united nations
educational, scientific
and cultural
organization

international atomic
energy agency

the
abdus salam
international centre for theoretical physics

301/1246-4

Microprocessor Laboratory
Third Regional Course on
Advanced VLSI Design Techniques
13 November - 1 December 2000

Lima - Peru

EFFECTIVE IMPLEMENTATION OF A 32-BIT RISC PROCESSOR

Pirouz BAZARGAN SABET
University Paris 6
LIP6-ASIM
4, place Jussieu
75252 Paris Cedex)5
FRANCE

Effective Implementation of a 32-bit RISC Processor

Pirouz Bazargan Sabet

University of Paris 6 - LIP6 - ASIM

Pirouz.Bazargan-Sabet@lip6.fr

1

Outline

- ❏ Architecture of a RISC Processor
- ❏ Implementation

Introduction

Architecture ?

- External view
- All aspects visible from the user (programmer) point of view
- Specifications of the processor

What it is supposed to do

Introduction

Implementation ?

- Internal view
- Designer's point of view
- How many time does it take to perform some operation

Which hardware can I use and how can it be organized to make the specifications feasable

Architecture

- ❑ Software visible registers
- ❑ Memory addressing
- ❑ The instruction set
- ❑ The exception mecanism

Architecture

Architecture of the Mips processor

Mips ?

- A 32-bit processor
- One of the first two RISC processors
- Defined in 1981 by the Architecture Research Group of the Stanford University (J. Hennessy)

Architecture

Simplified Mips-R3000

- No floating point operations
- No virtual memory

Architecture

➢ □ **Software visible registers**

□ Memory addressing

□ The instruction set

□ The exception mecanism

Architecture

Software visible registers

Registers that can be manipulated
in the assembly language

Software visible registers

- 32 common registers - 32-bits

  called Integer Registers

  R_0 .... R_{31}

  addressable from their number

Software visible registers

- R_0 **: The Trash Register**

  R_0 contains always 0
  A value written in R_0 is lost

- R_{31} **: The Link Register**

  When calling a sub-program, the return address is saved in R_{31}

Software visible registers

- 2 32-bit special registers : **HI** and **LO** used by multiply and divide instructions

Multiply	HI	32 most significant bits
	LO	32 least significant bits
Divide	HI	Result
	LO	Remainder

Software visible registers

- ❑ 4 32-bit special registers : Coprocessor Registers (needed to implement an operating System)

SR	Status Register
CAUSE	Cause Register (cause of exceptions)
EPC	Exception Program Counter (return address in case of exception)
BAR	Bad Address Register (invalid memory address)

Architecture

- ❑ Software visible registers
- ➢ ❑ **Memory addressing**
- ❑ The instruction set
- ❑ The exception mecanism

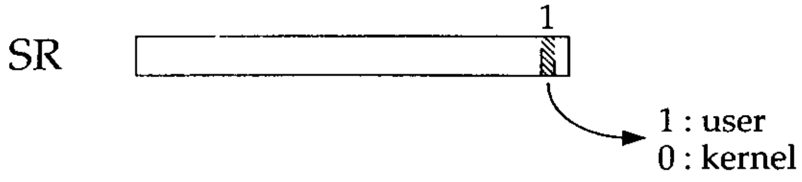
Memory addressing

- memory space of 4 Gbyte (32-bit address)
- Read / Write operations
- Byte / Half-word / Word
  (2 bytes) (4 bytes)

Memory addressing

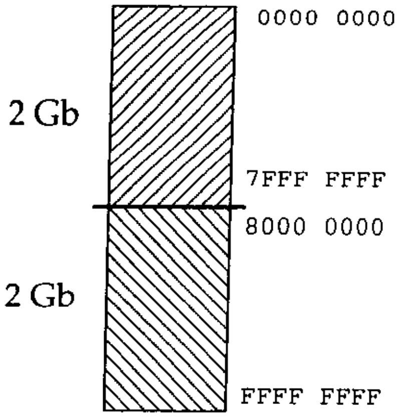
- The processor can operate under 2 modes
  User / Kernel

One bit in the Status Register defines the current mode



needed to implement an OS

Memory addressing

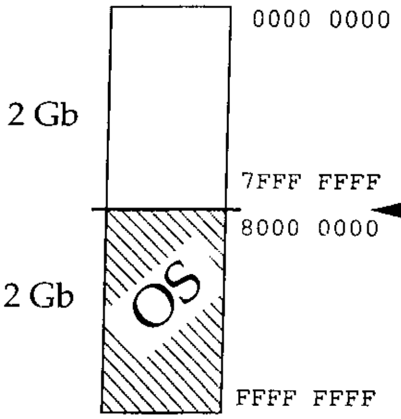


The memory space is divided into two parts

In User mode the processor can only access the addresses ranged from 0000 0000 to 7FFF FFFF

Memory addressing

The Os protects the hardware from a miss-working user progam



Frontier protects the OS

➥ Exception

Memory addressing

- Mips respects the address alignment convention

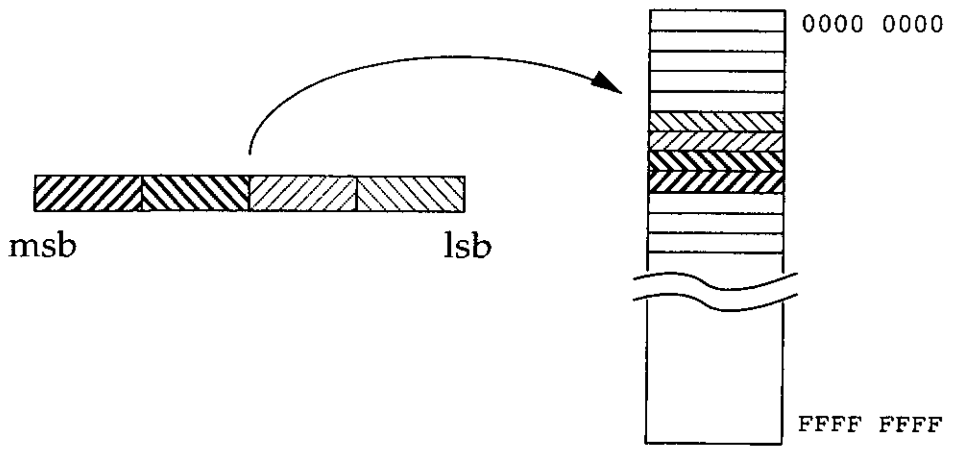
The address of an object of N bytes must be a multiple of N

address of words	▷	multiple of 4
address of half-words	▷	multiple of 2
address of bytes	▷	multiple of 1

↳ Exception

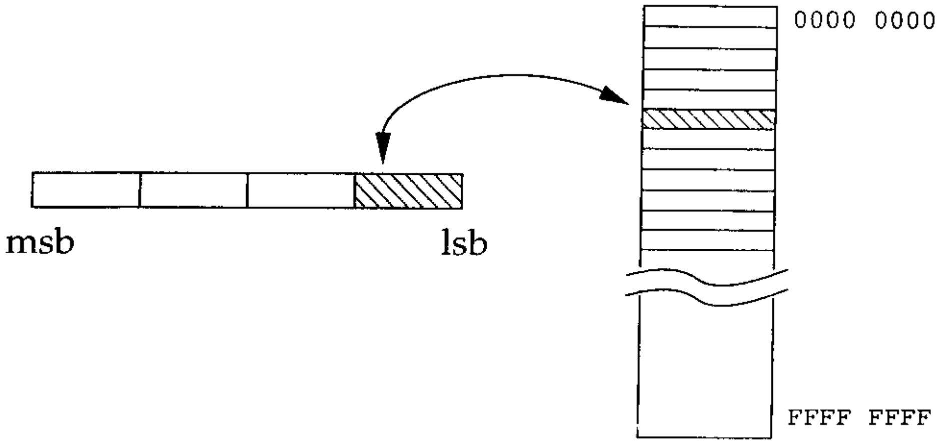
Memory addressing

- Memory organization : Little Endian



Memory addressing

- Memory organization : Data Alignment convention



Instruction Set

RISC Architecture

- Simple instructions
- All the instructions have the same size (32 bits)
- Instructions with 3 operands (2 read, 1 write)
- No operation involving memory operands (only load and store operations)

Instruction Set

- ❏ Arithmetic and Logic
- ❏ Control
- ❏ Memory Access
- ❏ System

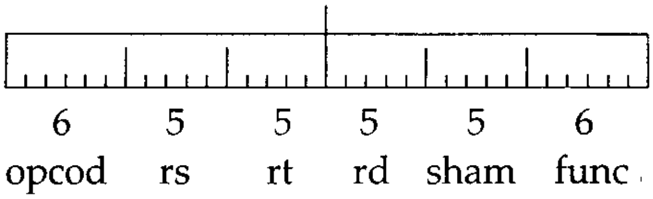
Instruction Set

3 instruction formats

- R Register / register instructions
- I Immediate instructions
- J Jump

Instruction Set

R format

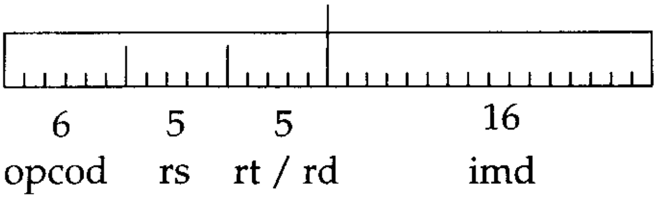

6 5 5 5 5 6

opcod rs rt rd sham func

opcod	operation code
func	extended operation code
rs	# of source operand
rt	# of source operand
rd	# of destination operand
sham	# of bits the operand is shifted

Instruction Set

I format

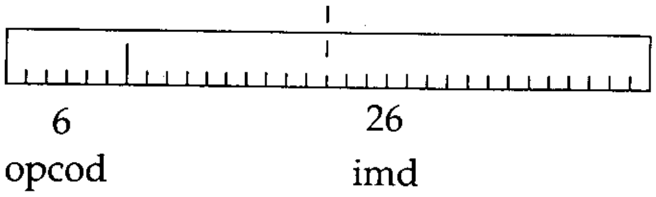

6 5 5 16

opcod rs rt / rd imd

opcod	operation code
rs	# of source operand
rt / rd	# of source or destination operand
imd	an immediate value

Instruction Set

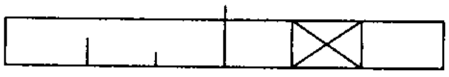
J format



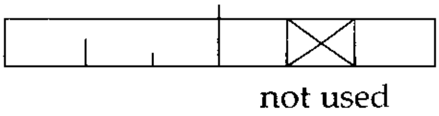
opcod operation code

imd an immediate value

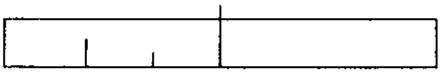
Arithmetic and Logic

assembly language	action	format
Add Rd, Rs, Rt	Rs + Rt -> Rd operands are signed (overflow exception)	 R

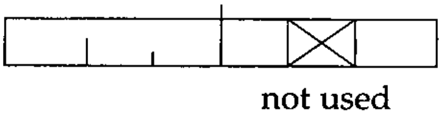
Arithmetic and Logic

assembly language	action	format
Addu Rd, Rs, Rt	Rs + Rt -> Rd operands are unsigned	 R

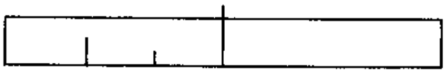
Arithmetic and Logic

assembly language	action	format
Addi Rd, Rs, I	Rs + I -> Rd operands are signed (overflow exception) I is sign extended	 I

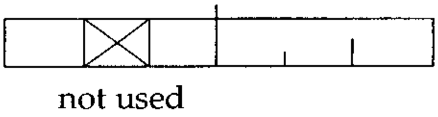
Arithmetic and Logic

assembly language	action	format
Or Rd, Rs, Rt	Rs or Rt -> Rd operands are unsigned	 R

Arithmetic and Logic

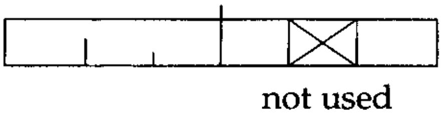
assembly language	action	format
Ori Rd, Rs, I	Rs or I -> Rd operands are unsigned I is extended with 0	 I

Arithmetic and Logic

assembly language	action	format
Shl Rd, Rt, n	Rt << n -> Rd operands are unsigned	

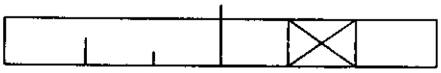
R

Arithmetic and Logic

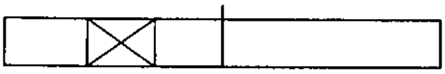
assembly language	action	format
Shlv Rd, Rt, Rs	Rt << Rs -> Rd operands are unsigned	

R

Arithmetic and Logic

assembly language	action	format
Slt Rd, Rs, Rt	Rs < Rt ? yes : Rd <- 1 no : Rd <- 0 operands are signed	 R

Arithmetic and Logic

assembly language	action	format
Lhi Rd, I	Rd <- I , 0000 I is loaded into the 16 high bits of Rd	 I

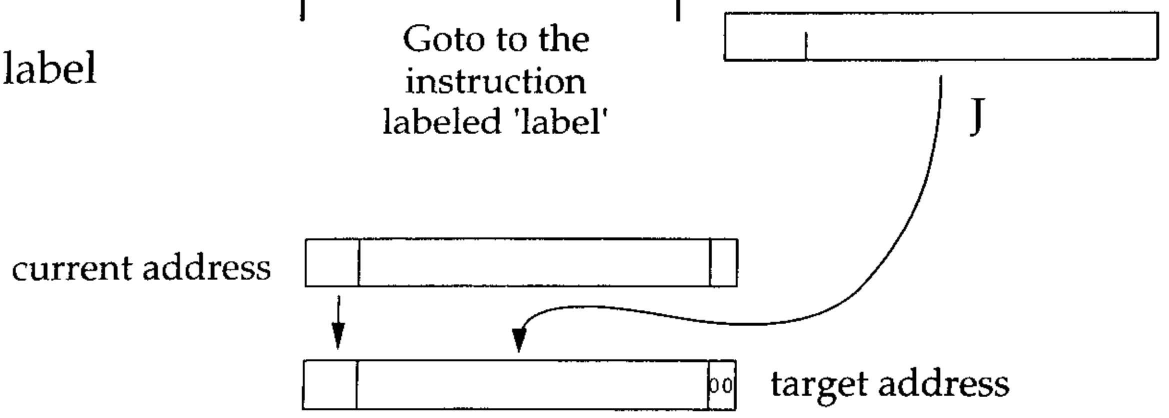
Control Instructions

- ❑ Unconditional Branches
- ❑ Conditional branches

Control Instructions

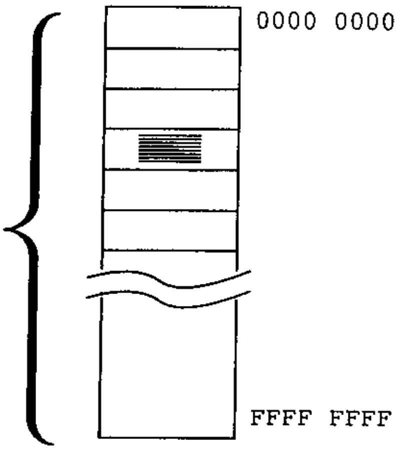
unconditional branch

assembly language	action	format
J label	Goto to the instruction labeled 'label'	



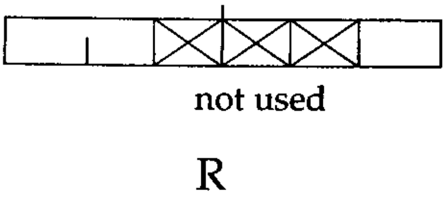
Control Instructions

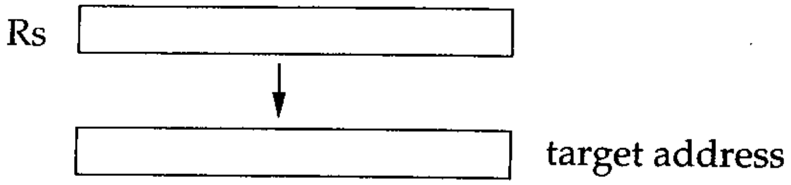
The memory space is divided into 16 blocs (256 Mb each)



Control Instructions

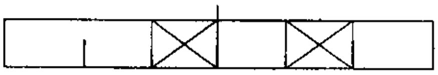
unconditional branch

assembly language	action	format
Jr Rs	Goto to the instruction labeled 'label'	



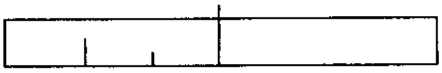
Control Instructions

unconditional branch

assembly language	action	format
Jalr Rs	Same as Jr The return address is saved into the Link Register (R31)	 not used R

Control Instructions

conditional branch

assembly language	action	format
Beq Rs, Rt, label	Compare Rs = Rt ? Branch to the 'label' if true	 I

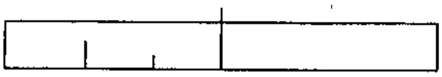
Rs = Rt ?

yes : current address + 4 + I * 4

no : continue

Control Instructions

conditional branch

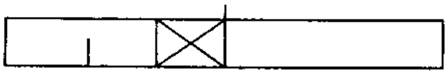
assembly language	action	format
Bne Rs, Rt, label	Compare Rs = Rt ? Branch to the 'label' if false	 I

Rs = Rt ? no : current address + 4 + I * 4

yes : continue

Control Instructions

conditional branch

assembly language	action	format
Blez Rs, label	Compare Rs ≤ 0 ? Branch to the 'label' if true	 not used I

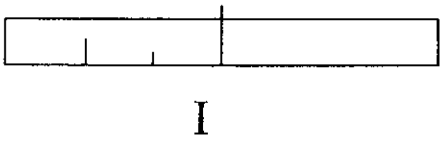
Rs ≤ 0 ? yes : current address + 4 + I * 4

no : continue

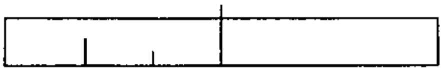
Memory Access

- ❑ Loads
- ❑ Stores

Memory Access

assembly language	action	format
Lw Rd, I (Rs)	Load 4 bytes from memory into Rd memory address = Rs + I signed immediate	

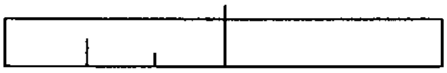
Memory Access

assembly language	action	format
Sw Rt, I (Rs)	Store the 4 bytes of Rt into the memory	

memory address = Rs + I

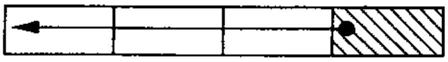
signed immediate

Memory Access

assembly language	action	format
Lb Rt, I (Rs)	Load 1 byte from the memory into Rd	

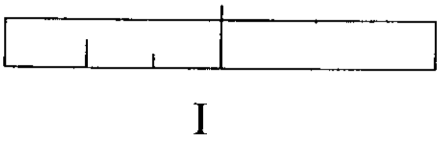
memory address = Rs + I

signed immediate

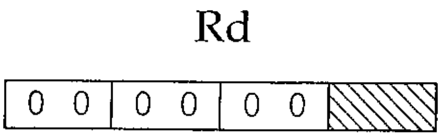


sign extension

Memory Access

assembly language	action	format
Lbu Rt, I (Rs)	Load 1 byte from the memory into Rd	

memory address = Rs + I
signed immediate



Architecture

- ❑ Software visible registers
- ❑ Memory addressing
- ➢ ❑ **The instruction set**
- ❑ The exception mecanism

Implementation

A given architecture can be implemented in many different ways

Performance Complexity

Implementation

- ❑ RISC vs. CISC concept
- ❑ Concept of pipeline
- ❑ An implementation of Mips
- ❑ Pipeline's problems

RISC vs. CISC concept

The RISC concept has been developped in early 80's

CISC	RISC
Complex Instruction Set Computer	Reduce Instruction Set Computer

RISC vs. CISC concept

The basic idea of CISC concept :

Use the improvement of the technology
to offer a more powerful architecture

RISC vs. CISC concept

Reduce the gap between high level languages and the assembly language (IBM 370, VAX, ...)

It is easier to program with a powerful assembly language

RISC vs. CISC concept

3 factors were at the origin of the RISC concept

- ❑ Economical factor
- ❑ User factor
- ❑ Marketing factor

RISC vs. CISC concept

Economical factor

A VAX complex instruction	The equivalent Mips code
Add @3, @2, @1	Lw R1, @1
	Lw R2, @2
	Add R3, R2, R1
	Sw R3, @3

RISC vs. CISC concept

Economical factor

A given program compiled for a RISC processor is 2 to 3 times bigger than the same code generated for a CISC

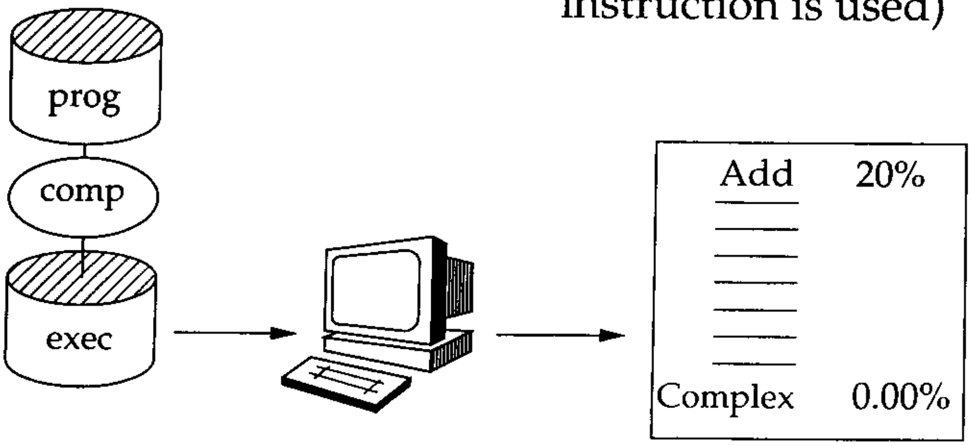
Strong argument in favor of CISC

in 50's - 60's !!

RISC vs. CISC concept

User factor

Designing a new processor -> check the **Mix**
(how frequently each
instruction is used)



RISC vs. CISC concept

User factor

Complex instructions are NOT used

in contradiction with
the CISC's concept

RISC vs. CISC concept

User factor

The CISC concept targets a human user

It is easier to program with a
powerful assembly language ……for a human

Strong argument in favor of CISC

in 50's - 60's !!

RISC vs. CISC concept

In 80's assembly language programmers
have been replaced by compilers

A compiler can only use simple instructions

RISC vs. CISC concept

Marketing factor

Complex architecture

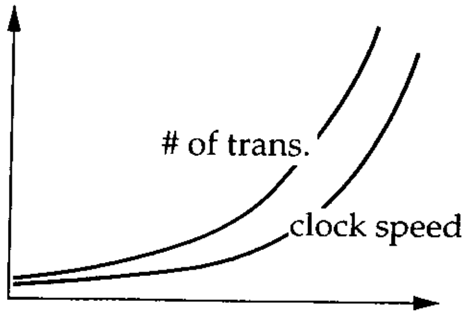
↳ Complex design

↳ long project (3-4 years)

Time-To-Market

RISC vs. CISC concept

Marketing factor



Moore's law

project's length ↗

∇

risk of fail ↗

RISC vs. CISC concept

RISC architecture

↓

Simple implementation

↓

Short time project

↓

Better market and technology prevision

↓

Low risk

RISC vs. CISC concept

RISC Concept

Simplify the hardware < gain in speed / gain in Time-To-Market

gap between high level languages and the assembly language is filled by the compiler

RISC vs. CISC concept

RISC

~~Reduce~~
~~Instrcution~~
~~Set~~
~~Computer~~

Reject
Important
Stuff into
Compiler

Implementation

Objective :

Execute 1 instruction in each cycle

➩ What hardware is needed ?

Implementation

	load	store	add	branch
Read the instruction	✓	✓	✓	✓
Decode	✓	✓	✓	✓
Read operands	✓	✓	✓	✓
Make an operation	✓	✓	✓	○
Memory access	✓	✓	○	○
Save the result	✓	○	✓	○
Compute next inst. @	✓	✓	✓	✓

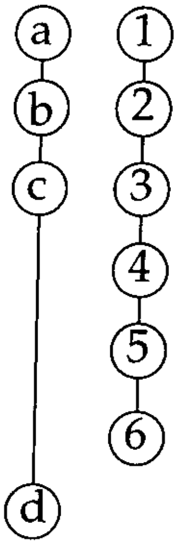
Implementation

All the instructions have the same execution scheme

- Read the instruction
- Decode
- Read operands
- Make an operation
- Memory access
- Save the result
- Compute next inst. @

Implementation

In which order ?



Read the instruction

Decode

Read operands

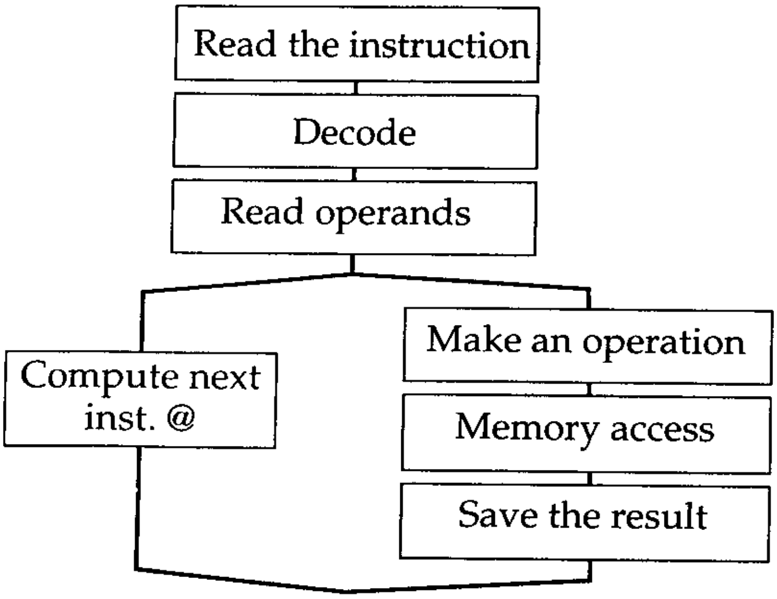
Make an operation

Memory access

Save the result

Compute next inst. @

Implementation


Read the instruction

Decode

Read operands

Compute next inst. @

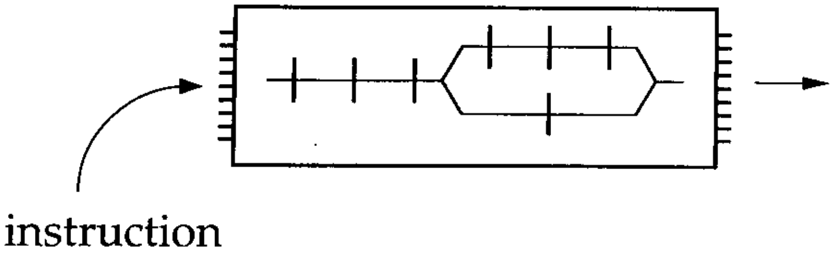
Make an operation

Memory access

Save the result

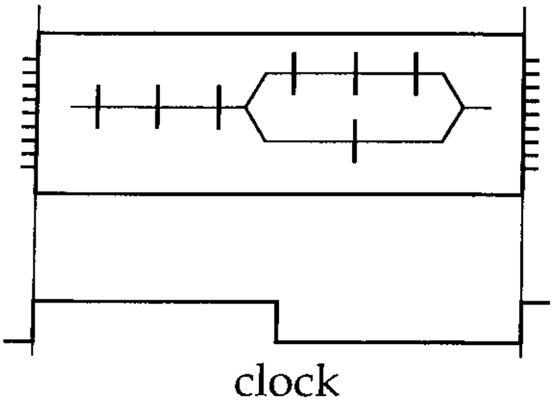
Implementation

A first implementation



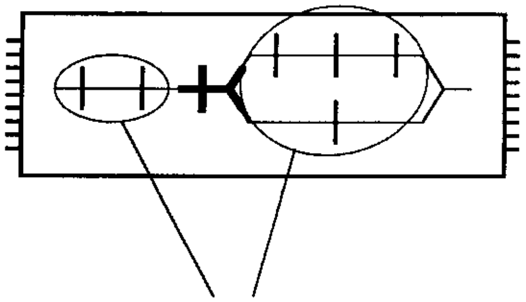
Implementation

Objective 1 instr = 1 cycle ?



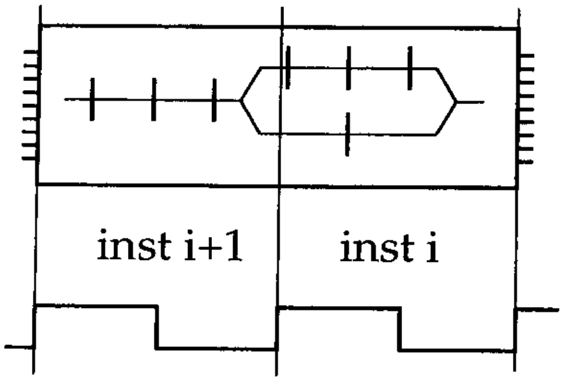
performance

Implementation

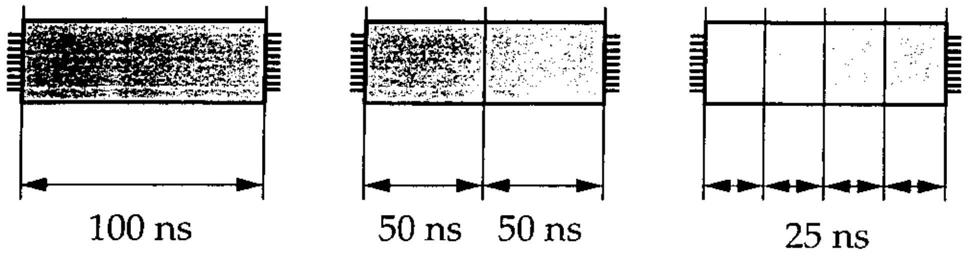


Sleeping hardware

Implementation



Implementation



10 Mi/s 20 Mi/s 40 Mi/s

Implementation

Is there any limitation ?

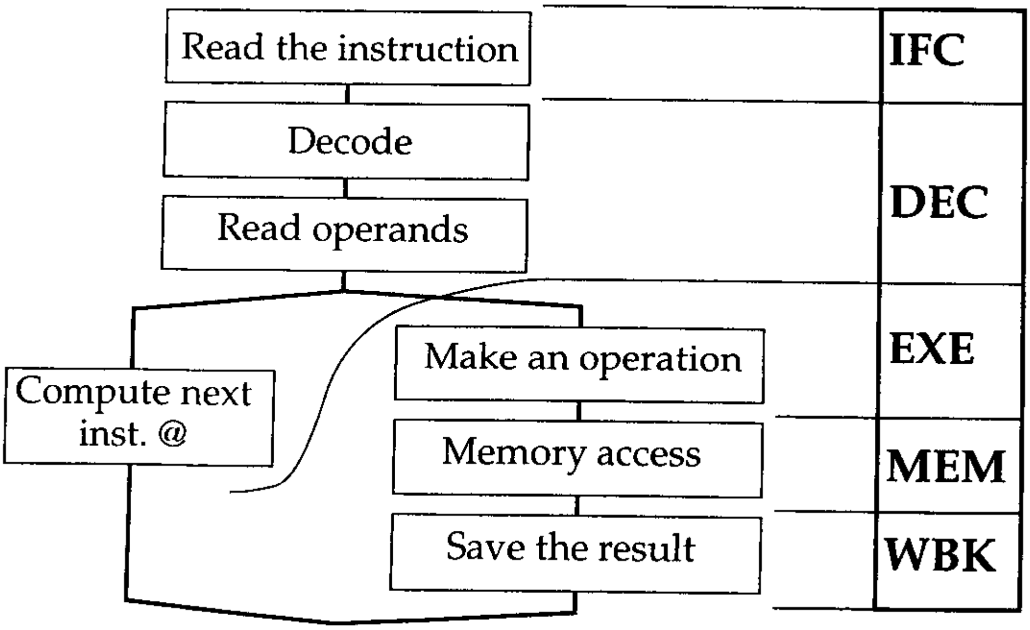
NO !

Implementation

Pipeline rules :

- ❏ pipeline stages must be separated by registers
- ❏ pipeline must be as "balanced" as possible (all stages must have the same length)

Implementation



Read the instruction

Decode

Read operands

Compute next inst. @

Make an operation

Memory access

Save the result

IFC

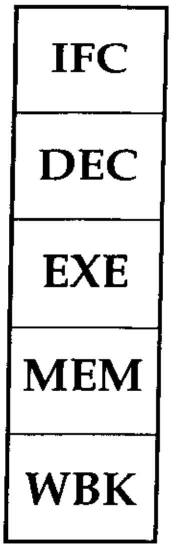
DEC

EXE

MEM

WBK

Implementation

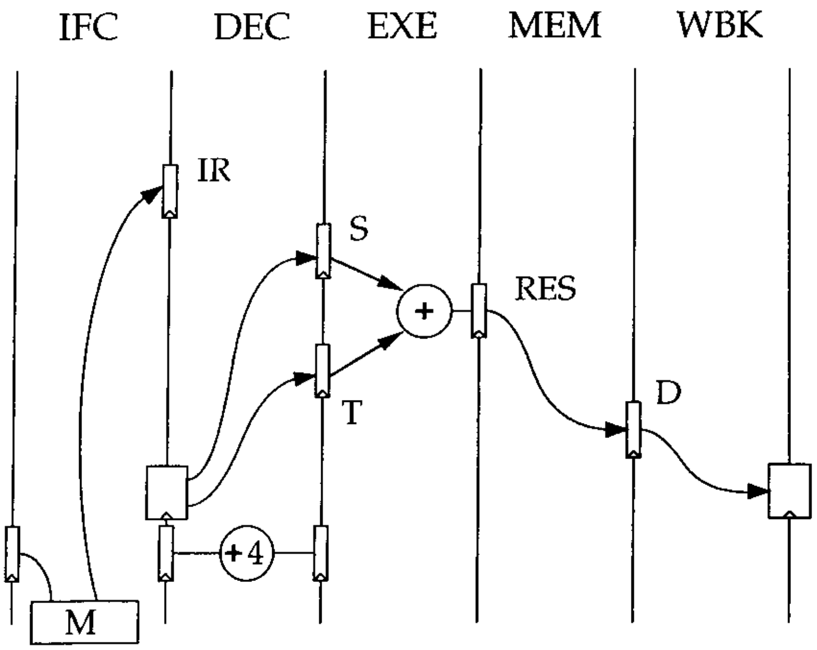


There is no relationship between pipeline stages and operations

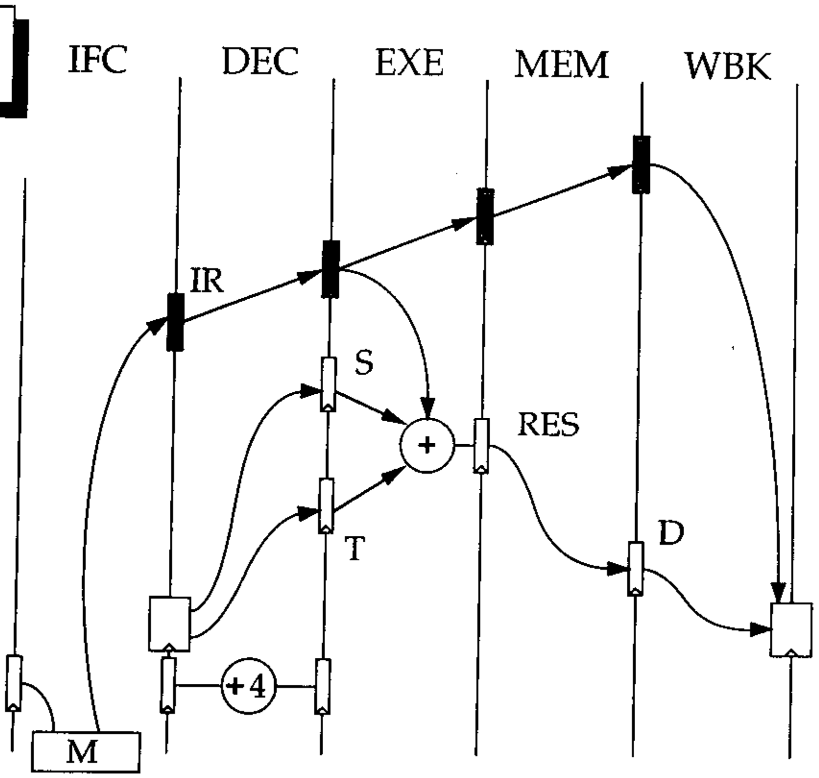
pipeline well balanced in 1981

Implementation

Example : Add Rd, Rs, Rt



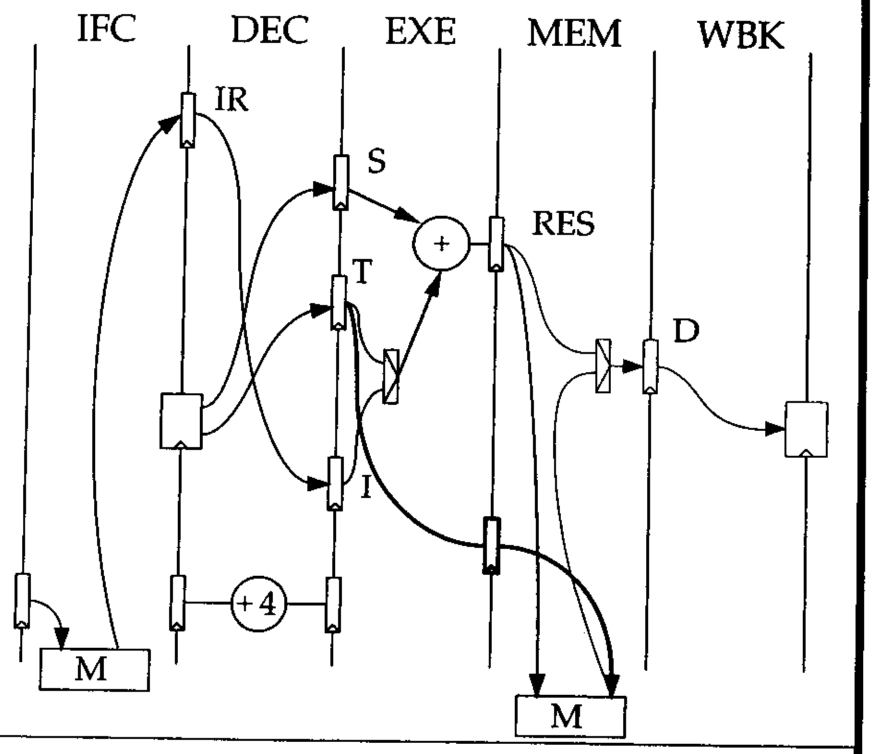
Implementation



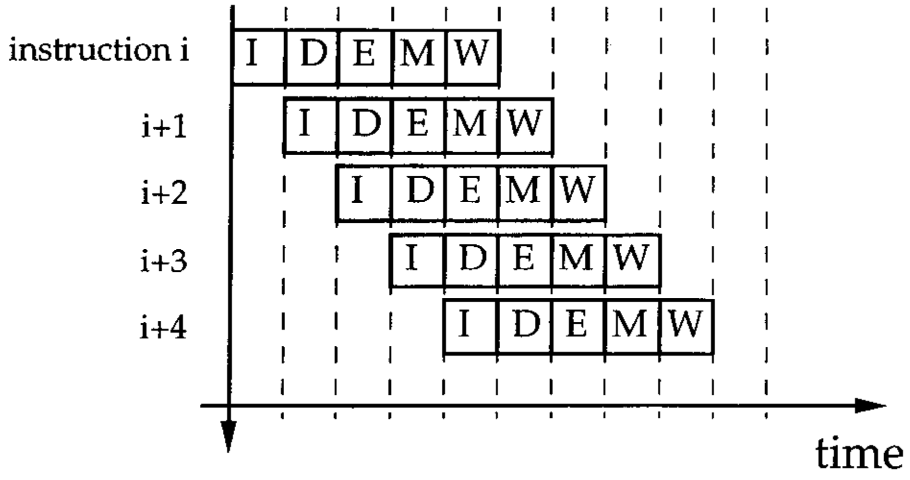
Implementation

Example :

Sw Rt, I (Rs)



Implementation

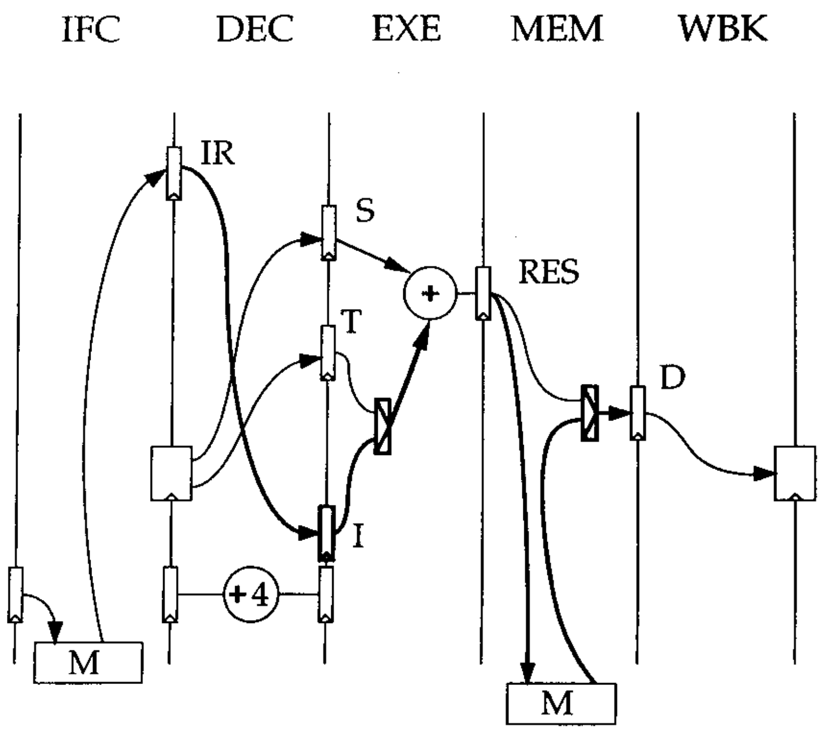


instruction i	I	D	E	M	W				
i+1		I	D	E	M	W			
i+2			I	D	E	M	W		
i+3				I	D	E	M	W	
i+4					I	D	E	M	W

Implementation

Example :

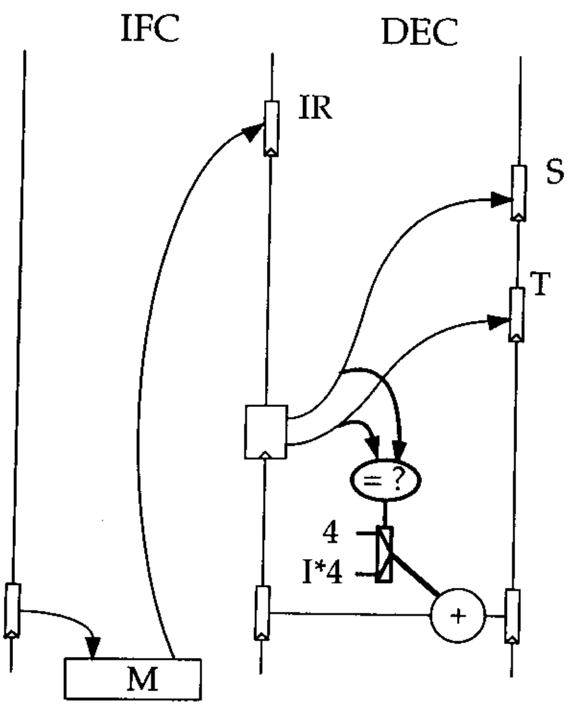
Lw Rd, I (Rs)



Implementation

Example :

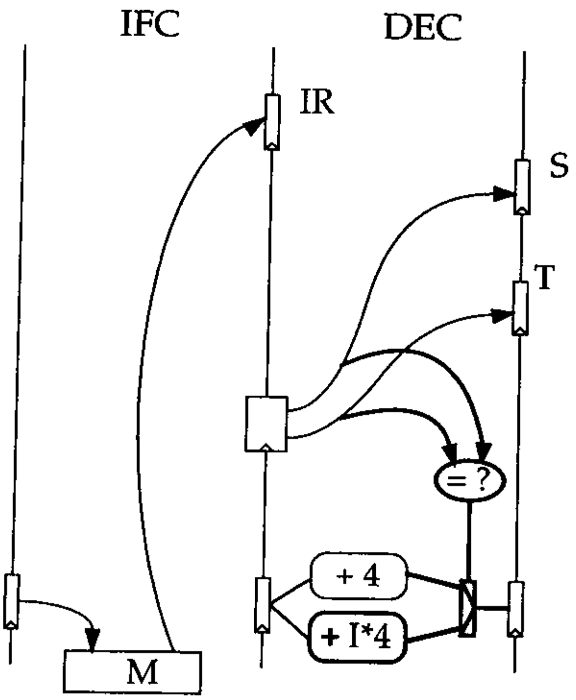
Beq Rs, Rt, label



Implementation

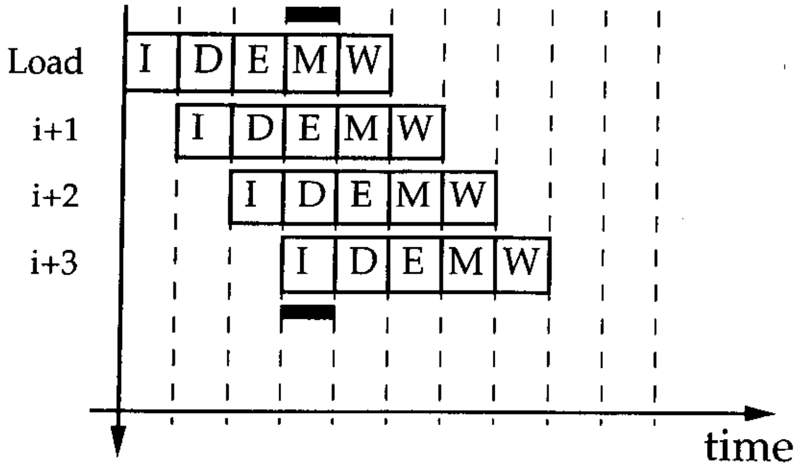
Example :

Beq Rs, Rt, label



Implementation

Can I use the same memory bus ?

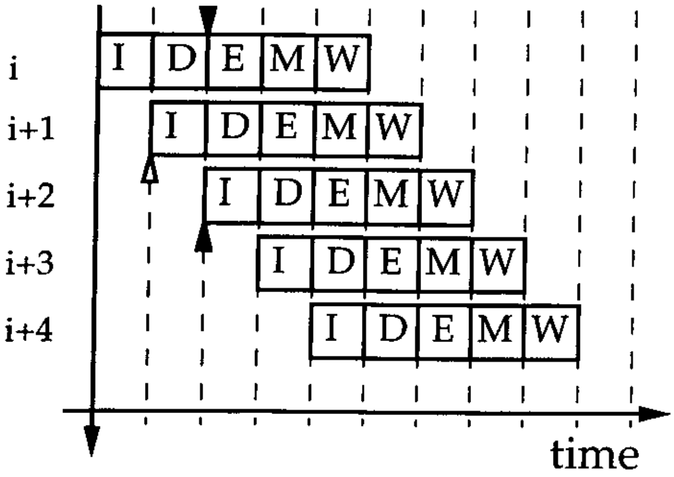


Implementation

Pipeline rule :

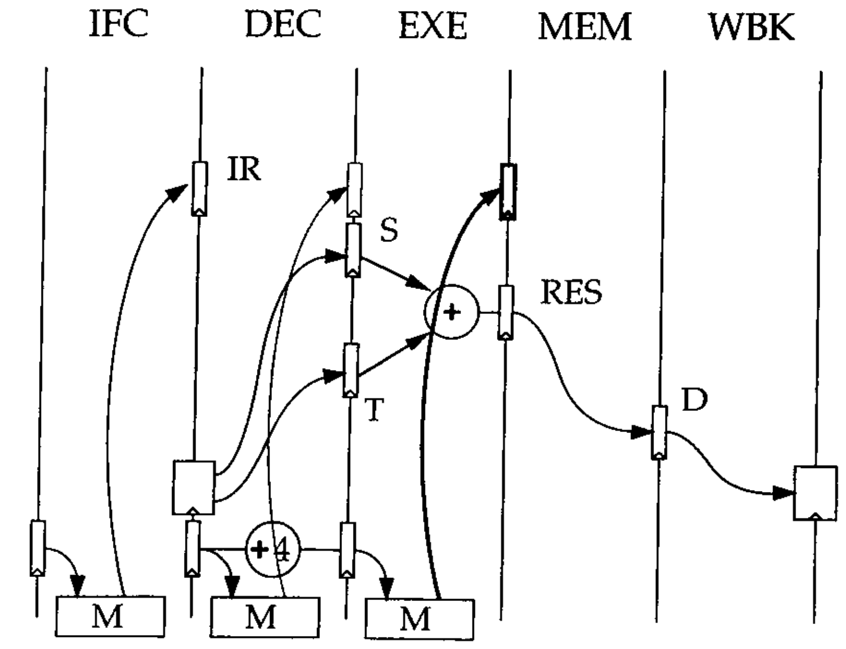
Each piece of hardware must belong to a UNIQUE pipeline stage

Implementation



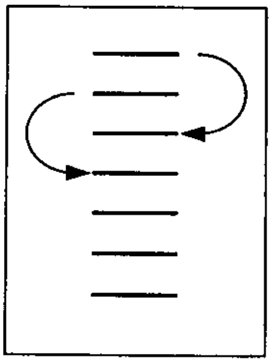
in a pipeline processor the address calculated by the instruction *i* is the address of the instruction *i* + *n* (n ≥ 2)

our case n = 2



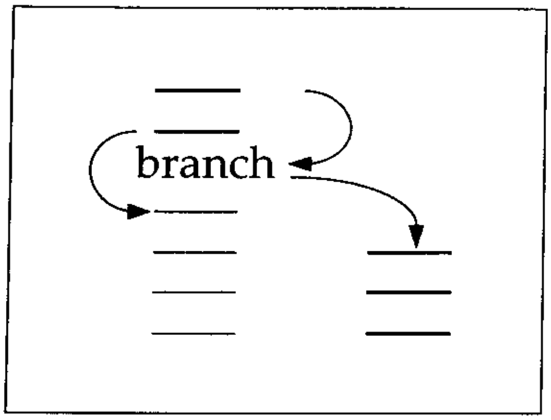
Example :

Add Rd, Rs, Rt



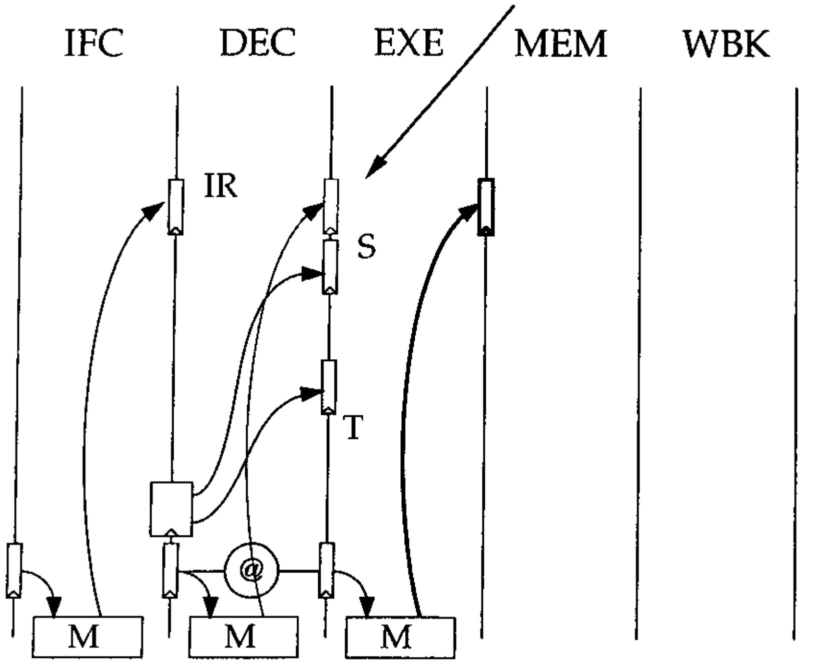
Implementation

Problem with branches :



Implementation

delayed slot instruction



Implementation

Solutions :

❑ Avoid the execution of the delayed slot instruction

▻ hardware control

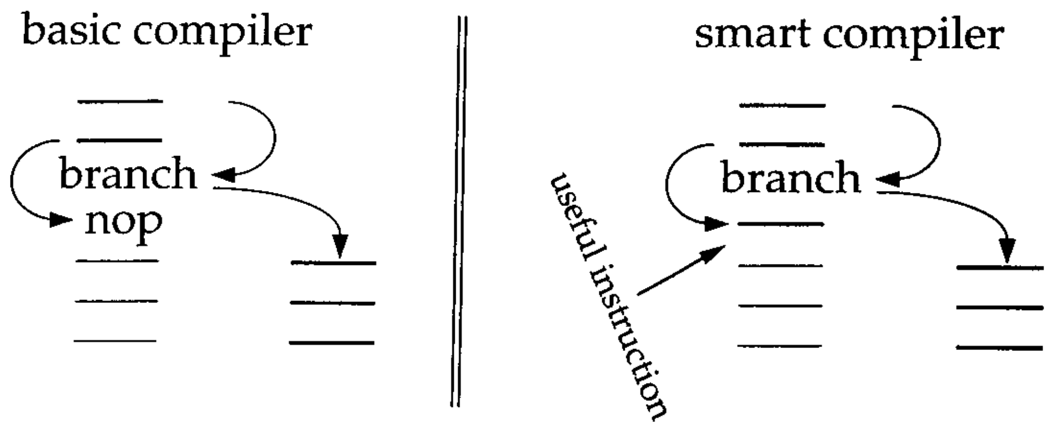
in contradiction with the RISC concept

❑ Let the instruction be executed

▻ ⚠ assembly programs

Implementation

The instruction after a branch is always executed



basic compiler

branch

nop

smart compiler

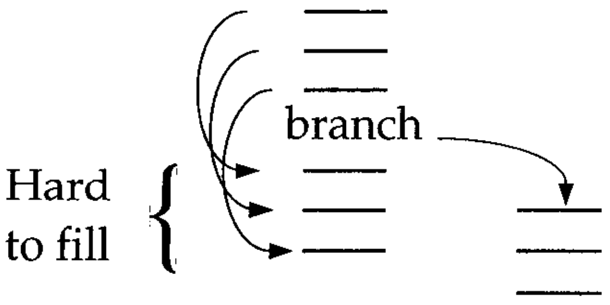
branch

useful instruction

Implementation

The later the address is calculated

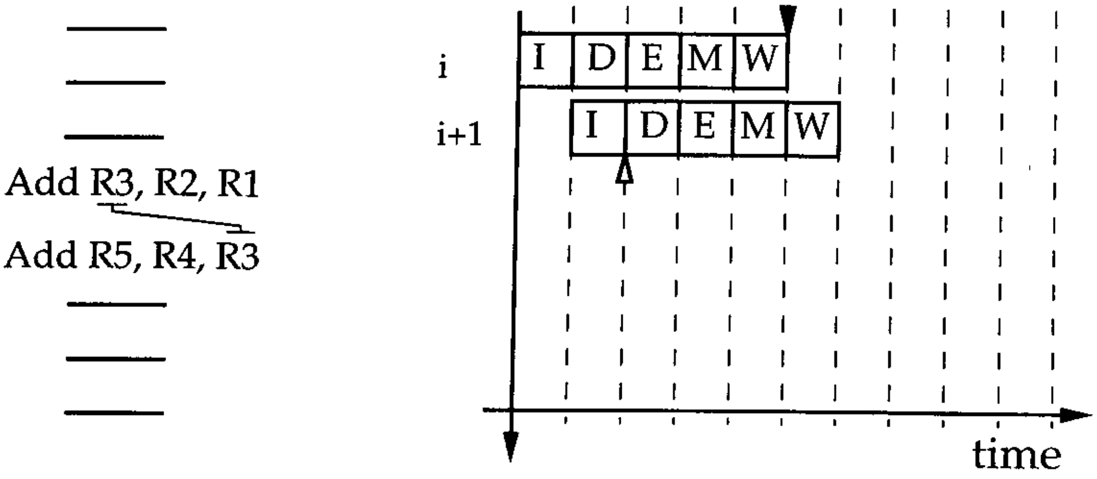
⇨ greater number of delayed slots



Implementation

Problem of data dependency

Implementation



Add R3, R2, R1

Add R5, R4, R3

Implementation

Solutions :

- ❑ Try to resolve the problem inside the hardware
- ❑ Ask the compiler to avoid the problem

Implementation

Ask the compiler to avoid the problem

Add R3, R2, R1

———

———

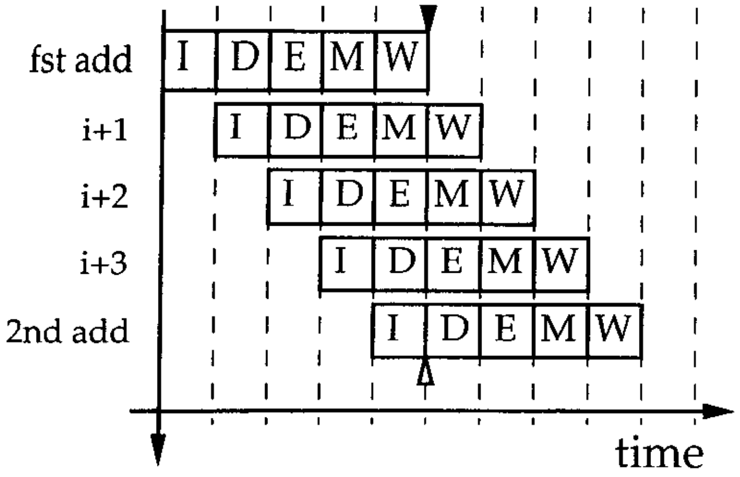
———

Add R5, R4, R3

———

———

———



fst add	I	D	E	M	W				
i+1		I	D	E	M	W			
i+2			I	D	E	M	W		
i+3				I	D	E	M	W	
2nd add					I	D	E	M	W

time

Implementation

Ask the compiler to avoid the problem

Add R3, R2, R1

———

nop

nop

Add R5, R4, R3

———

———

———

performance ↘

Implementation

Try to resolve the problem inside the hardware

———

———

———

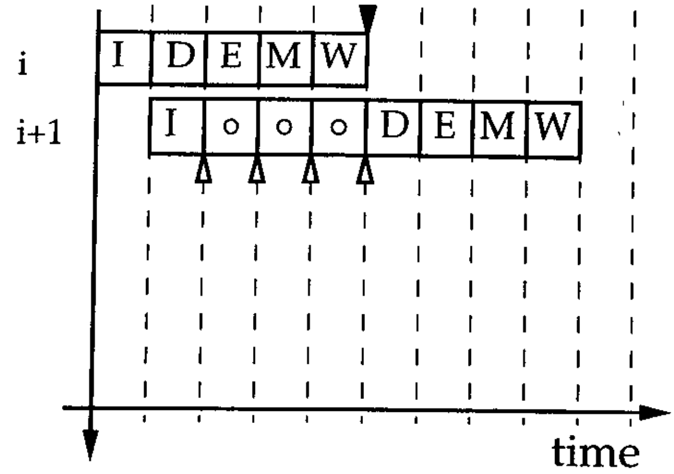
Add R3, R2, R1

Add R5, R4, R3

———

———

———



Implementation

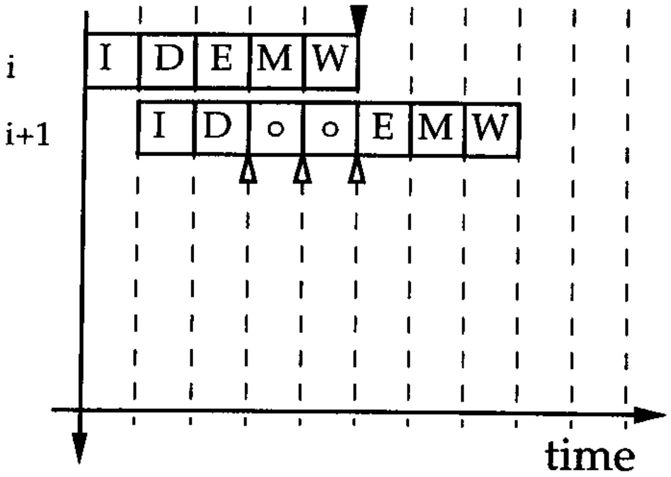
Hardware solution

3 wait cycles are really necessery ?

Data usage

Add R3, R2, R1

Add R5, R4, R3



Implementation

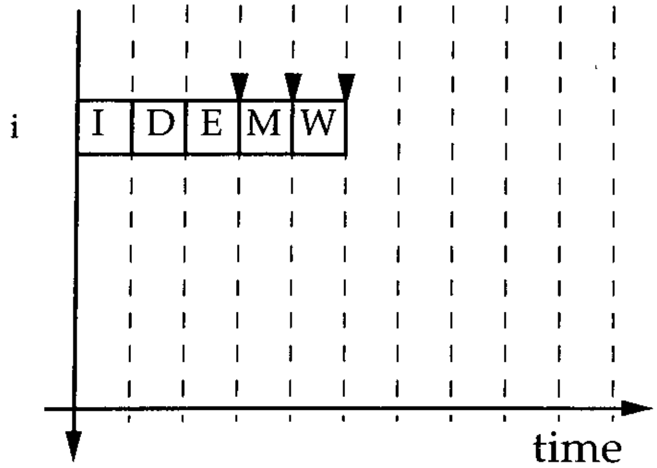
Hardware solution

3 wait cycles are really necessery ?

Availability of data

Add R3, R2, R1

Add R5, R4, R3

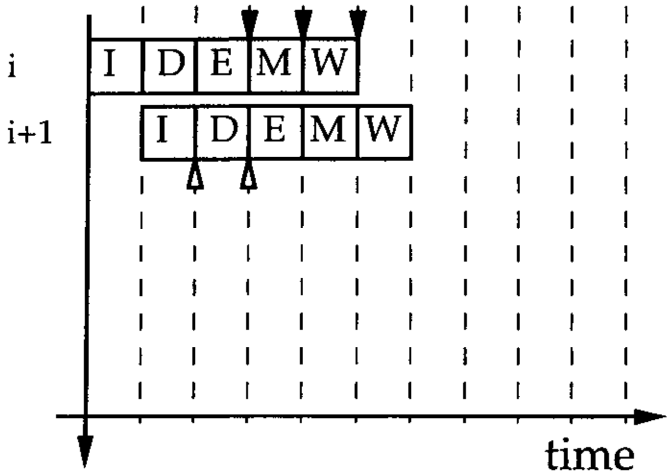


Implementation

Putting all together

Add R3, R2, R1

Add R5, R4, R3

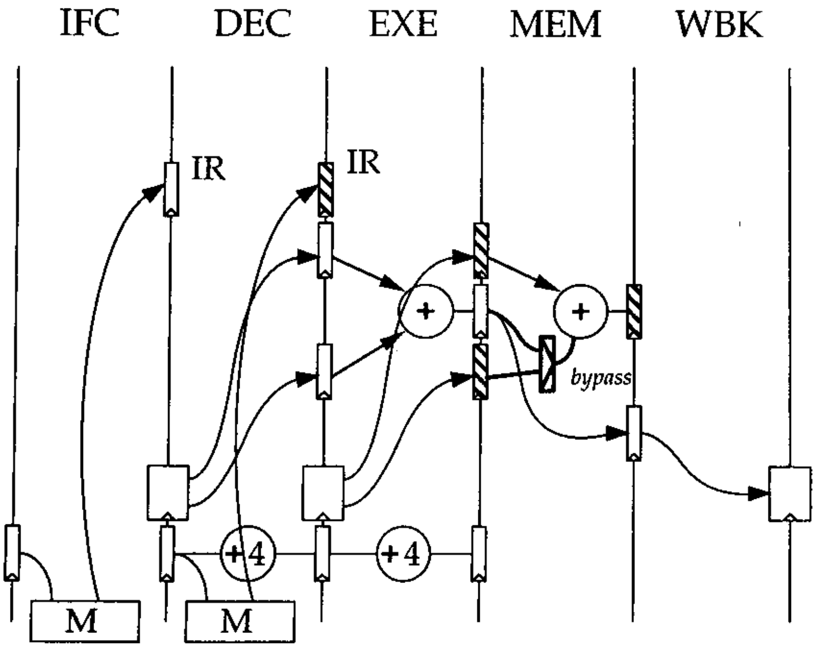


Implementation

Putting all together

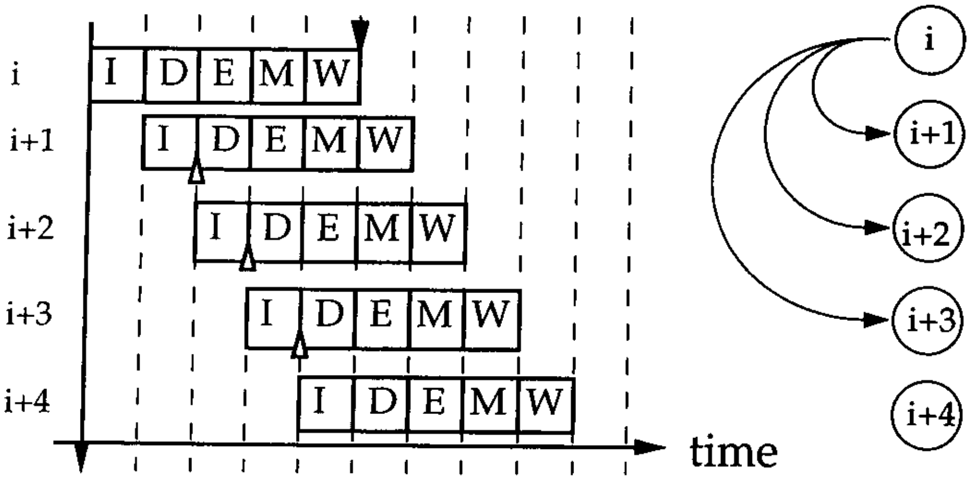
Add R3, R2, R1

Add R5, R4, R3



Implementation

How many data dependencies ?

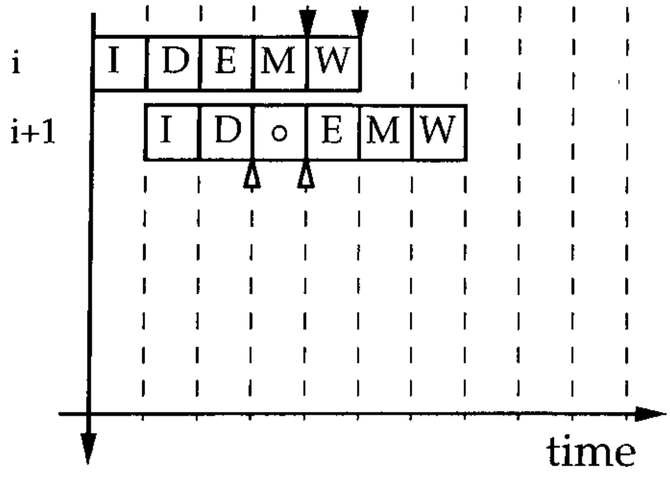


Implementation

All the data dependencies cannot be resolved by bypasses

Lw R3, I (R1)

Add R5, R4, R3



i	I	D	E	M	W		
i+1		I	D	o	E	M	W

Implementation

The hardware has to control the flow

through the pipeline

The problem of dependencies is the major limitation to deep pipelines