2 3. STORE …, …, R1	distance **k = 2** probability $\mathbf{d_2}$	$\mathbf{W_Q = -1t}$: when **3** is in IU, **1** has been already executed	$\Delta = d_2 (W_Q + L_s) = 0$
Case C: *1. LOAD …, …, …* 2. ADD …, …, R1 3. STORE …, …, R1	distance **k = 1** probability $\mathbf{d_1}$	$\mathbf{W_Q = 1t}$: when **3** is in IU, **1** is in EU (current service), **2** is **waiting in queue**	$\Delta = d_1 (W_Q + L_s) = 2\, d_1\, t$
Case D: *1. LOAD …, …, …* 2. ADD …, …, *R1* 3. SUB …, …, R2 4. STORE …, …, R1	distance **k = 2** probability $\mathbf{d_2}$	$\mathbf{W_Q = 0}$: when **4** is in IU, **2** is in EU (current service)	$\Delta = d_2 (W_Q + L_s) = d_2\, t$

In all other cases $\Delta = 0$, in particular for $k > 2$.

Δ formula for D-RISC: proof

- W_{Qk} = number of queued instructions preceding the instruction that induces the logical dependency
- Maximum value of W_{Qk} = 1t:
  - the instruction currently executed in EU has been *delayed by a previous LOAD, and* the instruction inducing the logical dependency is *still in queue*
- Otherwise: W_{Qk} = 0 or W_{Qk} = - 1t
  - W_{Qk} = 0: the instruction currently executed in EU induces the logical dependency,
  - W_{Qk} = 0: the instruction currently executed in EU has been *delayed by a previous LOAD, and it is* the instruction inducing the logical dependency
  - W_{Qk} = -1t: the instruction inducing the logical dependency has been already executed
- To be proven that: $W_{Qk} = (N_{Qk} - k)t$
  - $N_{Qk} = 2$ if the instruction inducing the logical dependency is the last instruction of a sequence containing all *EU-executable instructions* and *at least one LOAD*,
  - otherwise $N_{Qk} = 1$.

Δ formula for D-RISC: proof

Case A: 1. ADD …, …, R1 2. STORE …, …, R1	distance **k = 1** probability $\mathbf{d_1}$	$\mathbf{W_Q = 0}$: when **2** is in IU, **1** is in EU (current service)	$\Delta = \mathbf{d_1}(W_Q + L_s) = \mathbf{d_1}\, t$ $\mathbf{N_Q = 1}$, $W_Q = (N_Q - k)t = 0$
Case B: 1. ADD …, …, R1 2. SUB …, …, R2 3. STORE …, …, R1	distance **k = 2** probability $\mathbf{d_2}$	$\mathbf{W_Q = -1t}$: when **3** is in IU, **1** has been already executed	$\Delta = \mathbf{d_2}(W_Q + L_s) = 0$ $\mathbf{N_Q = 1}$, $W_Q = (N_Q - k)t = -1t$
Case C: *1. LOAD …, …, …* 2. ADD …, …, R1 3. STORE …, …, R1	distance **k = 1** probability $\mathbf{d_1}$	$\mathbf{W_Q = 1t}$: when **3** is in IU, **1** is in EU (current service), **2** is **waiting in queue**	$\Delta = \mathbf{d_1}(W_Q + L_s) = 2\,\mathbf{d_1}\, t$ $\mathbf{N_Q = 2}$, $W_Q = (N_Q - k)t = 1t$
Case D: *1. LOAD …, …, …* *2. ADD …, …, R1* 3. SUB …, …, R2 4. STORE …, …, R1	distance **k = 2** probability $\mathbf{d_2}$	$\mathbf{W_Q = 0}$: when **4** is in IU, **2** is in EU (current service)	$\Delta = \mathbf{d_2}(W_Q + L_s) = \mathbf{d_2}\, t$ $\mathbf{N_Q = 2}$, $W_Q = (N_Q - k)t = 0$

In all other cases $\Delta = \mathbf{0}$: $k > 2$, or $k = 2$ and $N_Q = 1$

Example

1. LOAD R1, R0, R4
2. ADD R4, R5, R6
3. STORE R2, R0, R6

$\lambda = 0$

One logical dependency induced by instruction 2 on instruction 3:

distance k = 1, probability $d_1 = 1/3$, $\mathbf{N_Q = 2}$

$$\Delta = d_1 \cdot N_Q \cdot t = \frac{2}{3} t \cdot$$

$$T = (1 + \lambda) \cdot t + \Delta = \frac{5}{3} \cdot t \qquad \varepsilon = \frac{3}{5}$$

Example

```
1.      L :     LOAD    R1, R0, R4
2.              LOAD    R2, R0, R5
3.              ADD     R4, R5, R4
4.              STORE   R3, R0, R4
5.              INCR    R0
6.              IF <    R0, R6, L
7.              END
```

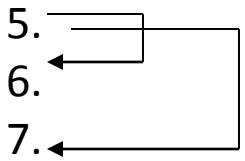
$$\lambda = \frac{1}{6} \quad , \quad d_1 = \frac{2}{6} \quad , \quad N_Q = \frac{3}{2}$$

Two logical dependencies with the same distance (= 1): the one with $N_Q = 2$, the other with $N_Q = 1$

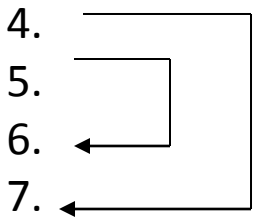
$$\Delta = d_1 \cdot N_Q \cdot t = \frac{1}{2} \cdot$$

$$T = (1 + \lambda) \cdot t + \Delta = \frac{5}{3} \cdot t \qquad \varepsilon = \frac{3}{5}$$

Note: intermixed/nested logical dependencies



Logical dependency induced by 5 on 7 has **no effect**.



Logical dependency induced by 4 on 7 has **no effect**.

Cache faults

- All the performance evaluations must be corrected taking into account also the effects of cache faults (Instruction Cache, Data Cache) on completion time.

- The added penalty for block transfer (***data***)
  - Worst-case: *exactly the same penalty* studied for sequential CPUs (See Cache Prerequisites),
  - Best-case: no penalty - *fully overlapped* to instruction pipelining

Compiler optimizations

- Minimize the impact of branch degradations:
  - **DELAYED BRANCH**

  = try to fill the branch bubbles with useful instructions

  ➢ *Annotated instructions*

- Minimize the impact of logical dependencies degradations:
  - **CODE MOTION**

  = try to increase the distance of logical dependencies

- Preserve program semantics, i.e. transformations that satisfy the **Bernstein conditions** for correct parallelization.

Bernstein conditions

Given the sequential computation:

$$R_1 = f_1(D_1) \; ; \; R_2 = f_2(D_2)$$

it can be transformed into the **equivalent** parallel computation:

$$R_1 = f_1(D_1) \; \| \; R_2 = f_2(D_2)$$

if all the following conditions hold:

$$\begin{cases} R_1 \cap D_2 = \varnothing & \textit{(true dependency)} \\ R_1 \cap R_2 = \varnothing & \textit{(antidependency)} \\ D_1 \cap R_2 = \varnothing & \textit{(output dependency)} \end{cases}$$

Notice: for the parallelism inside a *microinstruction* (see Firmware Prerequisites), only the first and second conditions are sufficient (*synchronous* model of computation). E.g.

$A + B \rightarrow C \; ; \; D + E \rightarrow A$ is equivalent to: $A + B \rightarrow C \; , \; D + E \rightarrow A$

Delayed branch: example

```
1.        L :     LOAD    R1, R0, R4
2.                LOAD    R2, R0, R5
3.                ADD     R4, R5, R4
4.                STORE   R3, R0, R4
5.                INCR    R0
6.                IF <    R0, R6, L
7.                END
```

$\lambda = 0$

$$T = (1+\lambda)\cdot t + \Delta = \frac{3}{2}\cdot t \qquad \varepsilon = \frac{2}{3}$$

(previous version: $\varepsilon = 3/5$)

```
1.                LOAD    R1, R0, R4
2.        L :     LOAD    R2, R0, R5
3.                ADD     R4, R5, R4
4.                STORE   R3, R0, R4
5.                INCR    R0
6.                IF <    R0, R6, L, delayed_branch
7.                LOAD    R1, R0, R4
8.                END
```

Increase logical dependency distance: example

```
1.          LOAD    R1, R0, R4
2.   L :    LOAD    R2, R0, R5
3.          ADD     R4, R5, R4
4.          STORE   R3, R0, R4
5.          INCR    R0
6.          IF <    R0, R6, L, delayed_branch
7.          LOAD    R1, R0, R4
8.          END
```

For this transformation, the compiler initializes register R3 at the value

Base_C - 1

where Base_C is the base address of the result array.

$d_1 = 1/6$

$$T = t + d_k\ t\left(N_{Q-medio} + 1 - k\right) = \frac{8}{6}\ t$$

$$\varepsilon = \frac{3}{4}$$

(previous version: $\varepsilon = 2/3$)

```
1.          LOAD    R1, R0, R4
2.   L :    LOAD    R2, R0, R5
3.          ADD     R4, R5, R4
4.          INCR    R0
5.          STORE   R3, R0, R4
6.          IF <    R0, R6, L, delayed_branch
7.          LOAD    R1, R0, R4
8.          END
```

$T_c = 6\ N\ T = 8\ N\ t$

Exploit both optimizations jointly

```
1.          LOAD    R1, R0, R4
2.   L :    LOAD    R2, R0, R5
3.          ADD     R4, R5, R4
4.          STORE   R3, R0, R4
5.          INCR    R0
6.          IF <    R0, R6, L, delayed_branch
8.          END
```

$$T = t + d_k\ t\left(N_{Q-medio} + 1 - k\right) = \frac{7}{6}\ t$$

$$\varepsilon = \frac{6}{7}$$

(previous version: $\varepsilon = 3/4$)

```
1.   L :    LOAD    R1, R0, R4
2.          LOAD    R2, R0, R5
3.          INCR    R0
4.          ADD     R4, R5, R4
6.          IF <    R0, R6, L, delayed_branch
6.          STORE   R3, R0, R4
7.          END
```

$T_c = 6\ N\ T = 7\ N\ t$

Exercizes

1. Compile, with optimizations, and evaluate the completion time of a program that computes the matrix-vector product (operands: int A[M][M], int B[M]; result: int C[M], where $M = 10^4$) on a D-RISC Pipeline machine.

   The evaluation must include the cache fault effects, assuming the Data Cache of 64K words capacity, 8 words blocks, operating on-demand.

2. Consider the curves at pages 33, 34. Explain their shapes in a qualitative way (e.g. why ε tends asymtotically to an upper bound value as T_G tends to infinity, …).

Exercize

3. Systems S1 and S2 have a Pipelined CPU architecture. S1 is a D-RISC machine. The assembler level of S2 is D-RISC enriched with the following instruction:

$$REDUCE\_SUM \quad RA, RN, Rx$$

with RG[RA] = base address of an integer array A[N], and RG[RN] = N.

The semantics of this instruction is

$$RG[Rx] = reduce\ (A[N], +)$$

Remember that

$$reduce\ (H[N], \otimes)$$

where $\otimes$ is any associative operator, is a second-order function whose result is equal to the scalar value

$$H[0] \otimes H[1] \otimes \ldots \otimes H[N-1]$$

a) Explain how a *reduce* (A[N], *+)* is compiled on S1 and how it is implemented on S2, and evaluate the difference *d* in the completion times of *reduce* on S1 and on S2.

b) A certain program includes *reduce* (A[N], *+)* executed one and only one time. Explain why the difference of the completion times of this program on S1 and on S2 could be different compared to *d* as evaluated in point a).

Execution Unit and “long” instructions

- The EU implementation of “long” arithmetic operations, e.g.
  - integer multiplication, division
  - floating point addition
  - floating point multiplication, division
  - other floating point operations (square root, sin, cosin, …)

  has to respect some basic guidelines:

  A. to mantain the EU **service time** equal to the CPU ideal service time T_{id},
  B. to minimize the EU **latency** in order to minimize the logical dependencies degradation.

- To deal with issues A, B we can apply:

  A. Parallel paradigms at the firmware level
  B. Data-parallel paradigms and special hardware implementations of arithmetic operations.

Integer multiplication, division

- A simple *firmware* implementation is a loop algorithm (similar to the familiar operation "by hand"), exploting additions/subtractions and shifts
  - Latency = **O(**word lenght) $\tau \sim 50\ \tau$ (32 bit integers)
- *Hardware* implementations in 1-2 τ exist, though expensive in terms of silicon area if applied to the entire word
- *Intermediate* solution: firmware loop algorithm, in which few iterations are applied to parts of the word (e.g. byte), each iteration implemented in hardware.
- Intermediate solutions exploiting **parallel paradigms**:
  - **farm** of firmware multiply-&-divide units: reduced service time, but not latency
  - **loop-unfolding pipeline**, with stages corresponding to parts (byte) implemented in **hardware**: reduced service time, latency as in the sequential version (intermediate).

Multiplication through loop-unfolding pipeline

Multiply *a*, *b*, result *c*

Let *byte (v, j)* denote the *j*-th byte of integer word *v*

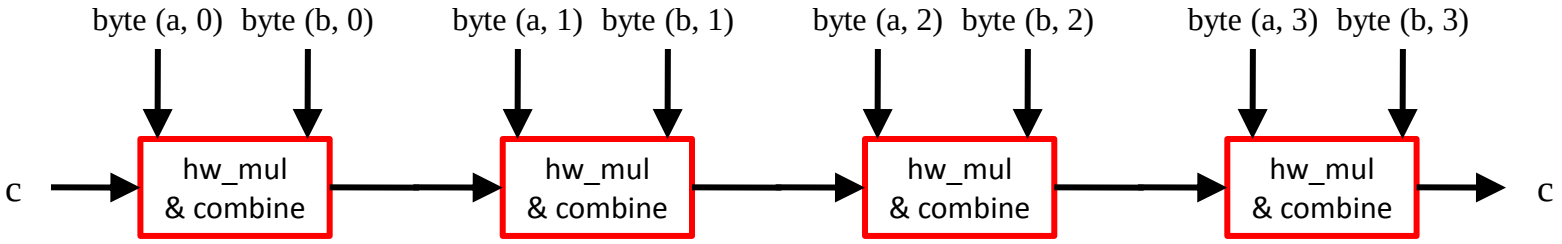
Sequential multiplication algorithm (32 bit) exploiting the hardware implementation applied to bytes:

init c;

for (i = 0; i < 4; i++)

c = *combine* (c, *hardware_multiplication* (*byte* (a, i), *byte* (b, i)))

Loop-unfolding pipeline (**"systolic"**) implementation:



service time = 2τ, latency = 8τ

Exercize

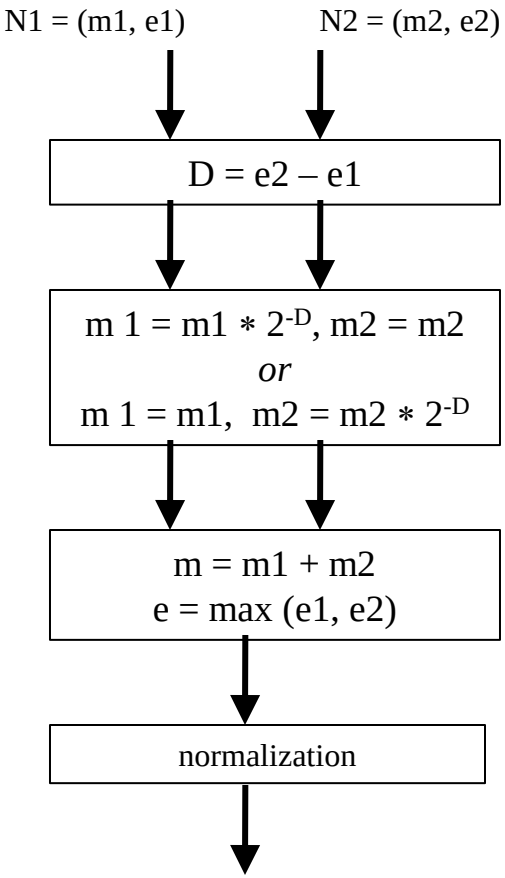
4. The cost model formula for Δ (given previously) is valid for a D-RISC-like CPU in which the integer multiplication/division operations are implemented entirely in hardware (1- 2 clock cycles).

   Modify this formula for a Pipelined CPU architecture in which the integer multiplication/division operations are implemened as a 4-stage pipeline.

Pipelined floating point unit

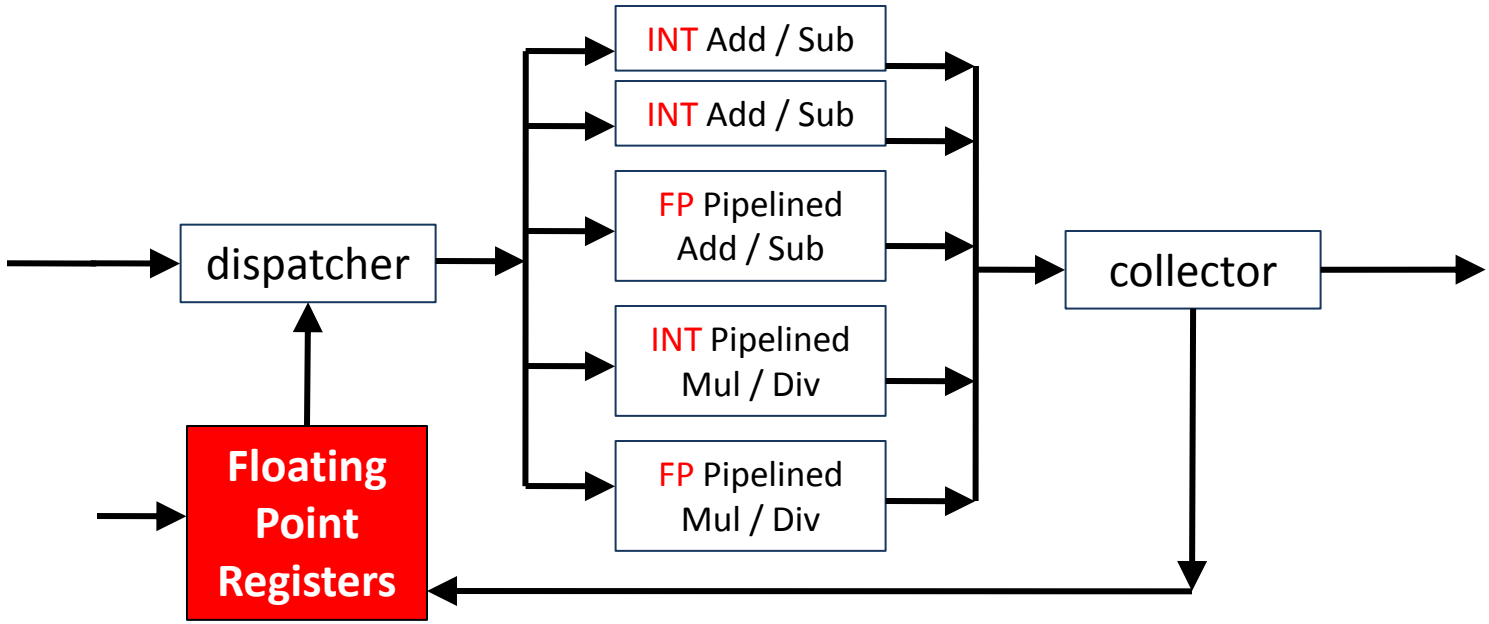
Example: FP addition

FP number = (m, e), where: m = mantissa, e = exponent



Parallel EU schemes

- **Farm**, where each worker is general-purpose, implementing addition/subtraction & multiply/divide on integer or floating point numbers.
- **Functional partitioning**, where each worker implements a specific operation on integer or floating point numbers:



Parallel EU: additional logical dependencies

- More arithmetic instructions can be executed simultaneoulsy in a parallel EU.
- This introduces additional logical dependencies "**EU-EU**" (till now: "IU-EU" only). For example, in the sequence:

```
MUL   Ra, Rb, Rc
ADD   Rd, Re, Rf
```

  the second instruction starts in EU as soon the first enters the pipelined multiplier, while in the sequence:

```
MUL  Ra, Rb, Rc
ADD   Rc, Re, Rf
```

  the second instruction is blocked in EU until the first is completed.

  The *dispatcher* units (see previous slide) is in charge of implementing the synchronization mechanism, according to a semaphoric technique similar to the one for synchronizing IU and EU.

- Because of EU latency and EU-EU dependencies, it is more convenient to increase the asynchrony degree of the IU-EU channel, i.e. a Queueing Unit is inserted. In practice, this increases the number of pipeline stages of the machine.

Data parallelism for arithmetic operations

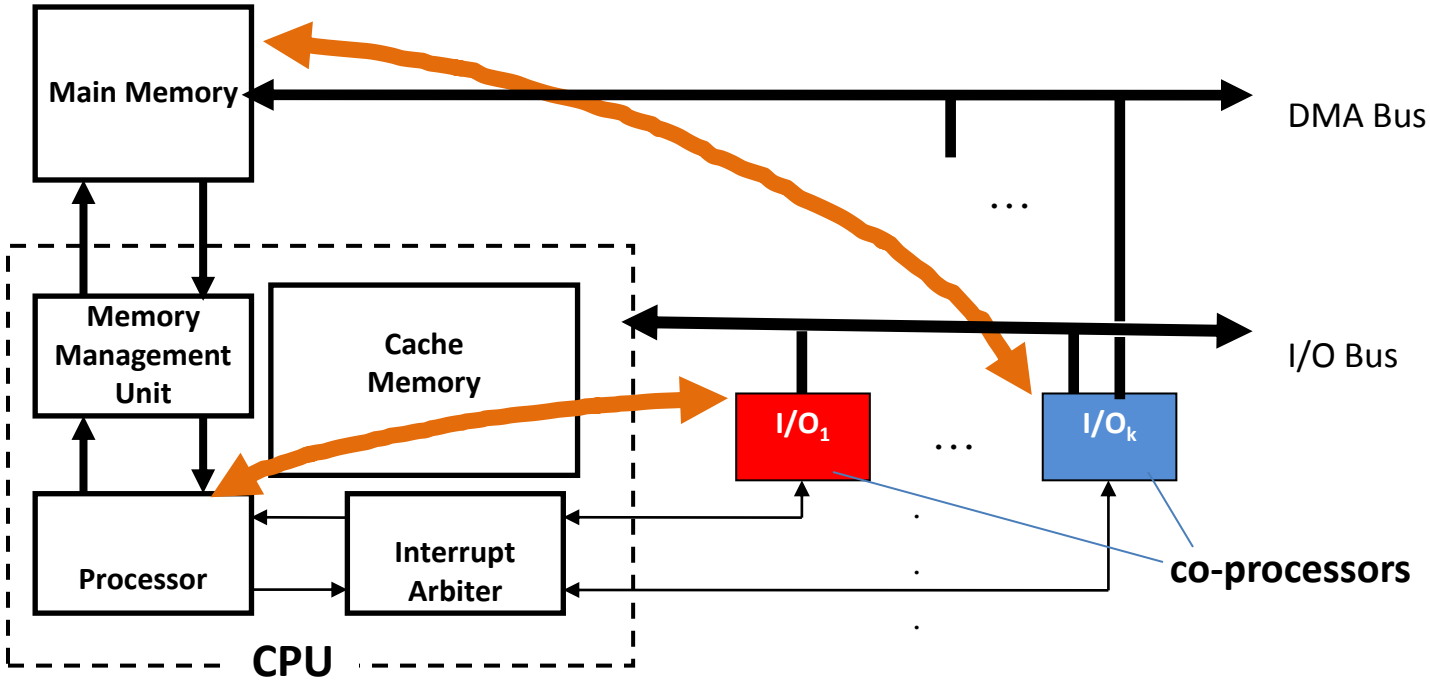
- Data parallelism is meaningful on *large data structures*, i.e. *arrays*, only
- **Vectorized** assembler instructions operate on arrays, e.g.

  SUM_VECT base_address A, base_address B, base_address C, size N

- *Data parallel implementation*: ***map*** at the firmware level
  - *scatter* arrays A, B
  - *map* execution
  - *gather* to obtain array C
- Also: ***reduce***, ***parallel prefix*** operations
- Also: operations requiring ***stencil-based*** data parallel implementation, e.g. matrix-vector or matrix-matrix product, convolution, filters, Fast Fourier Transform.

Pure Risc enriched by floating point pipelined and vectorized co-processors

- **Mathematical co-processors**, connected as *I/O units*
  - *instead of assembler instructions*
- FP / vector operations called as libraries containing I/O transfer operations (LOAD, STORE - Memory Mapped I/O - DMA)



Solutions of Exercizes

- Exercize 1
- Exercize 3
- Exercize 4

Exercize 1

Compile, with optimizations, and evaluate the completion time of a program that computes the matrix-vector product (operands: int A[M][M], int B[M]; result: int C[M], where $M = 10^4$) on a D-RISC Pipeline machine.

The evaluation must include the cache fault effects, assuming the Data Cache of 64K words capacity, 8 words blocks, operating on-demand.

Solution of Exercize 1

Sequential algorithm, $O(M^2)$:

```
int A[M][M]; int B[M]; int C[M];
    for (i = 0; i < M; i++)
      {C[i] = 0;
       for (j = 0; j < M; j++)
                C[i] = C[i] + A[i][j] * B[j]
      }
```

Process virtual memory:

- matrix A stored *by row*, M^2 consecutive words;
- compilation rule for 2-dimension matrices used in loops: for each iteration,

$$base\ address = base\ address + M$$

Solution of Exercize 1

Basic compilation (without optimizations):

initialize RA, RB, RC, RM, Ri (=0); allocate Rj, Ra, Rb, Rc

```
LOOP_i:     CLEAR   Rc
            CLEAR   Rj

LOOP_j:     LOAD    RA, Rj, Ra
            LOAD    RB, Rj, Rb
            MUL     Ra, Rb, Ra
            ADD     Rc, Ra, Rc
            INCR    Rj
            IF <    Rj, RM, LOOP_j

            STORE   RC, Ri, Rc
            ADD     RA, RM, RA
            INCR    Ri
            IF <    Ri, RM, LOOP_i

            END
```

Completion time:

$T_c \sim M^2 *$ completion time of innermost loop

$$T_c \sim 6\, M^2\, T$$

where T = service time per instruction

Performance degradations in the outermost loop are not relevant (however, the compiler applies optimizations to this part too).

Solution of Exercize 1

Analysis of innermost loop (initially: perfect cache):

```
LOOP_j:   LOAD   RA, Rj, Ra
          LOAD   RB, Rj, Rb
          MUL    Ra, Rb, Ra
          ADD    Rc, Ra, Rc
          INCR   Rj
          IF <   Rj, RM, LOOP_j
```

$\lambda = 1/6$; $k = 1$, $d_k = 1/6$, $N_{Qk} = 2$

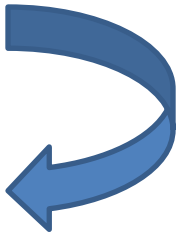
$T = (1 + \lambda)\, t + d_k\, t\, (N_{Qk} + 1 - k) = 9t/6$ $\qquad \varepsilon = 6/9$

$T_c \sim 6\, M^2\, T = 9\, M^2\, t = 18\, M^2\, \tau$

Solution of Exercize 1

Optimization and evaluation of innermost loop (perfect cache):

```
                LOAD   RA, Rj, Ra
LOOP_j:         LOAD   RB, Rj, Rb
                INCR   Rj
                MUL    Ra, Rb, Ra
                ADD    Rc, Ra, Rc
                IF <   Rj, RM, LOOP_j, delayed_branch
                LOAD   RA, Rj, Ra
```



$\lambda = 0; k = 3 \Rightarrow \Delta = 0$

$T = t = T_{id}$ $\varepsilon = 1$

$T_c \sim 6\ M^2\ T = 6\ M^2\ t = 12\ M^2\ \tau$

Completion time difference = $6\ M^2\ \tau$, ratio = 1.5

Solution of Exercize 1

Impact of caching:

Instruction faults: not relevant; reuse ; 2 fixed blocks in working set

Matrix A, array C: locality only ; 1 block at the time for A and for C in working set

$$N_{fault\text{-}A} = M^2/\sigma,\ N_{fault\text{-}C} = M/\sigma$$

Array B: reuse (+ locality) ; all B blocks in working set; due to its size, B can be allocated entirely in cache

$$N_{fault\text{-}B} = M/\sigma$$

LOAD RB, Rj, Rb, **no_dellocation**

Assuming a secondary cache on chip: $T_{block} = \sigma\ \tau$

$$T_{fault} = N_{fault} * T_{block} \sim N_{fault\text{-}A} * T_{block} = M^2\ \tau$$

Worst–case assumption – *pipeline is blocked when a block transfer occurs* :

$$T_c = T_{c\text{-}perfect\text{-}cache} + T_{fault} = 12\ M^2\ \tau + M^2\ \tau = 13\ M^2\ \tau \qquad \varepsilon_{cache} = 12/13$$

In this case, if prefetching were applicable, $N_{fault\text{-}A} = 0$ and $T_C = T_{c\text{-}perfect\text{-}cache}$, $\varepsilon_{cache} = 1$.

LOAD RA, Rj, Ra, **prefetching**

Solution of Exercize 1

Best-case assumption about block transfer corresponds a more realistic behaviuor : pipeline is working normally during a block transfer; only if a data dependency on the block value occurs, then a bubble in naturally introduced.

In our case:

```
            LOAD   RA, Rj, Ra
LOOP_j:     LOAD   RB, Rj, Rb
            INCR   Rj
            MUL    Ra, Rb, Ra
            ADD    Rc, Ra, Rc
            IF <   Rj, RM, LOOP_j, delayed_branch
            LOAD   RA, Rj, Ra
```

During an A-block transfer,

- *DM serves other requests in parallel* (B is in cache),
- LOAD RB, … can be correctly executed **out-of-order**
- INCR Rj can be executed,
- MUL can start; EU waits for the element of A

With secondary cache on chip, and $\sigma = 8$, with these assumptions the block transfer is overlapped to pipeline behaviour:

- when EU is ready the execute MUL, the A-block transfer has been completed in DM,
- *no (relevant) cache degradation, even without relaying on prefetching of A blocks.*

For out-of-order behaviour: see the slides after Branch Prediction.

Exercize 3

Systems S1 and S2 have a Pipelined CPU architecture. S1 is a D-RISC machine. The assembler level of S2 is D-RISC enriched with the following instruction:

REDUCE_SUM RA, RN, Rx

with RG[RA] = base address of an integer array A[N], and RG[RN] = N.

The semantics of this instruction is

RG[Rx] = reduce (A[N], *+)*

Remember that

reduce (H[N], ⊗)

where ⊗ is any associative operator, is a second-order function whose result is equal to the scalar value

H[0] ⊗ H[1] ⊗ … ⊗ H[N-1]

a) Explain how a *reduce* (A[N], *+)* is compiled on S1 and how it is implemented on S2, and evaluate the difference *d* in the completion times of *reduce* on S1 and on S2.

b) A certain program includes *reduce* (A[N], *+)* executed one and only one time. Explain why the difference of the completion times of this program on S1 and on S2 could be different compared to *d* as evaluated in point a).

Solution of Exercize 3

a) System S1. The sequential algorithm for

$$x = reduce\ (\mathrm{A[N]},\ +)$$

having complexity *O(N),* can be compiled into the following optimized code for :

```
        LOAD   RA, Ri, Ra
LOOP:   INCR   Ri
        ADD    Rx, Ra, Rx
        IF <   Ri, RN, LOOP, delayed_branch
        LOAD   RA, Ri, Ra
```

Performance analysis, without considering possible cache degradations:

$$\lambda = 0;\ k = 2,\ d_k = 1/4,\ N_{Qk} = 2$$

$$T = (1 + \lambda)\ t + d_k\ t\ (N_{Qk} + 1 - k) = 5t/4 \qquad \varepsilon = 4/5$$

$$T_{c1} \sim 4\ N\ T = 5\ N\ t = 10\ N\ \tau$$

Solution of Exercize 3

System S2. *The firmware interpreter* of instruction *REDUCE_SUM RA, RN, Rx on a pipeline CPU* can be the following:

IM microprogram: the same of S1.

IU microprogram for REDUCE: send instruction encoding to EU, produce a stream of N read-requests to DM with consecutive addresses beginning at RG[RA].

DM microprogram: the same of S1.

EU microprogram for REDUCE: initialize RG[Rx] at zero; loop for N times: wait data *d* from DM, RG[Rx] + d $\rightarrow$ RG[Rx]; on completion, send RG[Rx] content to IU.

Alternatively: IU send to DM just one request (multiple_read, N, base address RG[RA]), and DM produces a stream of consecutive-address data to EU (there is no advantage).

Solution of Exercize 3

Considering the graphic simulation of this S2 interpreter, we obtain the latency of REDUCE instruction on S2, which is the completion time of a S2 program consisting of REDUCE only (see the general formula for the completion time of the pipeline paradigm):

$$T_{c2} \sim N\, t = 2\, N\, \tau$$

Notice that *there are no logical dependency nor branch degradations.*

The completion time ratio of the two implementations on S1 and S2 is equal to 5, and the difference

$$d = 8\, N\, \tau$$

This example confirms the analysis of the comparison of **assembler** *vs* **firmware implementation of the same functionality** (see Exercize in prerequisites): the difference in latency is mainly due to the *parallelism in microinstructions* wrt sequential execution at the assembler level.

In the example, this parallelism is relevant in the IU and EU behaviour.

Moreover, the degradations are often reduced, because some computations occurs locally in some units (EU in our case) instead of requiring the interactions of distinct units with consistency synchronization problems (in our case, only the final result is communicated from EU to IU).

Solution of Exercize 3

b) Consider a S1 program containing the REDUCE routine and executing it just one time. The equivalent S2 program contains the REDUCE instruction in place of the REDUCE routine.

The differenze of completion times is *less or equal than the d value* of part a):

- In S1, the compiler has more degrees of freedom to introduce optimizations concerning the code sections before and after the REDUCE routine:
  - the REDUCE loop can be, at least partially, unfolded,
  - the compiler can try to introduce some code motions, in order *to mix* some instructions of the code sections before and after the REDUCE routine with the instructions of the REDUCE routine itself.

This example shows **the potentials for compiler optimizations of RISC systems compared to CISC**: in some cases, such optimizations are able to compensate the RISC latency disadvantage, at least partially.

Exercize 4

The cost model formula for Δ (given previously) is valid for a D-RISC-like CPU in which the integer multiplication/division operations are implemented entirely in hardware (1- 2 clock cycles).

Modify this formula for a Pipelined CPU architecture in which the integer multiplication/division operations are implemened as a 4-stage pipeline.

Solution of Exercize 4

Cost model for pipelined D-RISC CPU:

$$\Delta \approx t \cdot \sum_{k=1}^{\bar{k}} d_k \cdot \left(N_{Q_k} + 1 - k \right)$$

where

- $N_Q = 2$ if the instruction inducing the logical dependency is the last instruction of a sequence containing all *EU-executable instructions* and *at least one LOAD*,

  otherwise $N_Q = 1$
- summation is extended to non-negative terms only. In D-RISC, for $k \geq 3$, $\Delta = 0$.

The meaning is that each Δ_k can be expressed as

$$\Delta_k = d_k \, R_{Qk} \qquad R_{Qk} = W_{Qk} + L_S$$

$W_{Qk} = N_{Qk} - k$ = average number of queued instructions that (with probability d_k) precede the instruction inducing the logical dependency

L_S = Latency of EU. With hardware-implemented arithmetic operations on integers:

$$L_S = t$$

Solution of Exercize 4

A 4-stage pipelined MUL/DIV operator has a latency:

$$L_{s\text{-}MUL/DIV} = 4\ t$$

If *p = prob(istruction is MUL or DIV)*, and all the other arithmetic operations are "short", the average EU latency is:

$$L_s = (1 + 3p)\ t$$

Then (the proof can be verified using the graphical simulation):

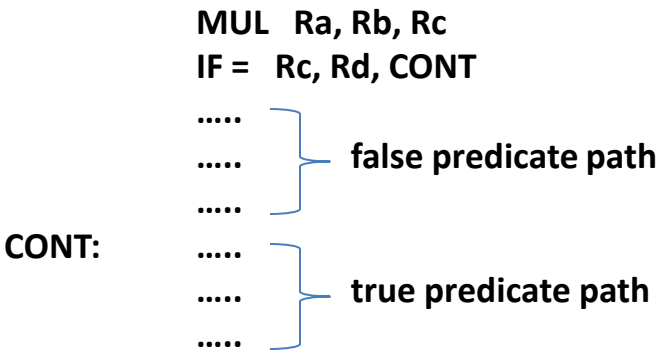
$$\Delta = t \sum_{k=1}^{\bar{k}} d_k \ (N_{Qk} - \ k + 1 + 3p)$$

$$\bar{k} = 4$$

This result can be generalized to any value of EU latency, for D-RISC like machines. Moreover, it is valid also for EU-EU dependencies.

Branch prediction

- For **high-latency** pipelined EU implementations, the effect on logical dependencies degradation is significant
- When logical dependencies are applied to predicate evaluation in branch instructions, e.g.

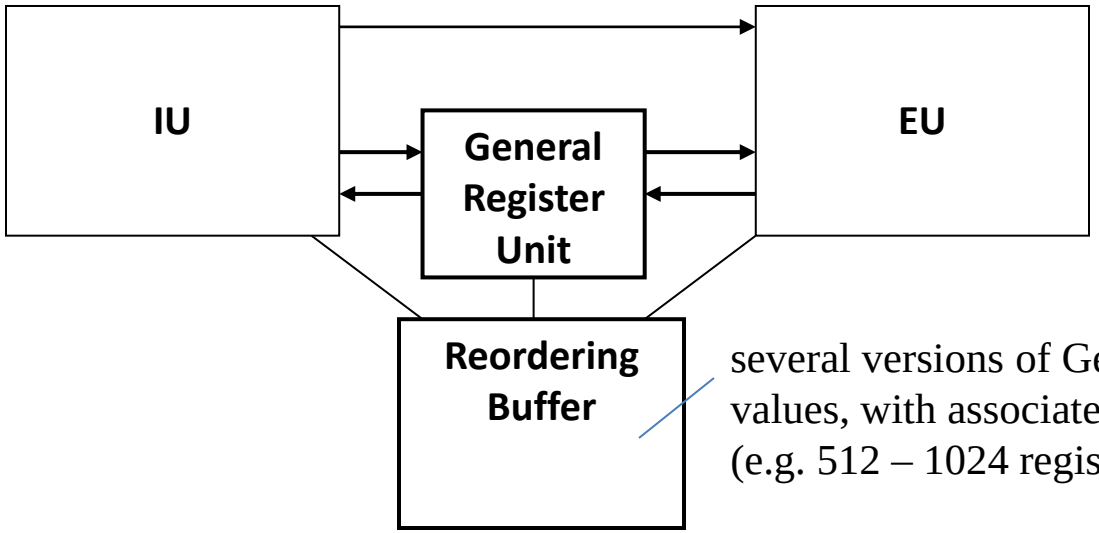


the so-called Branch Prediction technique can be applied.

Branch prediction

- Try to continue along the path corresponding to false predicate (no branch)
  - Compile the program in such a way that the false predicate corresponds to the most probable event, if known or predictable
- On- condition execution of this instruction stream
  - Save the RG state during this on-condition phase
  - Additional unit for RG copy; unique identifiers (IC) associated
- When the values for predicate evaluation are updated, then verify the correctness of the on-condition execution, and, if the prediction was incorrect, apply a recovery action using the saved RG state.

Branch prediction



- Example of **out-of-order execution**.
- From a conceptual viewpoint, the problem is similar to the implementation of an "ordering collector" in a farm structure.
- Complexity, chip area and power consumption are increased by similar techniques.
- Not necessarily it is a cost-effective technique.

Out-of-order behaviour

- Branch prediction is a typical case of out-of-oder behaviour.
- Other out-of-order situations can be recognized: for example see Solution of Exercize 1 about the best-case assumption for cache block transfer evaluation:

```
        LOAD   RA, Rj, Ra
LOOP_j: LOAD   RB, Rj, Rb
        INCR   Rj
        MUL    Ra, Rb, Ra
        ADD    Rc, Ra, Rc
        IF <   Rj, RM, LOOP_j,
               delayed_branch
        LOAD   RA, Rj, Ra
```

The base-case behaviour is implementable provided that LOAD RB, … can be executed during the cache fault handling caused by LOAD RA, …. That is, EU must be able to "skip" instruction LOAD RA, … and to execute LOAD RB, …, postponing the execution of LOAD RA, … until the source operand is sent by DM (LOAD RA, … instruction has been received by EU, but it has been saved in a EU internal buffer).

This can be done

- by a proper instruction ordering in the program code (e.g. the two LOADs can be exchanged), or
- by compiler-provided *annotations* in instructions (if this facility exists in the definition of the assembler machine), or
- by some *rules in the firmware interpreter* of EU: e.g., the execution order of a sequence of LOAD instructions, received by EU, can be altered (no logical dependencies can exist between them if they have been sent to EU).

Superscalar architectures: a first broad outline

Basic case: 2-issue superscalar

- In our Pipelined CPU architecture, each element of the instruction stream generated by IM consists of 2 consecutive instructions
  - Instruction cache is 2-way interleaved or each cell contains 2 words ("Long Word")
- IU prepares both instructions in parallel (during at most 2 clock cycles) and delivers them to DM/EU, or executes branches, provided that they are executable (valid instructions and no logical dependency)
  - It is also possibile that the first instruction induces a logical dependency on the second one, or it is a branch
- DM and EU can be realized with sufficient bandwidth to execute 2 instructions in 2 clock cycles
- Performance achievement: the communication latency (2τ) is fully masked:

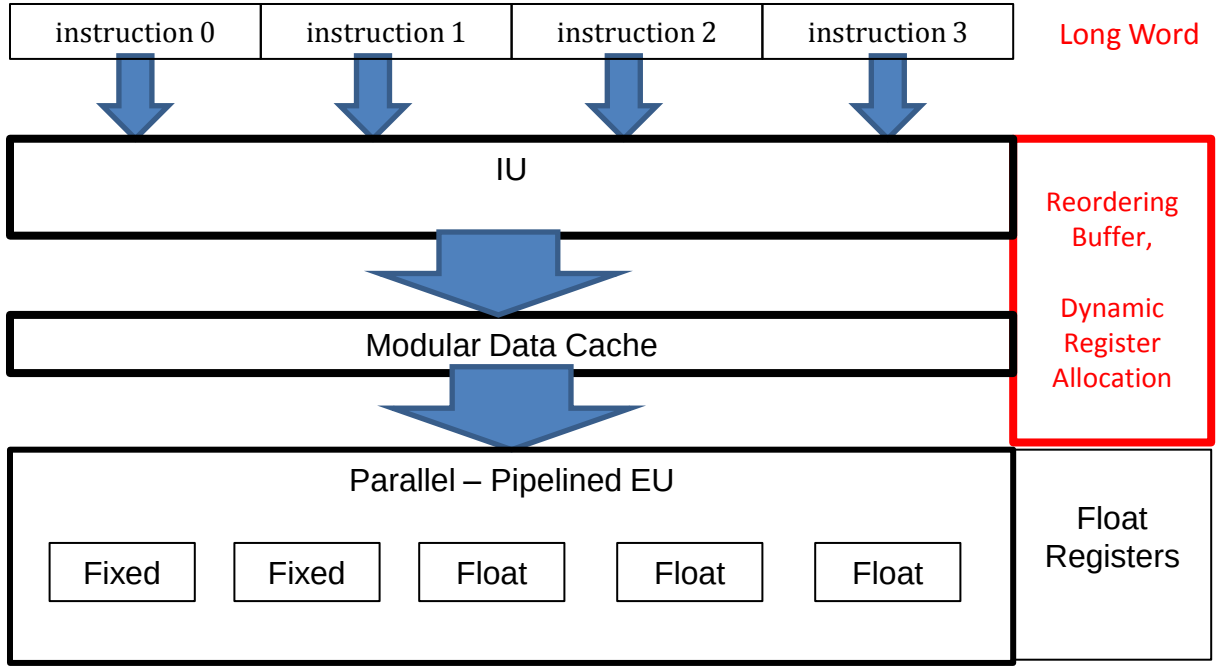
$$\mathcal{P}_{id} = 2/T_{id} = 1/\tau = f_{clock} \quad \text{instructions/sec}$$

  - e.g. f_{clock} = 4 GH2, $\mathcal{P}_{id}$= 4 GIPS

  though performance degradations are greater than in the basic Pipelined CPU (ε is lower), unless more powerful optimizations are introduced
  - branch prediction, out-of-order

Superscalar CPUs

- In general, *n-issue superscalar* architectures can be realized
  - n = 2 – 4 – 8 – 16



Superscalar and VLIW

- All the performance degradation problems are increased
- The Long Word processing and the out-of-order control introduces serious problems of *power consumption*
  - unless some features are sacrificed,
  - e.g. less cache capacity (or no cache at all !) in order to find space to Reordering Buffer and to large intra-chip links
- This is one of the main reasons for the “Moore Law” crisis
- *Trend*: multicore chips, where each core is a basic, in-order Pipelined CPU, or a 2-issue in-order superscalar CPU.
- **VLIW (Very Long Instruction Word) architectures**: the *compiler* ensures that all the instructions belonging to a (Very) Long Word are independent (and static branch prediction techniques are applied)
  - achievement: reduced complexity of the firmware machine.

A general architectural model for Instruction Level Parallelism

- The subject of Superscalar architectures will not be studied in more depth
  - further elements will be introduced when needed
  - in a succesive part of the Course, the *Multithreading* architectues will be studied
- However, it is important to understand the general conceptual framework of Instruction Level Parallelism:
  - **Data-flow computational model** and **Data-flow machines**

(all the superscalar/multithreading architectures apply this model, at least in part)

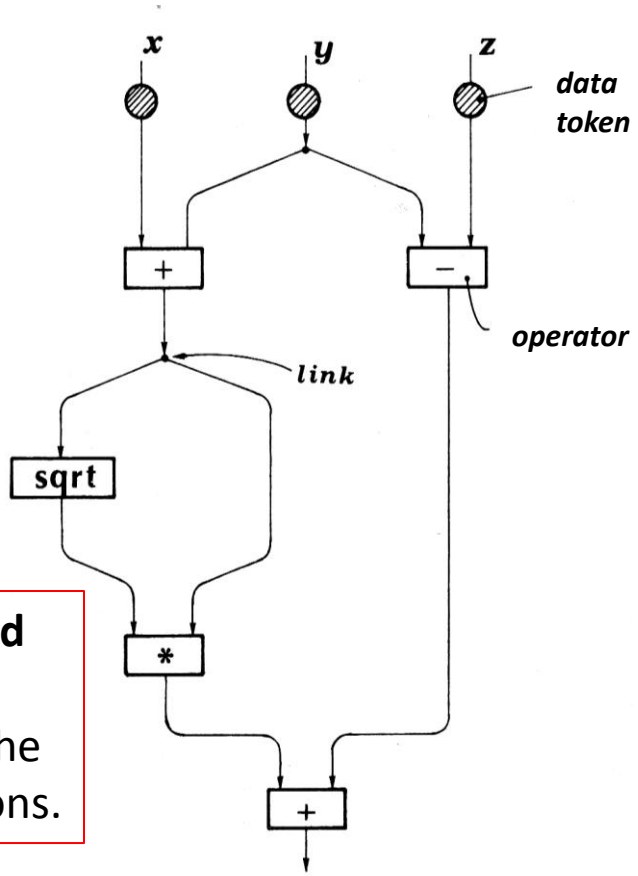
(the commercial flop of some interesting architectures –including VLIW and pure data-flow – remains an ICT mystery …)

Data-flow computational model

- The executable version (assembler level) of a program is represented as a **graph** (called *data-flow graph*), and *not* as a linear sequence of instructions
  - Nodes correspond to instructions
  - Arcs correspond to channels of values flowing between instructions
- Example:

  $(x + y) * sqrt(x + y) + y - z$

  - *application of* ***Bernstein conditions***



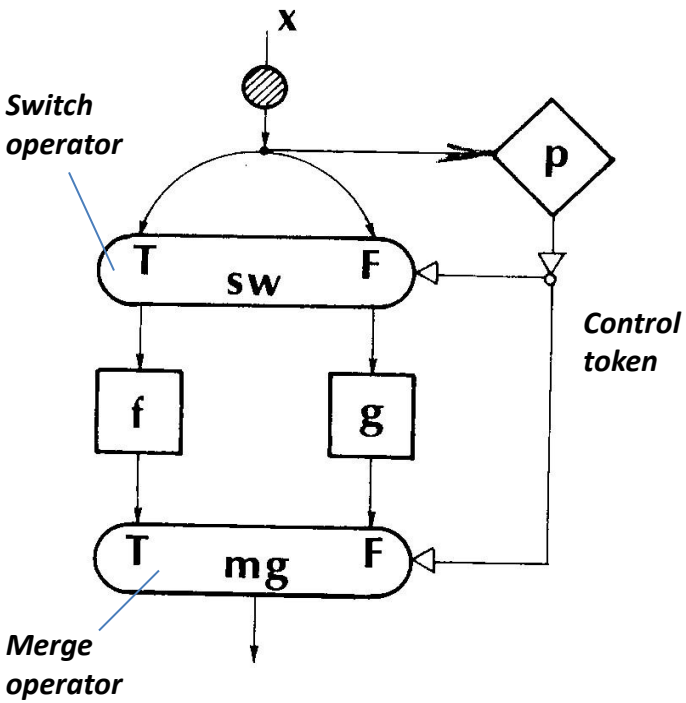
An instruction is **enabled**, and can be executed, **if and only if** its input values ("tokens") are ready.
Once executed, the input tokens are removed, and the result token is generated and sent to other instructions.

Data-flow computational model

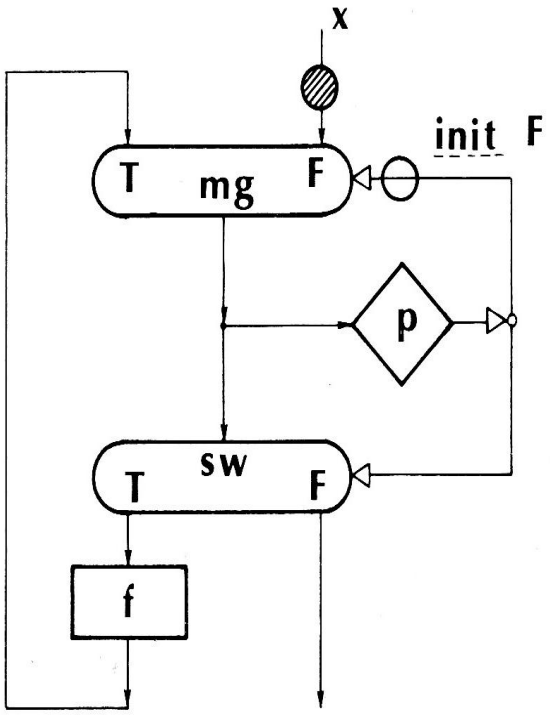
- Purely functional model of computation at the assembler level
  - a *non Von Neumann* machine.
- No variables (conceptually: no memory)
- No instruction counter
- Logical dependencies are not viewed as performance degradation sources, on the contrary they are the only mechanism for instruction ordering
  - *only the strictly necessary logical dependencies are present in a data-flow program,*
  - *consistency problems and out-of-order execution issues don't exist or are solved implicitly and automatically, without additional mechanisms except communication.*
- Moreover, the process / thread concept disappears: *instructions belonging to distinct programs can be executed simultaneously*
  - a data-flow instruction is the unit of parallelism (i.e., it is a process)
  - *this concept is the basic mechanism of multithreading / hyperthreading machines.*

Data-flow graphs

Conditional expression:
if p(x) then f(x) else g(x)



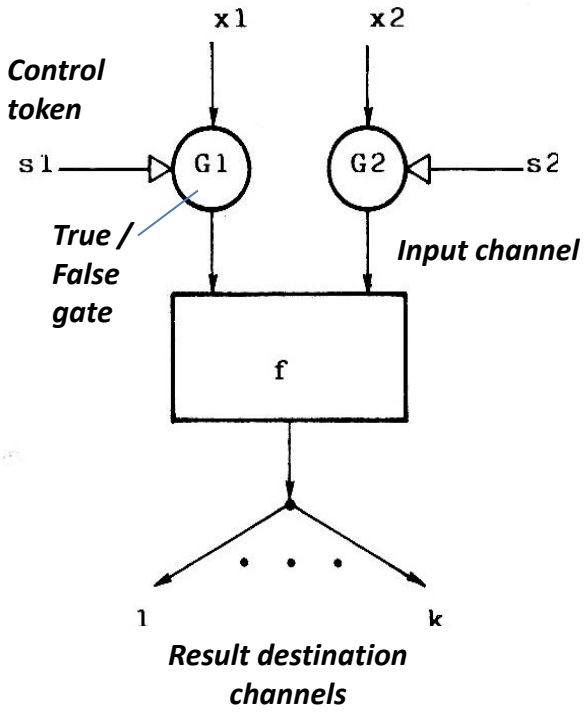
Iterative expression:
while p(x) do new X = f(X)

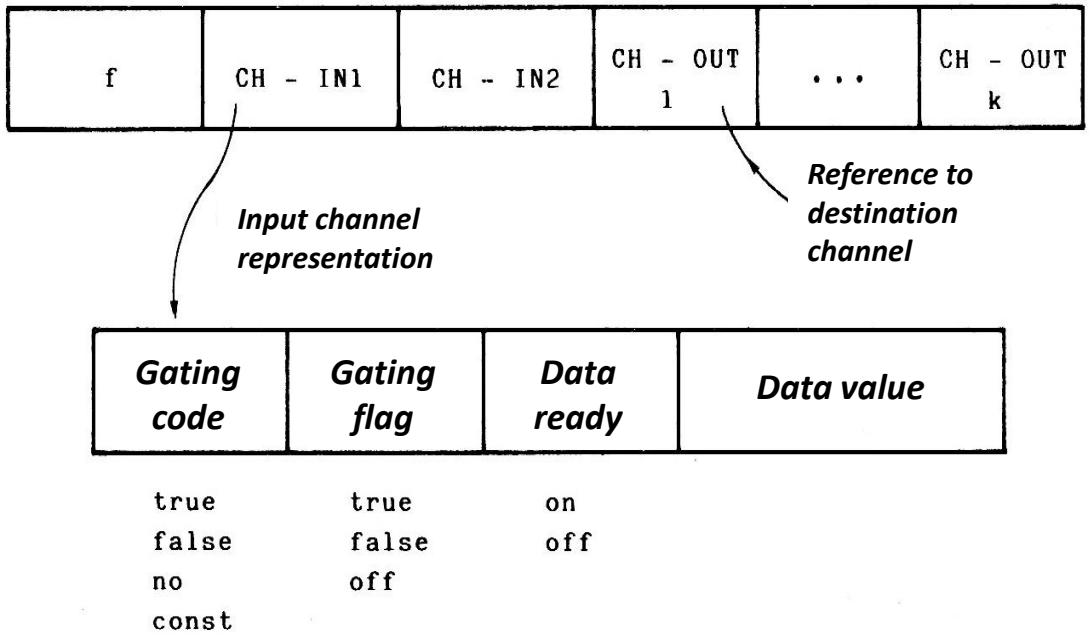


Any data-flow program (data-flow graph) is compiled as the *composition* of arithmetic, conditional and iterative expressions (data-flow subgraphs), as well as of recursive functions (not shown).

Basic data-flow architecture

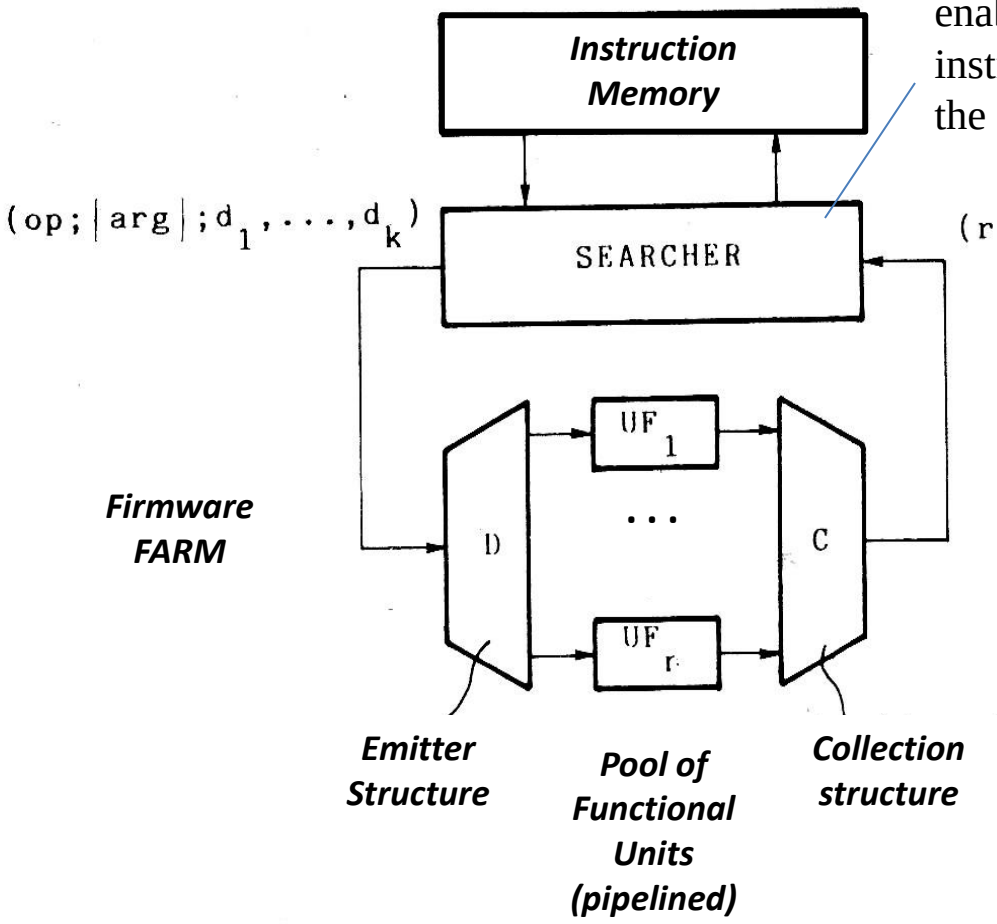
Data-flow instructions are encoded as *information packets* corresponding to segments of the data-flow graph:





Analogy: a (fine-grain) process in the message-passing model.

Basic data-flow architecture



Stores a result value into an input channel of the destination instruction, verifies the instruction enabling (do all input channels of this instruction contain data?), and, if enabled, sends the instruction to the execution farm.

Notes:

1. in this model, a memory is just an implementation of queueing and communication mechanisms;
2. instructions of more than one program can be in execution simultaneously.