

**Modern Microprocessor  
Architectures:  
Evolution of RISC into Super-Scalars**

by

Prof. Vojin G. Oklobdzija

# Outline of the Talk

- 1 Definitions
- 1 Main features of RISC architecture
- 1 Analysis of RISC and what makes RISC
- 1 What brings performance to RISC
- 1 Going beyond one instruction per cycle
- 1 Issues in super-scalar machines
- 1 New directions

# What is Architecture ?

- 1 The first definition of the term “architecture” is due to Fred Brooks (Amdahl, Blaaw and Brooks 1964) while defining the IBM System 360.
- 1 Architecture is defined in the “principles of operation” which serves the programmer to write correct time independent programs, as well as to an engineer to implement the hardware which is to serve as an execution platform for those programs.
- 1 Strict separation of the architecture (definition) from the implementation details.

# How did RISC evolve ?

- 1 The concept emerged from the analysis of how the software actually uses resources of the processor ( trace tape analysis and instruction statistics - IBM 360/85)
- 1 The 90-10 rule: it was found out that a relatively small subset of the instructions (top 10) accounts for over 90% of the instructions used.
- 1 If addition of a new complex instruction increases the “critical path” (typically 12-18 gate levels) for one gate level, than the new instruction should contribute at least 6-8% to the overall performance of the machine.

# Main features of RISC

1 The work that each instruction performs is simple and straight forward:

# the time required to execute each instruction can be shortened and the number of cycles reduced.

# the goal is to achieve execution rate of one cycle per instruction (CPI=1.0)

# Main features of RISC

1 The instructions and the addressing modes are carefully selected and tailored upon the most frequently used ones.

1 Trade off:

$$\text{time (task)} = I \times C \times P \times T_0$$

I = no. of instructions / task

C = no. of cycles / instruction

P = no. of clock periods / cycle (usually P=1)

$T_0$  = clock period (nS)

# What makes architecture RISC ?

- 1 Load / Store : Register to Register operations, or decoupling of the operation and memory access.
- 1 Carefully Selected Set of Instructions implemented in hardware:
  - not necessarily small
- 1 Fixed format instructions (usually the size is also fixed)
- 1 Simple Addressing Modes
- 1 Separate Instruction and Data Caches: Harvard Architecture

# What makes architecture RISC ?

- 1 Delayed Branch instruction (Branch and Execute)\*  
also delayed Load
- 1 Close coupling of the compiler and the architecture:  
optimizing compiler
- 1 Objective of one instruction per cycle:  $CPI = 1$

] Pipelining

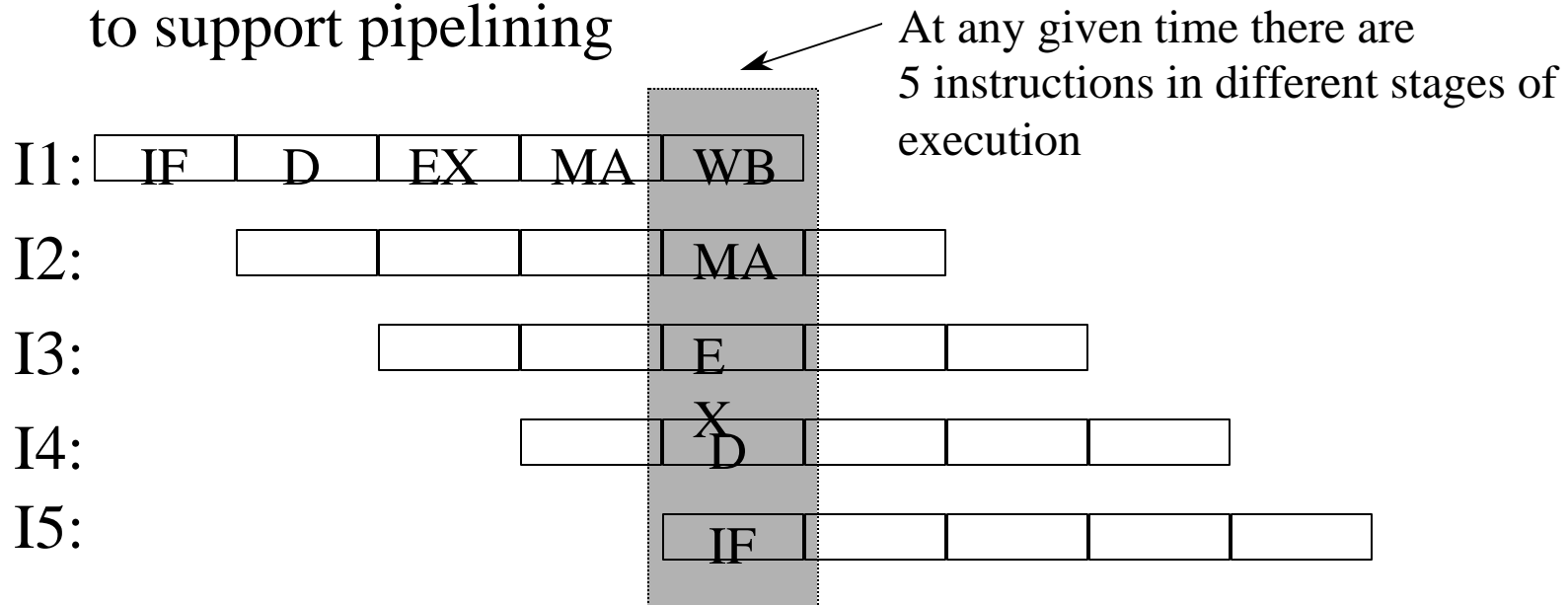
*\*no longer true of new designs*

# RISC: Features Revisited

- 1 Exploitation of Parallelism on the pipeline level is the key to the RISC Architecture

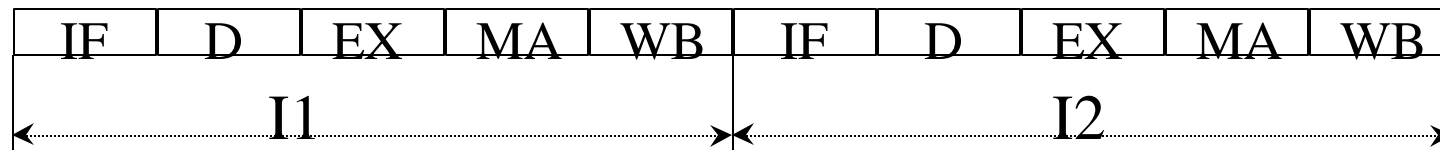
## Inherent parallelism in RISC

- 1 The main features of RISC architecture are there in order to support pipelining



# RISC: Features Revisited

- 1 Without pipelining the goal of  $CPI = 1$  is not achievable



Total of 10 cycles for two instructions

- 1 Degree of parallelism in the RISC machine is determined by the depth of the pipeline (maximal feasible)

# RISC: Carefully Selected Set of Instructions

## 1 Instruction selection criteria:

# only those instructions that fit into the pipeline structure are included

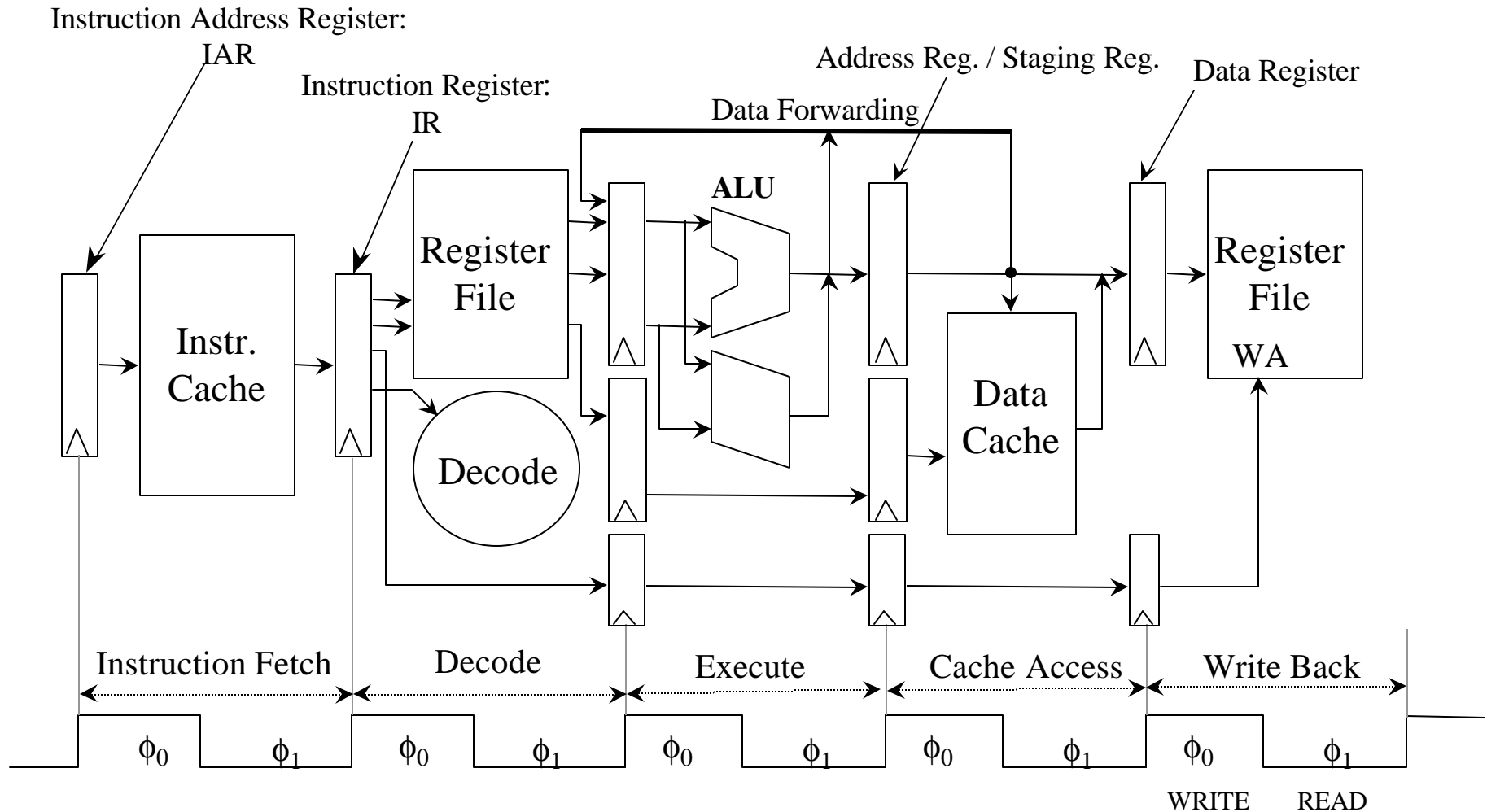
# the pipeline is derived from the core of the most frequently used instructions

Such derived pipeline must serve efficiently the three main classes of instructions:

- , Access to Cache (Load/Store)
- , Operation: Arithmetic/Logical
- , Branch



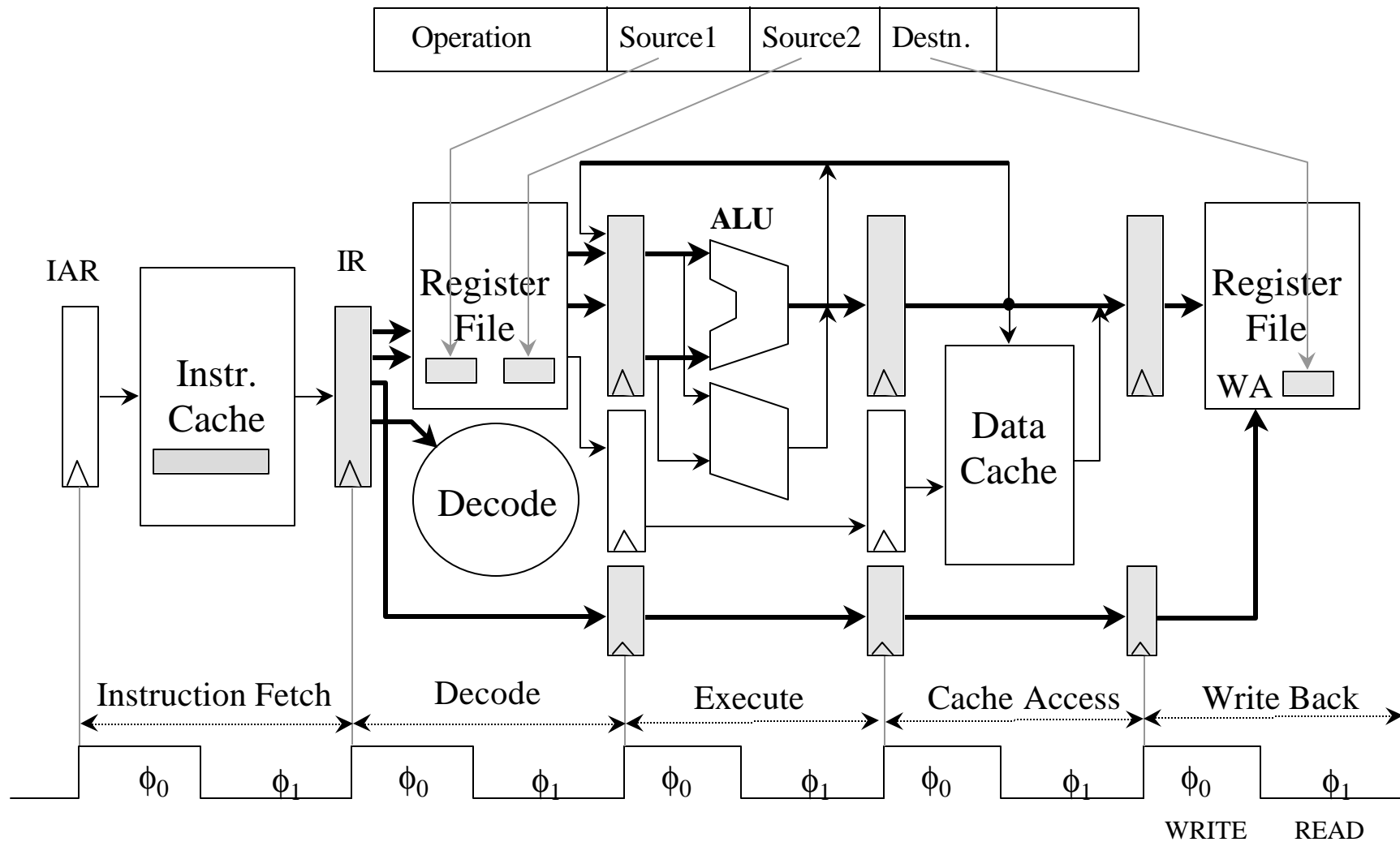
# Pipeline



# RISC: Support for the Pipeline

- 1 The instructions have fixed fields and are of the same size (usually 32-bits):
  - # This is necessary in order to be able to perform instruction decode in one cycle
  - # This feature is very valuable for super-scalar implementations (two sizes: 32 and 16-bit are seen, IBM-RT/PC)
  - # Fixed size instruction allow IF to be pipelined (know next address without decoding the current one). Guarantees only single I-TLB access per instruction.
- 1 Simple addressing modes are used: those that are possible in one cycle of the Execute stage (B+D, B+IX, Absolute) They also happen to be the most frequently used ones.

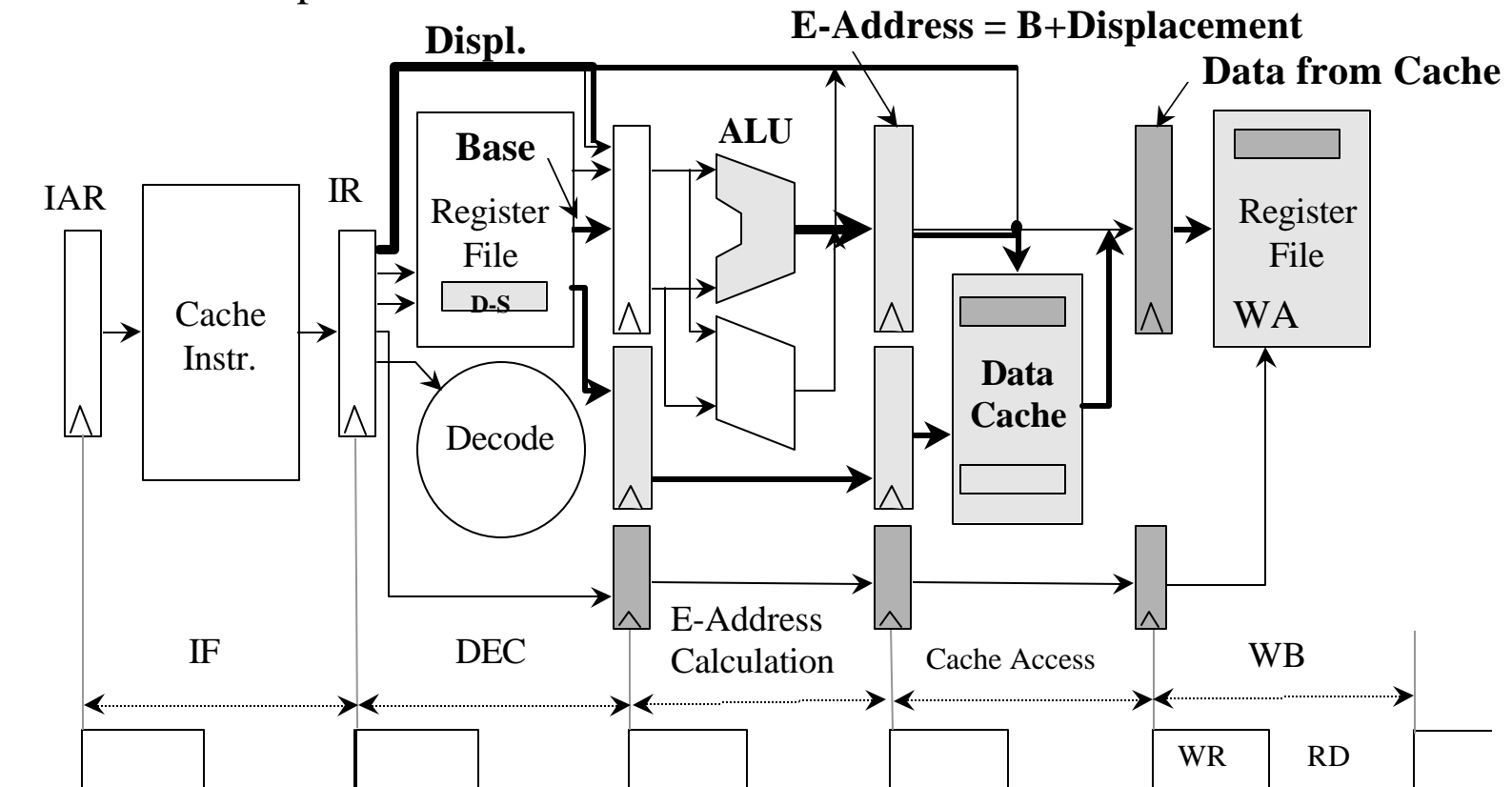
# RISC: Operation: Arithmetic/Logical



# RISC: Load (Store)

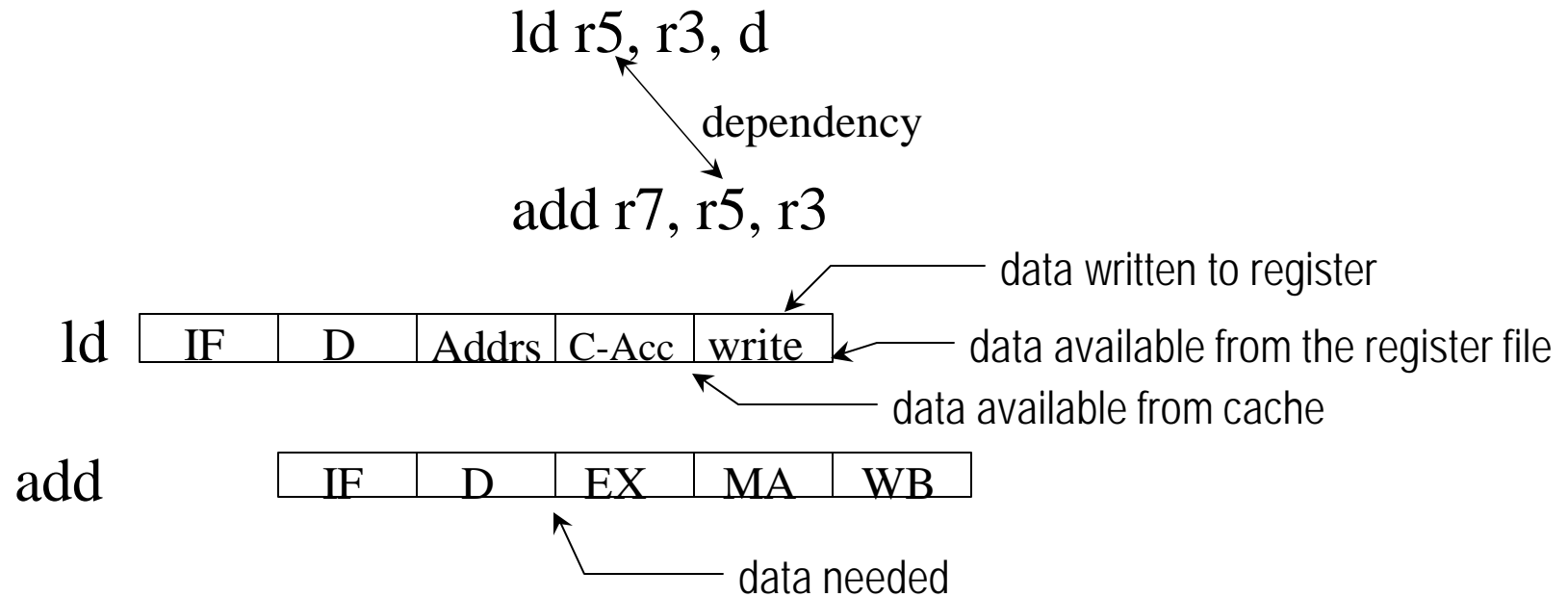
- 1 Decomposition of memory access (*unpredictable and multiple cycle operation*) from the operation (*predictable and fixed number of cycles*)

# RISC implies the use of caches



## RISC: Load (Store)

- 1 If Load is followed by an instruction that needs the data, one cycle will be lost:



- 1 Compiler “schedules” the load (moves it away from the instruction needing the data brought by load)
- 1 It also uses the “bypasses” (logic to forward the needed data) - they are known to the compiler.

## RISC: “Scheduled” Load - Example

**Program to calculate:**

$$A = B + C$$

$$D = E - F$$

*Sub-optimal:*

```
ld r2, B
ld r3, C
add r1, r2, r3
st r1, A
ld r2, E
ld r3, F
sub r1, r2, r3
st r1, F
```

*data dependency:  
one cycle lost*

*data dependency:  
one cycle lost*

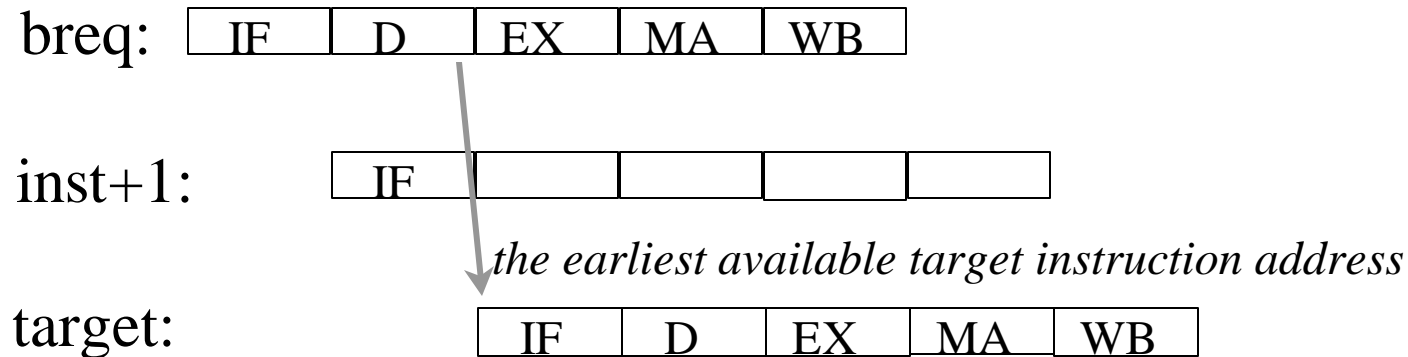
*Total = 10 cycles*

*Optimal:*

```
ld r2, B
ld r3, C
ld r4, E
add r1, r2, r3
ld r3, F
st r1, A
sub r1, r4, r3
st r1, F
```

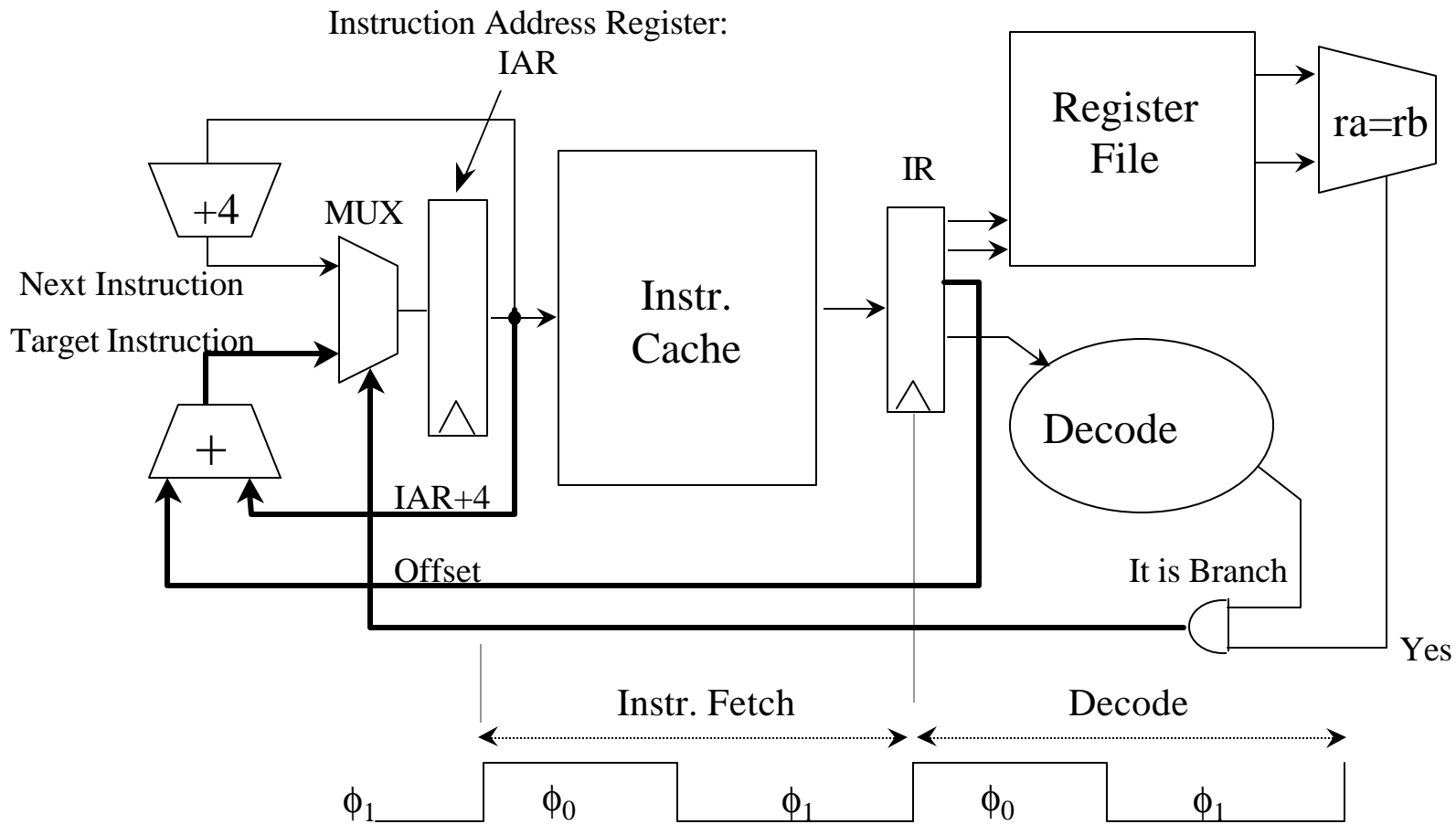
*Total = 8 cycles*

# RISC: Branch



- 1 In order to minimize the number of lost cycles, Branch has to be resolved during Decode stage. This requires a separate address adder as well as comparator which are used during Decode stage.
- 1 In the best case one cycle will be lost when Branch instruction is encountered. (this slot is used for an independent instruction which is scheduled in this slot - “branch and execute”)

# RISC: Branch



# RISC: “Branch and Execute”

- 1 One of the most useful instruction defined in RISC architecture (it amounts to up to 15% increase in performance) (also known as “delayed branch”)
- 1 Compiler has an intimate knowledge of the pipeline (violation of the architecture principle, the machine is defined as “visible through the compiler”)
- 1 Branch and Execute fills the empty instruction slot with:
  - # an independent instruction before the Branch
  - # instruction from the target stream (that will not change the state)
  - # instruction from the fail pathIt is possible to fill up to 70% of empty slots (Patterson-Hennesey)

# RISC: “Branch and Execute” - Example

**Program to calculate:**

$a = b + 1$

if (c=0) d = 0

*Sub-optimal:*

```
ld r2, b    # r2=b    ↘ load stall
add r2, 1   # r2=b+1
st r2, a    # a=b+1
ld r3, c    # r3=c    ↘ load stall
bne r3,0, tg1 # skip  ↙
st 0, d     # d=0     ← lost cycle
```

tg1: .....

*Total = 9 cycles*

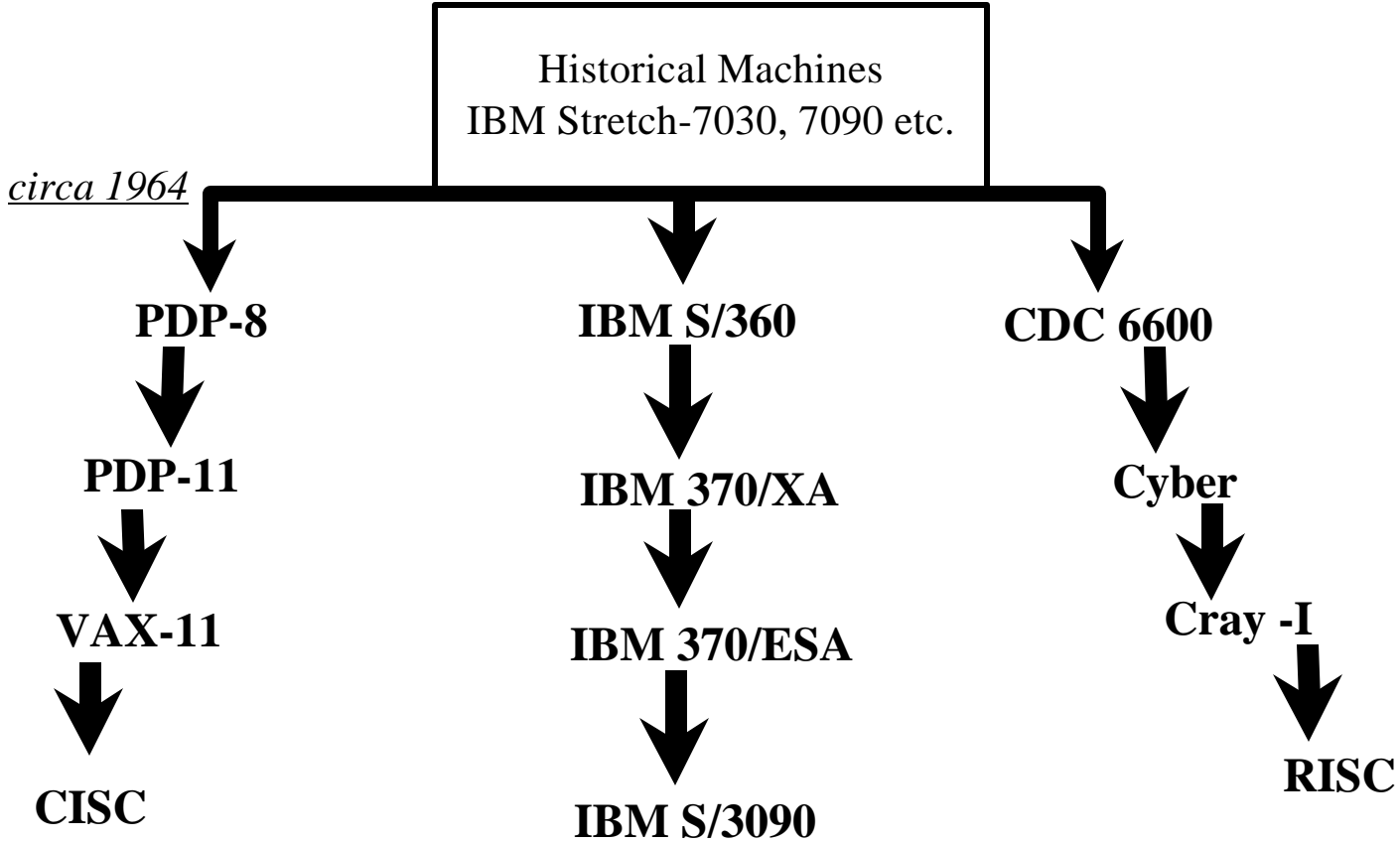
*Optimal:*

```
ld r2, b    # r2=b
ld r3, c    # r3=c
add r2, 1   # r2=b+1
bne r3,0, tg1 # skip
st r2, a    # a=b+1
st 0, d     # d=0
```

tg1: .....

*Total = 6 cycles*

# A bit of history



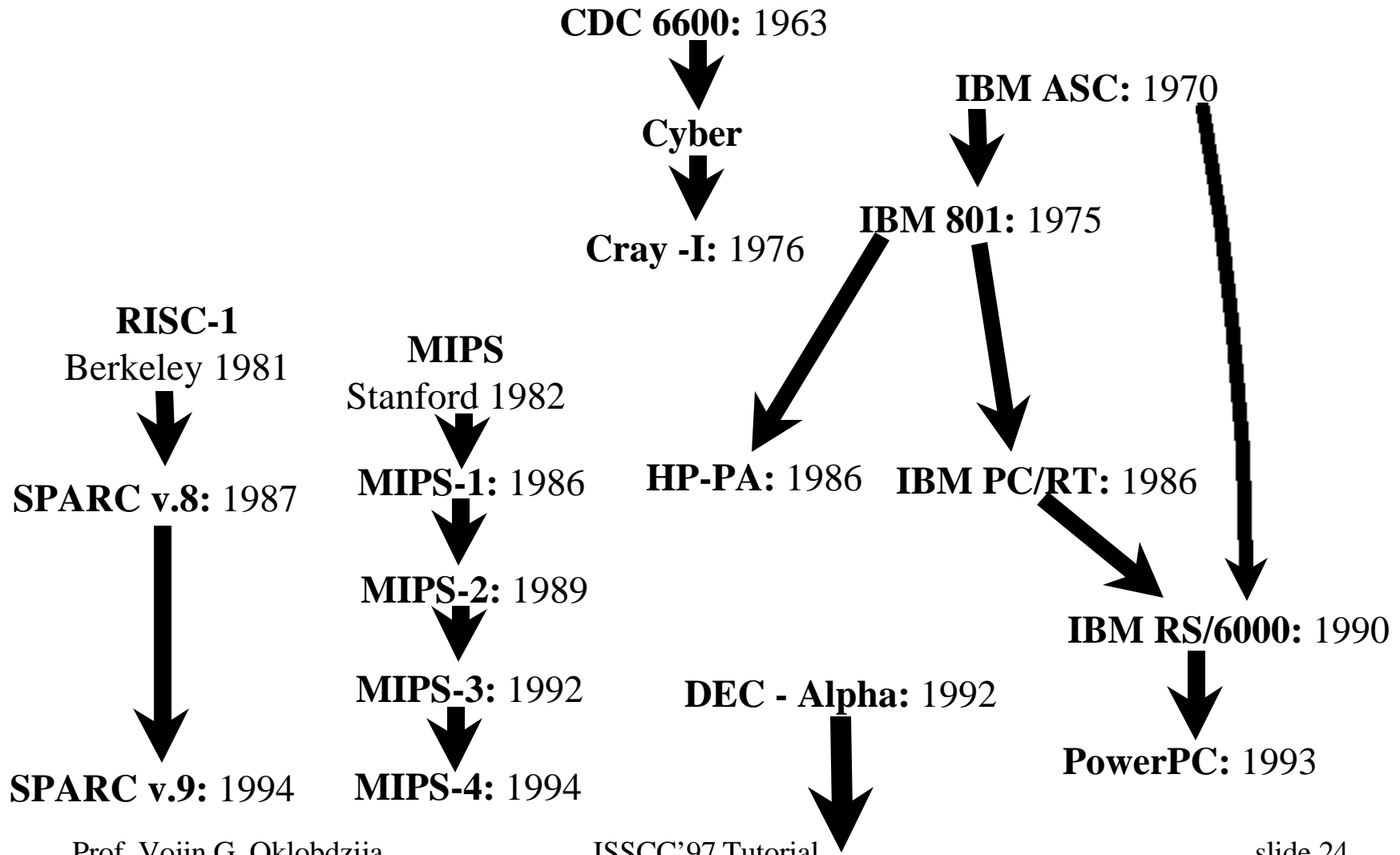
# Important Features Introduced

- 1 Separate Fixed and Floating point registers (IBM S/360)
- 1 Separate registers for address calculation (CDC 6600)
- 1 Load / Store architecture (Cray-I)
- 1 Branch and Execute (IBM 801)


## Consequences:

- # Hardware resolution of data dependencies (Scoreboarding CDC 6600, Tomasulo's Algorithm IBM 360/91)
- # Multiple functional units (CDC 6600, IBM 360/91)
- # Multiple operation within the unit (IBM 360/91)

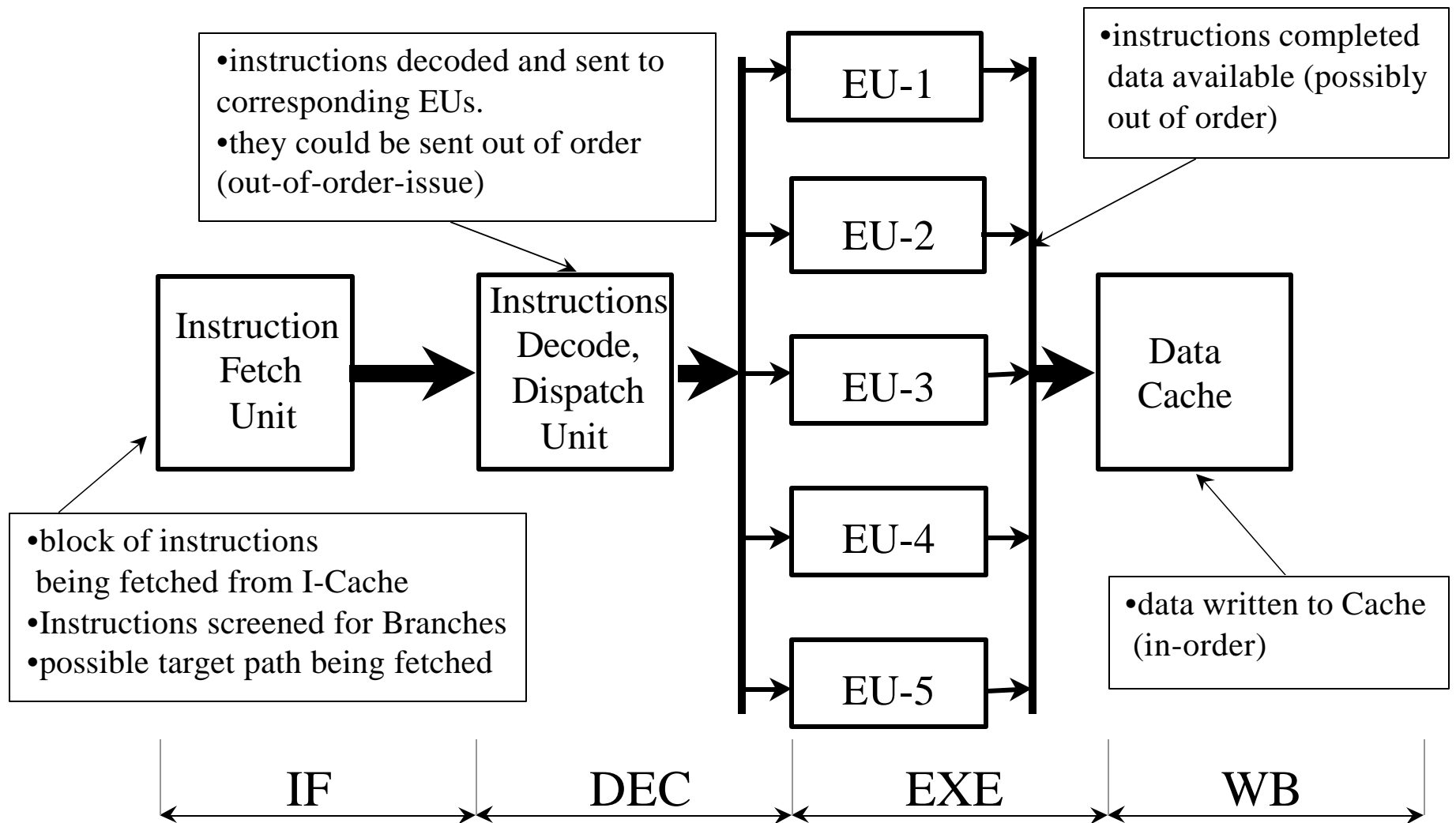
# RISC: History



# Reaching beyond the CPI of one: The next challenge

- 1 With the perfect caches and no lost cycles in the pipeline  
the CPI  1.00
- 1 The next step is to break the 1.0 CPI barrier and go beyond
- 1 How to efficiently achieve more than one instruction per  
cycle ?
- 1 Again the key is exploitation of parallelism:
  - # on the level of independent functional units
  - # on the pipeline level

# How does super-scalar pipeline look like ?



# Super-scalar Pipeline

- 1 One pipeline stage in super-scalar implementation may require more than one clock. Some operations may take several clock cycles.
- 1 Super-Scalar Pipeline is much more complex - therefore it will generally run at lower frequency than single-issue machine.
- 1 The trade-off is between the ability to execute several instructions in a single cycle and a lower clock frequency (as compared to scalar machine).

- *“Everything you always wanted to know about computer architecture can be found in IBM 360/91”*

Greg Grohowsky, Chief Architect of IBM RS/6000

# Super-scalar Pipeline (cont.)

IBM 360/91 pipeline

IBM 360/91 reservation table

# Deterrents to Super-scalar Performance

- 1 The cycle lost due to the Branch is much costlier in case of super-scalar. The RISC techniques do not work.
- 1 Due to several instructions being concurrently in the Execute stage data dependencies are more frequent and more complex
- 1 Exceptions are a big problem (especially precise)
- 1 Instruction level parallelism is limited

# Super-scalar Issues

- 1 contention for resources
  - # to have sufficient number of available hardware resources
- 1 contention for data
- 1 synchronization of execution units
  - # to insure program consistency with correct data and in correct order
- 1 to maintain sequential program execution with several instructions in parallel
- 1 design high-performance units in order to keep the system balanced

# Super-scalar Issues

## 1 Low Latency:

# to keep execution busy while Branch Target is being fetched  
requires one cycle I-Cache

## 1 High-Bandwidth:

# I-Cache must match the execution bandwidth (4-instructions issued  
IBM RS/6000, 6-instructions Power2, PowerPC620)

## 1 Scanning for Branches:

# scanning logic must detect Branches in advance (in the IF stage)

*The last two features mean that the I-Cache bandwidth must be greater than the raw bandwidth required by the execution pipelines. There is also a problem of fetching instructions from multiple cache lines.*

# Super-Scalars: Handling of a Branch

## RISC Findings:

### 1 BEX - Branch and Execute:

the subject instruction is executed whether or not the Branch is taken

# we can utilize:

(1) subject instruction (2) an instruction from the target (3) an instruction from the “fail path”

## Drawbacks:

### 1 Architectural and implementation:

# if the subject instruction causes an interrupt, upon return branch may be taken or not. If taken Branch Target Address must be remembered.

# this becomes especially complicated if multiple subject instructions are involved

# efficiency: 60% in filling execution slots

# Super-Scalars: Handling of a Branch

Classical challenge in computer design:

*In a machine that executes several instructions per cycle the effect of Branch delay is magnified. The objective is to achieve zero execution cycles on Branches.*

- 1 Branch typically proceed through the execution consuming at least one pipeline cycle (most RISC machines)
- 1 In the n-way Super-Scalar one cycle delay results in n-instructions being stalled.
- 1 Given that the instructions arrive n-times faster - the frequency of Branches in the Decode stage is n-times higher
- Ⓜ Separate Branch Unit required
- Ⓜ Changes made to decouple Branch and Fixed Point Unit(s) must be introduced in the architecture

# Super-Scalars: Handling of a Branch

## Conditional Branches:

1 Setting of the Condition Code (a troublesome issue)

1 Branch Prediction Techniques:

- # Based on the OP-Code

- # Based on Branch Behavior (loop control usually taken)

- # Based on Branch History (uses Branch History Tables)

- # Branch Target Buffer (small cache, storing Branch Target Address)

- # Branch Target Tables - BTT (IBM S/370): storing Branch Target instruction and the first several instructions following the target

- # Look-Ahead resolution (enough logic in the pipeline to resolve branch early)

# Techniques to Alleviate Branch Problem\*

## Loop Buffers:

- 1 Single-loop buffer
- 1 Multiple-loop buffers (n-sequence, one per buffer)

## Machines:

- # CDC Star-100: loop buffer of 256 bytes
- # CDC 6600: 60 bytes loop buffer
- # CDC 7600: 12 60-bit words
- # CRAY-I: four loop buffers, content replaced in FIFO manner (similar to 4-way associative I-Cache)

[\*Lee, Smith, "Branch Prediction Strategies and Branch Target Buffer Design", Computer January, 1984.]

# Techniques to Alleviate Branch Problem

## 1 Following Multiple Instruction Streams

### Problems:

- # BT cannot be fetched until BTA is determined (requires computation time, operands may not be available)
- # Replication of initial stages of the pipeline: additional branch requires another path:
  - , for a typical pipeline more than two branches need to be processed to yield improvement.
  - , hardware required makes this approach impractical
- # Cost of replicating significant part of the pipeline is substantial.

### Machines that Follow multiple I-streams:

IBM 370/168 (fetches one alternative path), IBM 3033 (pursues two alternative streams)

# Techniques to Alleviate Branch Problem

## Prefetch Branch Target:

### 1 Duplicate enough logic to prefetch branch target

# If taken, target is loaded immediately into the instruction decode stage

Several prefetches are accumulated along the main path

The IBM 360/91 uses this mechanism to prefetch a double-word target.

# Techniques to Alleviate Branch Problem

## Look-Ahead Resolution:

1 Placing extra logic in the pipeline so that branch can be detected and resolved at the early stage:

# Whenever the condition code affecting the branch has been determined

(“Zero-Cycle Branch”, “Branch Folding”)

1 This technique was used in IBM RS/6000:

# Extra logic is implemented in a separate Branch Execution Unit to scan through the I-Buffer for Branches and to:

(1) Generate BTA, (2) determine the BR outcome if possible and if not (3) dispatch the instruction in conditional fashion

# Techniques to Alleviate Branch Problem

## Branch Behavior:

### Types of Branches:

• Loop-Control: *usually taken, backward*

• If-then-else: *forward, not consistent*

• Subroutine Calls: *always taken*

Just by predicting that the Branch is taken we are guessing right 60-70% of the time [Lee,Smith](67% of the time, [Patterson-Hennessy])

# Techniques to Alleviate Branch Problem: Branch prediction

## Prediction Based on Direction of the Branch:

• Forward Branches are taken 60% of the time, backward branches 85% of the time [Patterson-Hennessy]

## Based on the OP-Code:

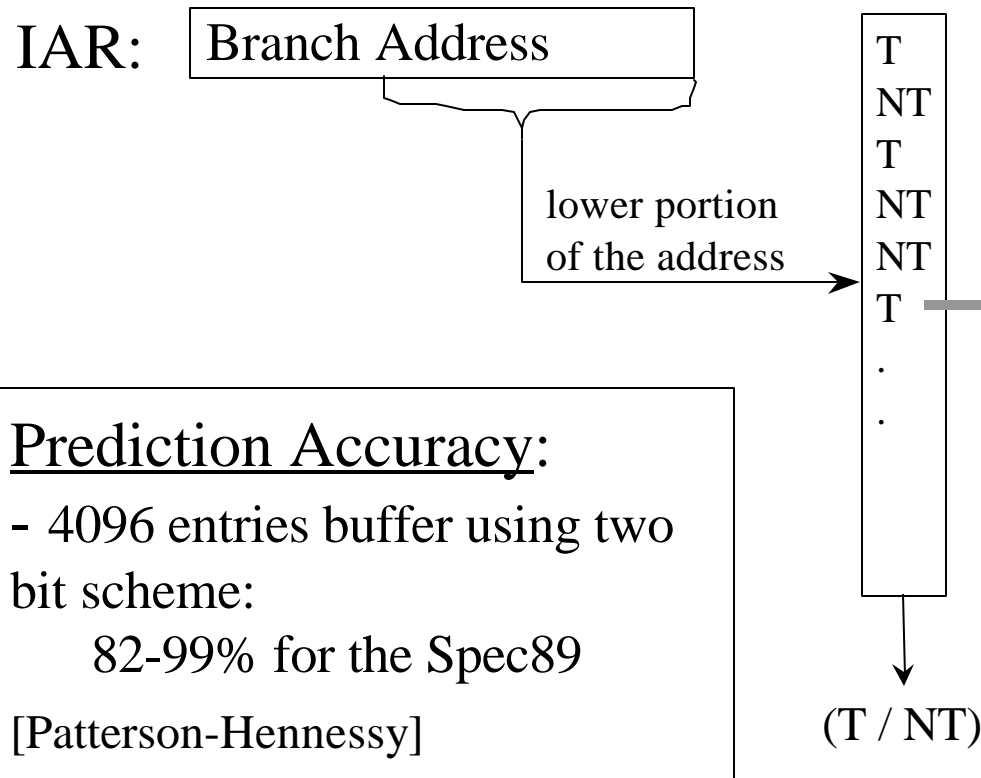
• Combined with the always taken guess (60%) the information on the opcode can raise the prediction to: 65.7-99.4% [J. Smith]

• In IBM CPL mix always taken is 64% of the time true, combined with the opcode information the prediction accuracy rises to 66.2%

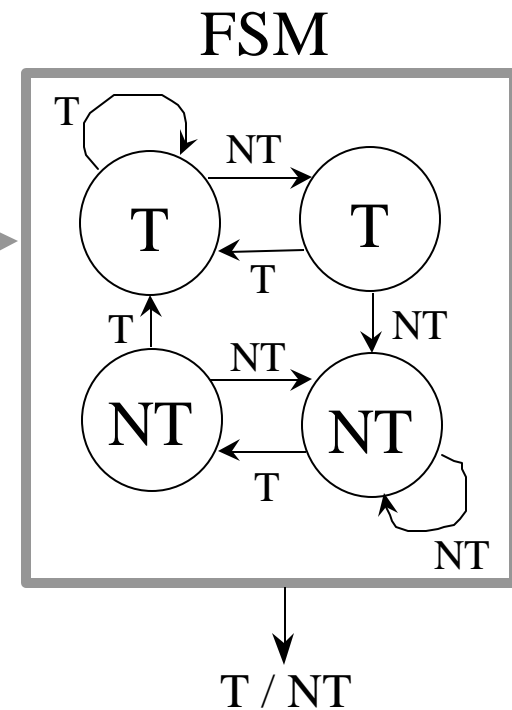
*The prediction based on the OP-Code is much lower than the prediction based on branch history.*

# Techniques to Alleviate Branch Problem: Branch prediction

## Prediction Based on Branch History:

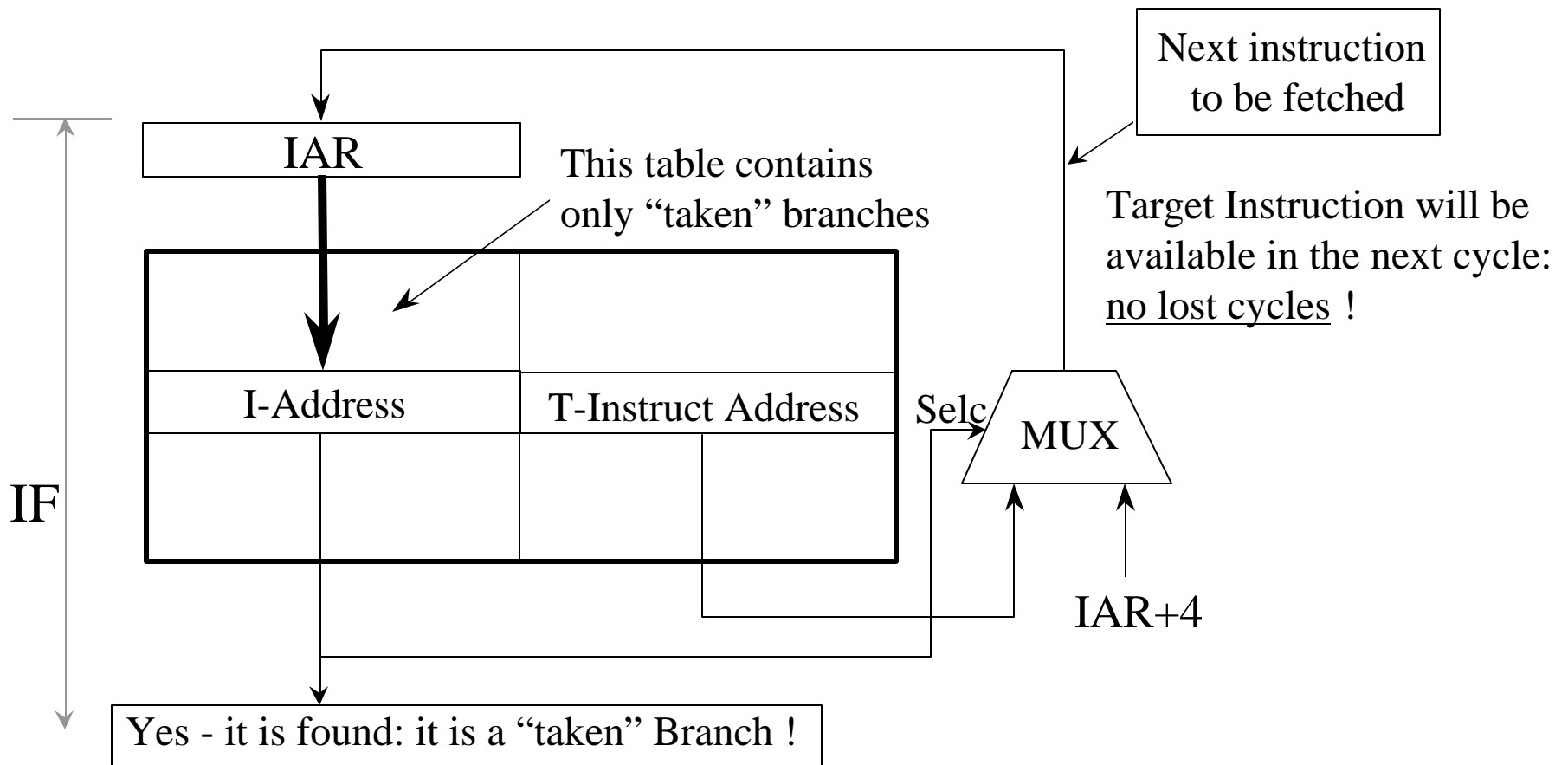


Two-bit prediction scheme based on Branch History



# Techniques to Alleviate Branch Problem: Branch prediction

## Prediction Using Branch Target Buffer (BTB):



# Techniques to Alleviate Branch Problem: Branch prediction

## Difference between Branch Prediction and Branch Target

### Buffer:

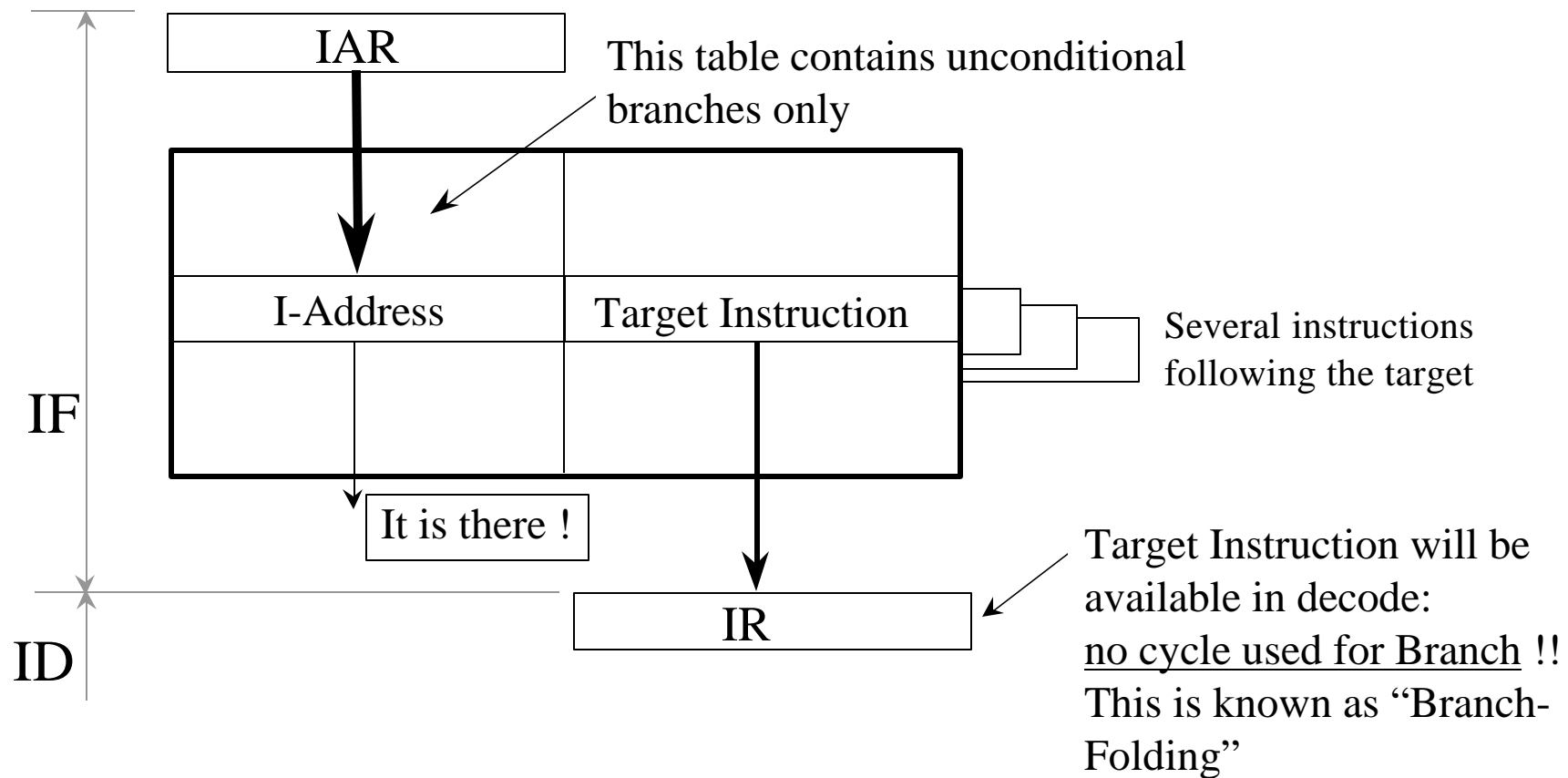
ÿ In case of Branch Prediction the decision will be made during Decode stage - thus, even if predicted correctly the Target Instruction will be late for one cycle.

ÿ In case of Branch Target Buffer, if predicted correctly, the Target Instruction will be the next one in line - no cycles lost.

*(if predicted incorrectly - the penalty will be two cycles in both cases)*

# Techniques to Alleviate Branch Problem: Branch prediction

## Prediction Using Branch Target Table (BTT):



# Techniques to Alleviate Branch Problem: Branch prediction

## Branch Target Buffer Effectiveness:

- ÿ BTB is purged when address space is changed  
(multiprogramming)
- ÿ 256 entry BTB has a hit ratio of 61.5-99.7% (IBM/CPL).
  - # prediction accuracy 93.8%
  - # Hit ratio of 86.5% obtained with 128 sets of four entries
  - # 4.2% incorrect due to the target change

overall accuracy =  $(93.8 - 4.2) \times 0.87 = 78\%$

- ÿ BTB yields overall 5-20% performance improvement

# Techniques to Alleviate Branch Problem: Branch prediction

## IBM RS/6000:

Statistic from 801 shows:

20% of all FXP instructions are Branches:

- # *1/3 of all the BR are unconditional (potential “zero cycle”)*
- # *1/3 of all the BR are used to terminate DO loop (“zero cycle”)*
- # *1/3 of all the BR are conditional: they have 50-50 outcome*

Unconditional and loop terminate branches (BCT instruction introduced in RS/6000) are “zero-cycle”, therefore:

*Branch Penalty =  $2/3 \times 0 + 1/6 \times 0 + 1/6 \times 2 = 0.33$  cycles for branch on the average*

# Techniques to Alleviate Branch Problem: Branch prediction

## IBM PowerPC 620:

- ÿ IBM RS/6000 did not have “Branch Prediction”. The penalty of 0.33 cycles for Branch seems to high. It was found that “prediction” is effective and not so difficult to implement.
  - # A 256-entry, two-way set associative BTB is used to predict the next fetch address, first.
  - # A 2048-entry Branch Prediction Buffer (BHT) used when the BTB does not hit but the Branch is present.
  - # Both BTB and BHT are updated, if necessary.
- ÿ There is a stack of return address registers used to predict subroutine returns.

# Techniques to Alleviate Branch Problem: *Contemporary Microprocessors*

## DEC Alpha 21264:

- 1 Two forms of prediction and dynamic selection of better one

## MIPS R10000:

- 1 Two bit Branch History Table and Branch Stack to restore misses.

## HP 8000:

- 1 32-entry BTB (fully associative) and 256 entry Branch History Table

## Intel P6:

- 1 Two-level adaptive branch prediction

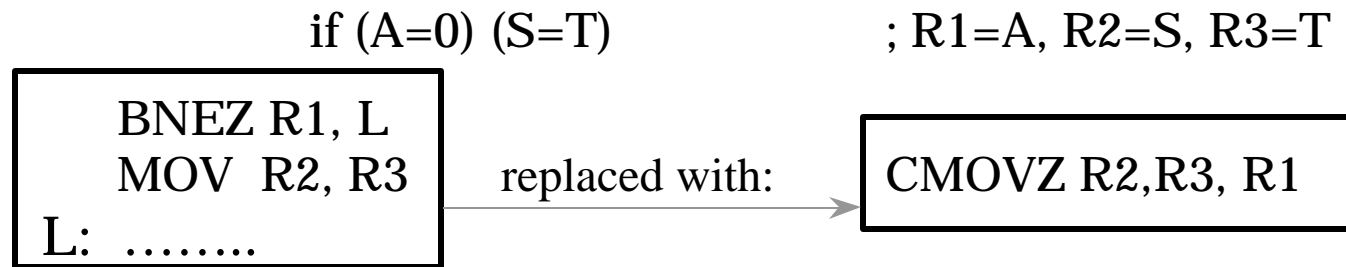
## Exponential:

- 1 256-entry BTB, 2-bit dynamic history, 3-5 cycle misspredict penalty

# Techniques to Alleviate Branch Problem: *How can the Architecture help ?*

## ÿ Conditional or Predicated Instructions

Useful to eliminate BR from the code. If condition is *true* the instruction is executed normally if *false* the instruction is treated as NOP:



## ÿ Loop Closing instructions: BCT (Branch and Count, IBM RS/6000)

The loop-count register is held in the Branch Execution Unit - therefore it is always known in advance if BCT will be taken or not (loop-count register becomes a part of the machine status)

# Super-scalar Issues: *Contention for Data*

## *Data Dependencies:*

### 1 Read-After-Write (RAW)

# also known as: *Data Dependency* or *True Data Dependency*

### 1 Write-After-Read (WAR)

# known as: *Anti Dependency*

### 1 Write-After-Write (WAW)

# known as: *Output Dependency*

WAR and WAW also known as: *Name Dependencies*

# Super-scalar Issues: *Contention for Data*

## True Data Dependencies: Read-After-Write (RAW)

An instruction  $j$  is data dependent on instruction  $i$  if:

- , Instruction  $i$  produces a result that is used by  $j$ , or
- , Instruction  $j$  is data dependent on instruction  $k$ , which is data dependent on instruction  $I$

*Examples*\*:

```
SUBI R1, R1, 8 ;decrement pointer  
BNEZ R1, Loop ; branch if R1 != zero
```

```
LD F0, 0(R1) ;F0=array element  
ADDD F4, F0, F2 ;add scalar in F2  
SD 0(R1), F4 ; store result F4
```

\*[Patterson-Hennessy]

# Super-scalar Issues: *Contention for Data*

## True Data Dependencies:

Data Dependencies are property of the program. The presence of dependence indicates the potential for hazard, which is a property of the pipeline (including the length of the stall)

## A Dependence:

- , indicates the possibility of a hazard
- , determines the order in which results must be calculated
- , sets the upper bound on how much parallelism can possibly be exploited.

*i.e. we can not do much about True Data Dependencies in hardware. We have to live with them.*

# Super-scalar Issues: *Contention for Data*

*Name Dependencies* are:

## 1 Anti-Dependencies ( Write-After-Read, WAR)

Occurs when instruction  $j$  writes to a location that instruction  $i$  reads, and  $i$  occurs first.

## 1 Output Dependencies (Write-After-Write, WAW)

Occurs when instruction  $i$  and instruction  $j$  write into the same location. The ordering of the instructions (write) must be preserved. ( $j$  writes last)

*In this case there is no value that must be passed between the instructions. If the name of the register (memory) used in the instructions is changed, the instructions can execute simultaneously or be reordered.*

The hardware CAN do something about *Name Dependencies* !

# Super-scalar Issues: *Contention for Data*

## Name Dependencies:

### 1 Anti-Dependencies ( Write-After-Read, WAR)

ADDD F4, F0, F2 ; F0 used by ADDD

LD F0, 0(R1) ; F0 not to be changed before read by ADDD

### 1 Output Dependencies (Write-After-Write, WAW)

LD F0, 0(R1) ;LD writes into F0

ADDD F0, F4, F2 ; Add should be the last to write into F0

*This case does not make much sense since F0 will be overwritten, however this combination is possible.*

Instructions with name dependencies can execute simultaneously if reordered, or if the name is changed. This can be done: *statically* (by compiler) or *dynamically* by the hardware

# Super-scalar Issues: *Dynamic Scheduling*

## 1 Thornton Algorithm (Scoreboarding): CDC 6600 (1964)

☛ *One common unit: Scoreboard which allows instructions to execute out of order, when resources are available and dependencies are resolved.*

## 1 Tomasulo's Algorithm: IBM 360/91 (1967)

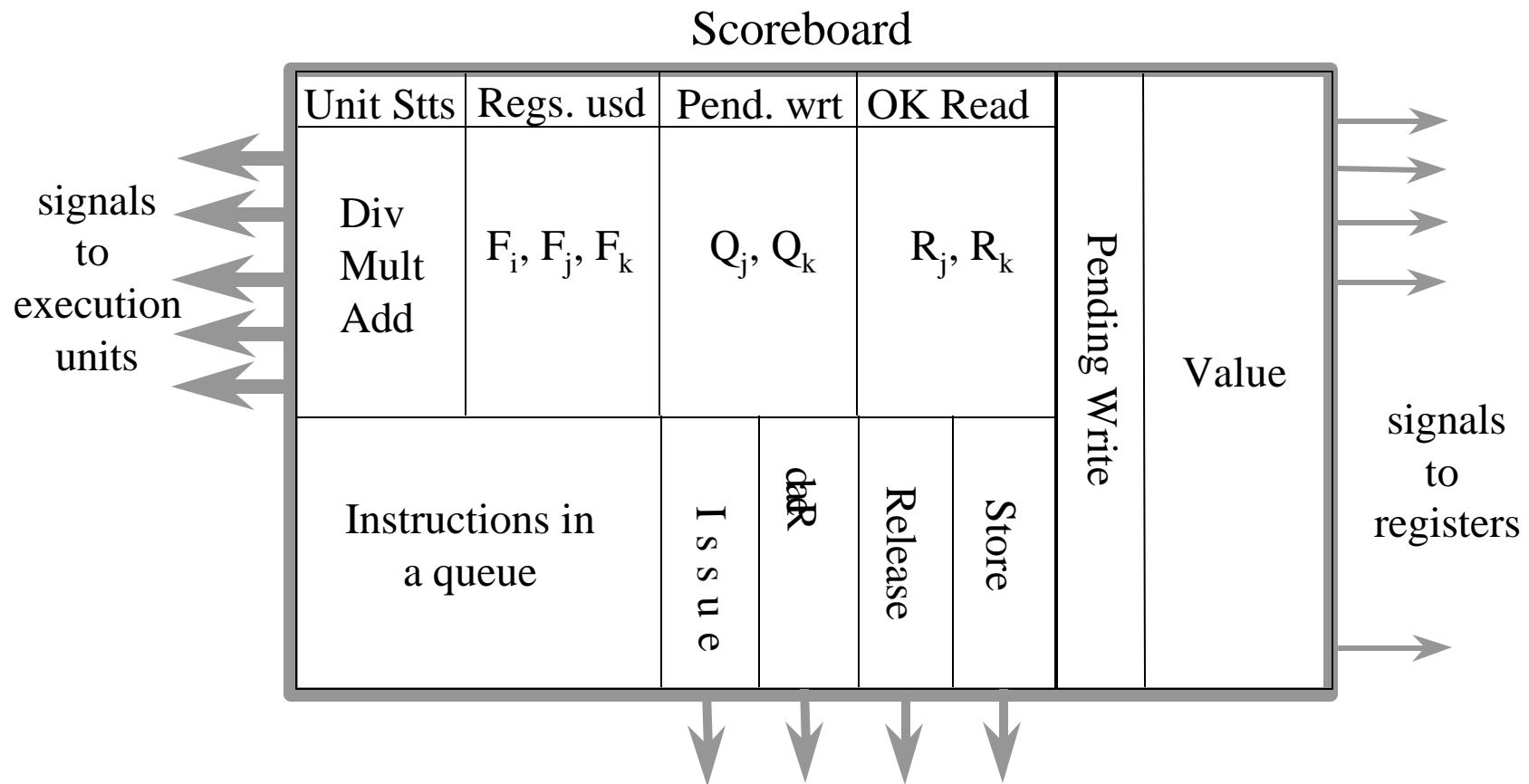
☛ *Reservation Stations used to buffer the operands of instructions waiting to issue and to store the results waiting for the register. Common Data Buss (CDB) used to distribute the results directly to the functional units.*

## 1 Register-Renaming: IBM RS/6000 (1990)

☛ *Implements more physical registers than logical (architect). They are used to hold the data until the instruction commit.*

# Super-scalar Issues: *Dynamic Scheduling*

Thornton Algorithm (Scoreboarding): CDC 6600



# Super-scalar Issues: *Dynamic Scheduling*

Thornton Algorithm (Scoreboarding): CDC 6600 (1964)

## Performance:

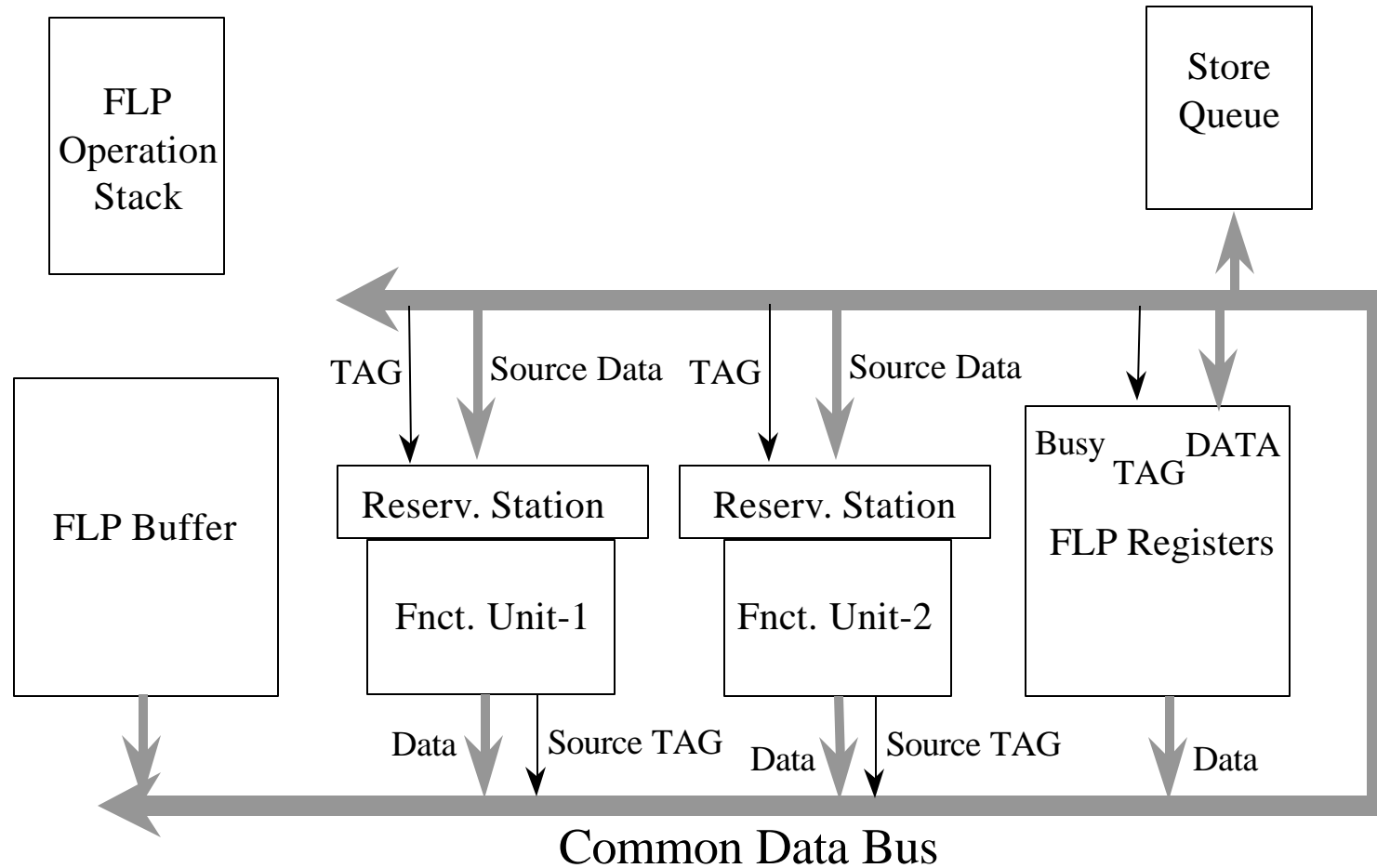
*CDC6600 was 1.7 times faster than CDC6400 (no scoreboard, one functional unit) for FORTRAN and 2.5 faster for hand coded assembly*

## Complexity:

*To implement the “scoreboard” as much logic was used as to implement one of the ten functional units.*

# Super-scalar Issues: *Dynamic Scheduling*

Tomasulo's Algorithm: IBM 360/91 (1967)



# Super-scalar Issues: *Dynamic Scheduling*

Tomasulo's Algorithm: IBM 360/91 (1967)

The key to Tomasulo's algorithm are:

## 1 Common Data Bus (CDB)

CDB carries the data and the TAG identifying the source of the data

## 1 Reservation Station

# Reservation Station buffers the operation and the data (if available) awaiting the unit to be free to execute. If data is not available it holds the TAG identifying the unit which is to produce the data. The moment this TAG is matched with the one on the CDB the data is taken and the execution will commence.

# Replacing register names with TAGs "name dependencies" are resolved. (sort of "register-renaming")

# Super-scalar Issues: *Dynamic Scheduling*

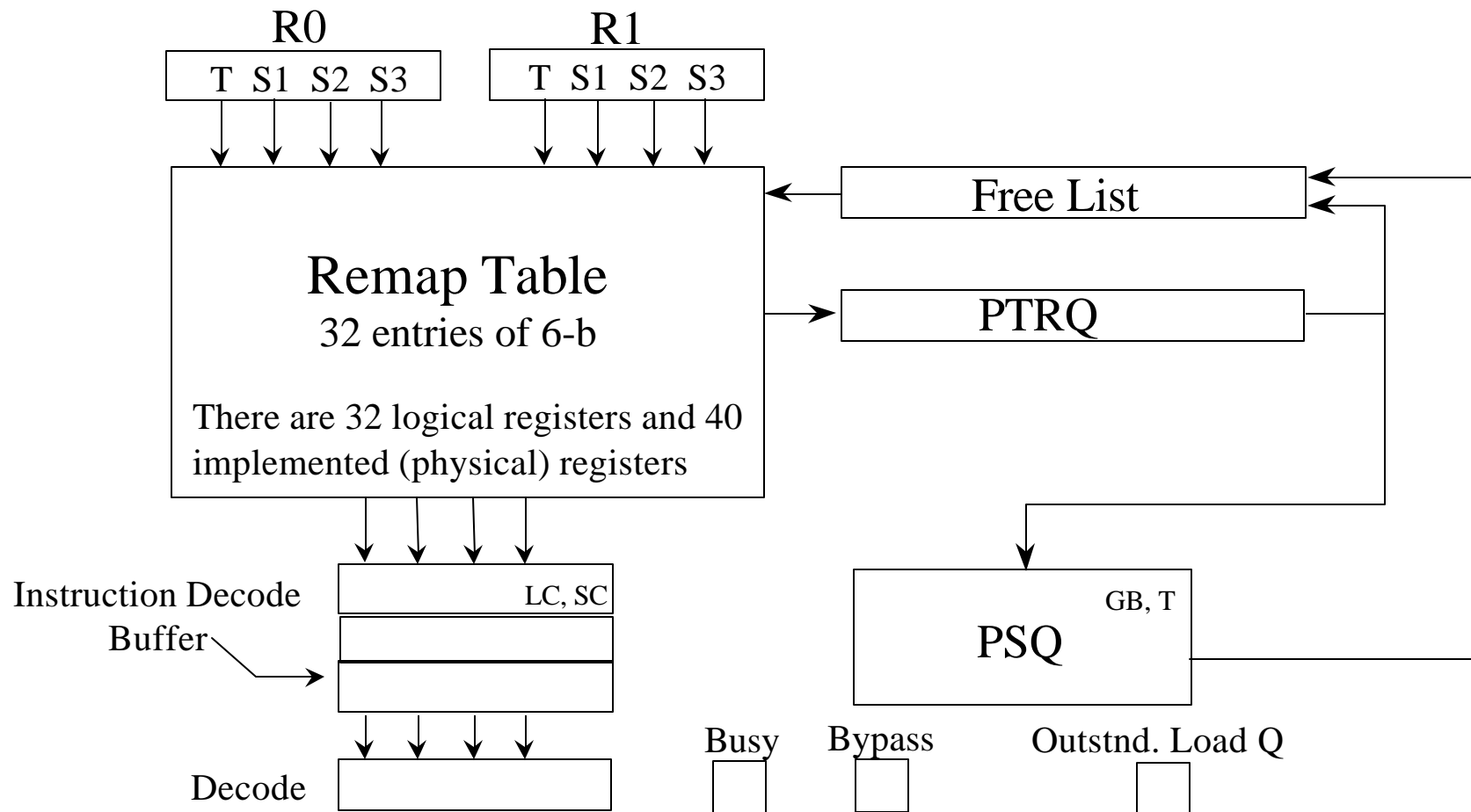
Register-Renaming: IBM RS/6000 (1990)

## Consist of:

- 1 Remap Table (RT): providing mapping from logical to physical register
- 1 Free List (FL): providing names of the registers that are unassigned - so they can go back to the RT
- 1 Pending Target Return Queue (PTRQ): containing physical registers that are used and will be placed on the FL as soon as the instruction using them pass decode
- 1 Outstanding Load Queue (OLQ): containing registers of the next FLP load whose data will return from the cache. It stops instruction from decoding if data has not returned

# Super-scalar Issues: *Dynamic Scheduling*

Register-Renaming Structure: IBM RS/6000 (1990)



# Power of Super-scalar Implementation

## Coordinate Rotation: IBM RS/6000 (1990)

```
FL FR0, sin theta      ;load rotation matrix
FL FR1, -sin theta     ;constants
FL FR2, cos theta     ;
FL FR3, xdis           ;load x and y
FL FR4, ydis           ;displacements
MTCTR I                ;load Count register with loop count

LOOP:  UFL FR8, x(i)    ;load x(i)
       FMA FR10, FR8, FR2, FR3 ;form x(i)cos + xdis
       UFL FR9, y(i)   ;load y(i)
       FMA FR11, FR9, FR2, FR4 ;form y(i)cos + ydis
       FMA FR12, FR9, FR1, FR10 ;form -y(i)sin + FR10
       FST FR12, x1(i) ;store x1(i)
       FMA FR13, FR8, FR0, FR11 ;form x(i)sin + FR11
       FST FR13, y1(i) ;store y1(i)
       BC LOOP        ;continue for all points
```

$$\begin{aligned}x1 &= x \cos\theta - y \sin\theta \\y1 &= y \cos\theta + x \sin\theta\end{aligned}$$

***This code, 18 instructions worth, executes in 4 cycles in a loop***

# Super-scalar Issues: *Dynamic Scheduling*

Register-Renaming: IBM RS/6000 (1990)

How does it work ?

Arithmetic:

- 1 5-bit register field replaced by a 6-bit physical register field instruction (40 physical registers)
- 1 New instruction proceeds to IDB or Decode (if available)
- 1 Once in Decode compare w/BSY, BP or OLQ to see if register is valid
- 1 After being released from decode
  - # the SC increments PSQ to release stores
  - # the LC increments PTRQ to release the registers to the FL (as long as there are no Stores using this register - compare w/ PSQ)

# Super-scalar Issues: *Dynamic Scheduling*

Register-Renaming: IBM RS/6000 (1990)

## How does it work ?

### Store:

- 1 Target is renamed to physical register and ST is executed in parallel
- 1 ST is placed on PSQ until value of the register is available. Before leaving REN the SC of the most recent instruction prior to it is incremented. (that could have been the instruction that generates the result)
- 1 When ST reaches a head of PSQ the register is compared with BYS and OLQ before executed
- 1 GB is set, tag returned to FL, FXP uses ST data buffer for the address

# Super-scalar Issues: *Dynamic Scheduling*

Register-Renaming: IBM RS/6000 (1990)

How does it work ?

Load:

- 1 Defines a new semantic value, causing REN to be updated
- 1 REN table is accessed and the target register name is placed on the PRTQ (can not be returned immediately)
- 1 Tag at the head of FL is entered in the REN table
- 1 The new physical register name is placed on OLQ and the LC of the prior arithmetic instruction incremented
- 1 GB is set, tag returned to FL, FXP uses ST data buffer for the address

# Super-scalar Issues: *Dynamic Scheduling*

Register-Renaming: IBM RS/6000 (1990)

How does it work ?

Returning names to the FL:

1 Names are returned to the FL from PTRQ when the content of the physical register becomes free - the last arithmetic instruction or store referencing that physical register has been performed:

# Arithmetic: when they complete decode

# Stores: when they are removed from the store queue

When LD causes new mapping, the last instruction that could have used that physical register was the most recent arithmetic instruction, or ST. Therefore when the most recent prior arithmetic decoded or store has been performed that physical register can be returned

# Super-scalar Issues: *Dynamic Scheduling*

## Register-Renaming: IBM RS/6000 (1990)

Example:

<i>Original stream</i>	<i>Rename Table</i>	<i>Free Head</i>	<i>Renamed stream</i>	<i>PTRQ</i>
FADD R3, R2, R1	(1,1);(2,2);(3,3)	32	R3, R2, R1	
FST R3	(3,3)	32	R3	
FLD R3	(3,3)	32	PR32	3
FMUL R6, R3, R1	(1,1);(3,32);(6,6)	33	R6, R32, R1	
FSUB R2, R6, R2	(2,2);(6,6);(2,2)	33	R2, R6, R2	
FLD R3	(3,32)	33	PR33	32

# Super-scalar Issues: *Exceptions*

Super-scalar processor achieves high performance by allowing instruction execution to proceed without waiting for completion of previous ones.

The processor must produce a correct result when an exception occurs.

Exceptions are one of the most complex areas of computer architecture, they are:

**Precise:** *when exception is processed, no subsequent instructions have begun execution (or changed the state beyond of the point of cancellation) and all previous instruction have completed*

**Imprecise:** *leave the instruction stream in the neighborhood of the exception in recoverable state*

RS/6000: precise interrupts specified for all program generated interrupts, each interrupt was analyzed and means of handling it in a precise fashion developed

External Interrupts: handled by stopping the I-dispatch and waiting for the pipeline to drain.

# Super-scalar Issues:

## *Instruction Issue and Machine Parallelism*

### 1 In-Order Issue with In-Order Completion:

- # The simplest instruction-issue policy. Instructions are issued in exact program order. Not efficient use of super-scalar resources. Even in scalar processors in-order completion is not used.

### 1 In-Order Issue with Out-of-Order Completion:

- # Used in scalar RISC processors (Load, Floating Point).
- # It improves the performance of super-scalar processors.
- # Stalled when there is a conflict for resources, or true dependency.

### 1 Out-of-Order Issue with I Out-of-Order Completion:

- # The decoder stage is isolated from the execute stage by the “instruction window” (additional pipeline stage).

# Super-scalar Examples:

## *Instruction Issue and Machine Parallelism*

### DEC Alpha 21264:

1 Four-Way ( Six Instructions peak), Out-of-Order Execution

### MIPS R10000:

1 Four Instructions, Out-of-Order Execution

### HP 8000:

1 Four-Way, Agressive Out-of-Order execution, large Reorder Window

1 Issue: In-Order, Execute: Out-of-Order, Instruction Retire: In-Order

### Intel P6:

1 Three Instructions, Out-of-Order Execution

### Exponential:

1 Three Instructions, In-Order Execution

# Super-scalar Issues:

*The Cost vs. Gain of Multiple Instruction Execution*

## PowerPC Example:

<b>Feature</b>	<b>601+</b>	<b>604</b>	<b>Difference</b>
<i>Frequency</i>	100MHz	100MHz	same
<i>CMOS Process</i>	.5u 5-metal	.5u 4-metal	~same
<i>Cache Total</i>	32KB Cache	16K+16K Cache	~same
<i>Load/Store Unit</i>	No	Yes	
<i>Dual Integer Unit</i>	No	Yes	
<i>Register Renaming</i>	No	Yes	
<i>Peak Issue</i>	2 + Branch	4 Instructions	~double
<i>Transistors</i>	2.8 Million	3.6 Million	+30%
<i>SPECint92</i>	105	160	+50%
<i>SPECfp02</i>	125	165	+30%

# Super-scalar Issues:

## *Comparisson of leading RISC microrpocessors*

<b>Feature</b>	<b>Digital 21164</b>	<b>MIPS 10000</b>	<b>PowerPC 620</b>	<b>HP 8000</b>	<b>Sun UltraSparc</b>
<i>Frequency</i>	500 MHz	200 MHz	200 MHz	180 MHz	250 MHz
<i>Pipeline Stages</i>	7	5-7	5	7-9	6-9
<i>Issue Rate</i>	4	4	4	4	4
<i>Out-of-Order Exec.</i>	6 loads	32	16	56	none
<i>Register Renam. (int/FP)</i>	none/8	32/32	8/8	56	none
<i>Transistors/ Logic transistors</i>	9.3M/ 1.8M	5.9M/ 2.3M	6.9M/ 2.2M	3.9M*/ 3.9M	3.8M/ 2.0M
<i>SPEC95 (Intg/FlPt)</i>	12.6/18.3	8.9/17.2	9/9	10.8/18.3	8.5/15
<i>Perform./ Log-trn (Intg/FP)</i>	7.0/10.2	3.9/7.5	4.1/4.1	2.77*/4.69	4.25/7.5

\* no cache

# Super-scalar Issues:

## *Value of Out-of-Order Execution*

<b>Feature</b>	<b>MIPS 5000</b>	<b>MIPS 10000</b>	<b>HP-PA 7300LC</b>	<b>HP 8000</b>	<b>Digital 21164</b>	<b>Digital 21264</b>
<i>Frequency</i>	180 MHz	200 MHz	160 MHz	180 MHz	500 MHz	600 MHz
<i>Pipeline Stages</i>	5	5-7	5	7-9	7	7/9
<i>Issue Rate</i>	2	4	2	4	4	4+2
<i>Out-of-Order Exec.</i>	none	32	none	56	6 loads	20i+15fp
<i>Register-Renam. (int/FP)</i>	none	32/32	none	56	none/8	80/72
<i>Transistors/ Logic transistors</i>	3.6M/ 1.1	5.9M/ 2.3M	9.2M/ 1.7M	3.9M*/ 3.9M	9.3M/ 1.8M	15.2M/ 6M
<i>Cache</i>	32/32K	32/32K	64/64K	none	8/8/96	64/64K
<i>SPEC95 (Intg/FlPt)</i>	4.0/3.7	8.9/17.2	5.5/7.3	10.8/18.3	12.6/18.3	~36/~60
<i>Perform./ Log-Tr (Intg/FP)</i>	3.6/3.4	3.9/7.5	3.2/4.3	2.77*/4.69	7.0/10.2	6.0/10.0

\* no cache

# The ways to exploit instruction parallelism

## 1 Super-scalar:

# takes advantage of instruction parallelism to reduce the average number of cycles per instruction.

## 1 Super-pipelined:

# takes advantage of instruction parallelism to reduce the cycle time.

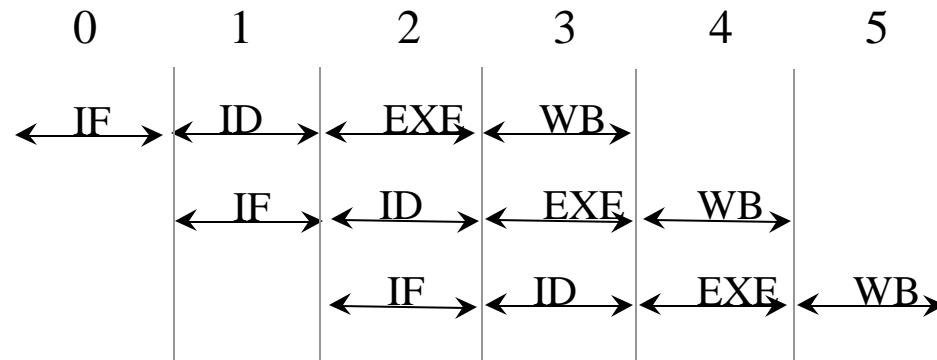
## 1 VLIW:

# takes advantage of instruction parallelism to reduce the number of instructions.

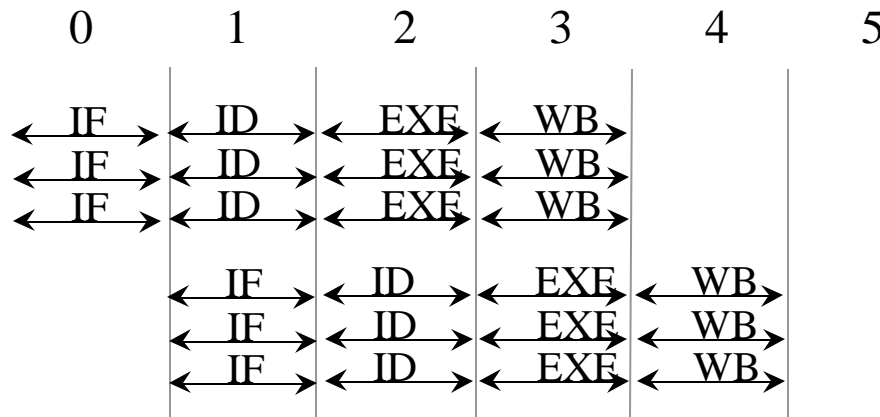
# The ways to exploit instruction parallelism:

## Pipeline

Scalar:

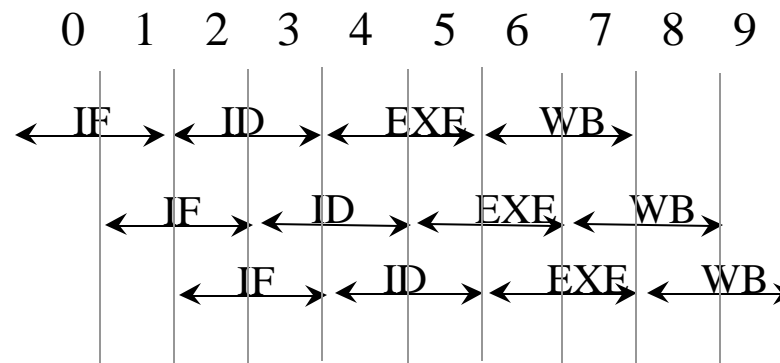


Super-scalar:

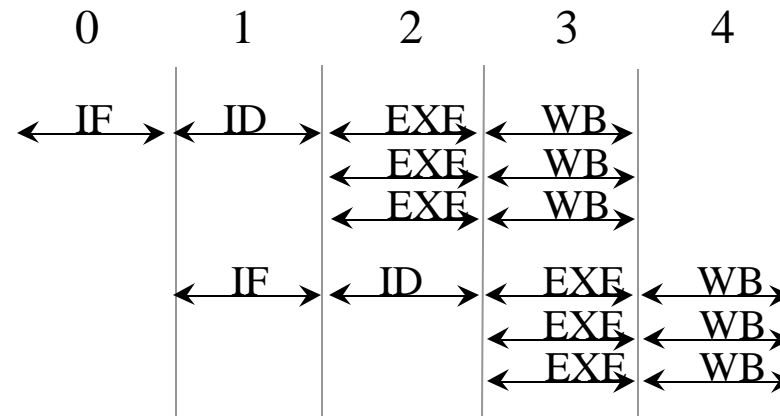


# The ways to exploit instruction parallelism: Pipeline

## Super-pipelined:



## VLIW:



# Very-Long-Instruction-Word Processors

- 1 A single instruction specifies more than one concurrent operation:
  - # This reduces the number of instructions in comparison to scalar.
  - # The operations specified by the VLIW instruction must be independent of one another.
- 1 The instruction is quite large:
  - # Takes many bits to encode multiple operations.
  - # VLIW processor relies on software to pack the operations into an instruction.
  - # Software uses technique called “compaction”. It uses no-ops for instruction operations that cannot be used.

VLIW processor is not software compatible with any general-purpose processor !

# Very-Long-Instruction-Word Processors

- 1 VLIW processor is not software compatible with any general-purpose processor !
- 1 It is difficult to make different implementations of the same VLIW architecture binary-code compatible with one another.
  - # because instruction parallelism, compaction and the code depend on the processor's operation latencies
- 1 Compaction depends on the instruction parallelism:
  - # In sections of code having limited instruction parallelism most of the instruction is wasted
- 1 VLIW lead to simple hardware implementation

# Super-pipelined Processors

- 1 In Super-pipelined processor the major stages are divided into sub-stages.
  - # The degree of super-pipelining is a measure of the number of sub-stages in a major pipeline stage.
  - # It is clocked at a higher frequency as compared to the pipelined processor ( the frequency is a multiple of the degree of super-pipelining).
  - # This adds latches and overhead (due to clock skews) to the overall cycle time.
  - # Super-pipelined processor relies on instruction parallelism and true dependencies can degrade its performance.

# Super-pipelined Processors

- 1 As compared to Super-scalar processors:
  - # Super-pipelined processor takes longer to generate the result.
  - # Some simple operation in the super-scalar processor take a full cycle while super-pipelined processor can complete them sooner.
  - # At a constant hardware cost, super-scalar processor is more susceptible to the resource conflicts than the super-pipelined one. A resource must be duplicated in the super-scalar processor, while super-pipelined avoids them through pipelining.
- 1 Super-pipelining is appropriate when:
  - # The cost of duplicating resources is prohibitive.
  - # The ability to control “clock skew” is good

*This is appropriate for very high speed technologies: GaAs, BiCMOS, ECL (low logic density and low gate delays).*

# Conclusion

- 1 Difficult competition and complex designs ahead, yet:  
*“Risks are incurred not only by undertaking a development, but also by not undertaking a development”* - \*Mike Johnson (Super-scalar Microprocessor Design, Prentice-Hall 1991)
- 1 Super-scalar techniques will help performance to grow faster, with less expense as compared to the use of new circuit technologies and new system approaches such as multiprocessing.\*
- 1 Ultimately, super-scalar techniques buy time to determine the next cost-effective techniques for increasing performance.\*

# Acknowledgment

I thank those people for reading my overheads - correcting my spelling mistakes and making useful and valuable suggestions and contributions toward improvement:

- William Bowhill, DEC
- Ian Young, Intel
- Krste Asanovic, UC Berkeley