KEEPING CONTROL TRANSFER INSTRUCTIONS OUT OF THE PIPELINE IN ARCHITECTURES WITHOUT CONDITION CODE

J. Cortadella
J.M. Llabería
A. González

RR 87/11

KEEPING CONTROL TRANSFER INSTRUCTIONS OUT OF THE PIPELINE IN ARCHITECTURES WITHOUT CONDITION CODES

Jordi Cortadella,
Josep M. Llabería and Antonio González

Departamento de Arquitectura de Computadores
Facultad de Informática de Barcelona (UPC)
Pau Gargallo, 5, 08028 Barcelona (SPAIN)

This work was funded by the Ministry of Education (CAICYT) under contract Number 314-85

KEEPING CONTROL TRANSFER INSTRUCTIONS OUT OF THE PIPELINE IN ARCHITECTURES WITHOUT CONDITION CODES

L'execució de les instruccciones de salt suposa una pèrdua de rendiment en el processadors segmentats. En aquest treball es presenta un mecanisme que permet l'execució d'aquest tipus d'instruccions amb un cost nul de temps. Aquest mecanisme és adequat per arquitectures sense codis de condició.

Abstract

The execution of branch instructions involves a loss of performance in pipelined processors. In this paper we present a mechanism for executing this kind of instruction with a zero delay. This mechanism has been proposed for architectures without condition codes.

1. INTRODUCTION

The execution of control transfer instructions causes a loss of performance in pipelined processors. Two operations must be performed to execute them: condition evaluation (in the case of conditional branches) and target address computation. Two kind of architectures can be distinguished considering the way of computing the condition evaluation: with condition codes and without them. In the first case the condition is established by some instruction that precede the conditional branch. Statistics of instruction sequences have shown that this instruction is usually the one that immediately precedes the branch and is a compare or test

instruction. For this reason some architectures have benefit of this fact to introduce a "compare&branch" in its instruction set and to eliminate condition codes. Lately, RISC architectures have followed this approach. (MIPS-X, SPUR, HP Spectrum)/Far86,Kat85,Mah86/. However, the execution frequency of this kind of instructions (20-30%) still involves a great penalization in processor performance.

So far several mechanisms have been proposed to reduce this negative effect in RISC architectures. Most of them have been implemented in existing processors (Delayed Branch, Squashed Branch) /Far86/. In /Kat83/ an Instruction Fetch Unit has been proposed to execute conditional and unconditional branches efficiently. This IFU keeps unconditional branches, transparent to the Execution Unit and allows to execute compare&branch instructions without any extra delay. Target address computation is performed in the IFU while comparison is computed in the EU. Compare&branch instructions spend one cycle only. A dual-port instruction memory is needed to prefetch the two candidate instructions that can be executed after a branch.

In this paper we propose an execution mechanism for RISC-like architectures that keeps "cmp&br" instructions out of the pipeline. Comparisons are computed in the EU in parallel with another useful instruction. In this way, a meaningful improvement of the useful instruction throughput can be achieved. (We name useful instructions those that are neither control transfer instructions nor NOP. Cmp&br are considered as non-useful instructions). We have also assumed that the IFU can accede to a dual-port instruction memory.

This mechanism is based on several points:

- A simple modification of the machine language.

- The desing of an Instruction Fetch and Sequencing Unit (IFSU) that executes control transfer instructions. Some simple modifications must be also introduced in the EU.
- A technique for optimizing the computation of comparisons at compile-time.

Next sections are dedicated to describe these point in more detail.

2. MODIFICATION OF MACHINE LANGUAGE

We can represent the code of a program as a graph of Basic Blocks (BB). An important property of BBs is that all their instructions are executed without any sequence break. Branches are always at the bottom of BBs.

Condition evaluation depends on the execution of instructions that precede the branch. On the opposite, branch target address computation does not (except in some rare cases). The model we present uses this feature to make an advanced two-way prefetch of candidate instructions to be executed after a branch. To make it possible we introduce the following modifications (see fig. 1).

a) Control transfer instructions are put at the top of the BB they belong to. However its execution isn't effective until BB ends.

b) Compare&branch instruction have the following format:
   - Operation code.
   - Condition to be evaluated.
   - Immediate (to be compared).
   - Offset (to do target address computation).

c) Two bits are included in the coding of every instruction. One of them (the "branch-bit") indicates wheter or not this is the last instruction in a BB with a control transfer instruction. The other one ("comparison-bit") indicates that the result computed by the instruction must be compared with another value when next conditional branch is executed.

You can notice that cmp&br instructions need not to specify registers to be compared because their values will be set and gathered during the execution of other instructions. The optimization technique explained in section 4 takes advantage of this property.

3. IFSU DESIGN

Fig. 2 shows a block diagram of the IFSU we propose. Their main components are:

PC: Registers that keep addresses of the two prefetched instructions. Each one has an associated validity bit (V).

COND_IMM: Register that keeps the code of the condition that EU must evaluate and the immediate value that can be compared to establish the condition.

IQ: Queues of the instructions candidate to be sent to EU. Each one stores one of the two possible instructions streams to be executed after a branch.

SEL: Control logic that selects the instruction stream that must be fed into EU.

TC: Logic to compute branch target address.

TAQ: Queues of target addresses of branches prefetched from the opposite instruction stream.

CIQ: Id. for conditions and immediates specified in branches.

BR: Signal sent from EU to indicate the result of the condition evaluation.

The general behavior of the IFSU is the following: In each cycle, instructions addressed by PC_a and PC_b are prefetched. Non-control transfer instructions are sent to IQs. In the case of those of control transfer, its information (target address, condition and immediate) is sent to TAQ and CIQ respectively. These transmitted values will fall to PC and COND_IMM if they are invalid (V=off).

The execution of an instruction with the branch-bit on can produce a change in the choice of the stream that must be fed into EU (if the branch is taken). However, in all cases the following actions have to be done:

- Clear IQ of the non-selected stream (NSS).
- Clear TAQ and CIQ of the selected stream (SS).
- Define SEL to select IQ of the SS.
- Define PC of NSS and COND_IMM with the first element of TAQ an CIQ of NSS. If these queues are empty, PC and COND_IMM will become invalid (V=off).

Figure 4 shows an example of the IFSU state during execution of code according to the specified BB subgraph.

Some simple modifications must be introduced in the EU to make condition evaluation efficiently (see figure 3). Two latches are needed to gather the values that have to be compared. They are loaded in a FIFO manner when an

instruction with the "comparison-bit" on is sent to the EU. A comparator is also used to evaluate the condition. This can be computed in parallel with the ALU operation from least to most significant bits, following the ALU carry propagation. The output of the comparator is valid at the end of each cycle and can be read by the IFSU to take branch decisions.

4. OPTIMIZATION OF COMPARISONS

In a RISC-machine without condition codes, the condition is established by the comparison of two registers or a register and an immediate. In the machine language we have proposed, the values that have to be compared are obtained from the execution of instructions with the "comparison-bit" on. This fact may involve to insert some extra instructions in the code to read the required registers before the end of the BB. However, if we analyze the code that the compiler generates we can see that compared registers are usually defined in the basic block where they are compared. In some cases, values defined at different basic blocks can also be used.

This is the case of many conditions at the end of a loop. For example:

```
for (i = 0; i> 50; i++)
```

is translated into

```
top_of_loop:

    cmp&br lt,  50, top_of_loop          last BB in
                                         the loop

    add R_i, 1, R_i, (comparison, branch)
end_of_loop
```

The compiler can take advantage of this feature to do a suitable optimization of condition evaluation easily. Comparison-bits save the explicit execution of a compare instruction.

Statistics obtained from the execution of several programs show that only 17% of conditional branches need extra instructions to fetch the values to be compared.

5. PERFORMANCE EVALUATION

The mechanism we have proposed has been evaluated by the simulation of the entire execution of several large representative programs (LEX[1], CEM[2], NROFF[1], YACC[1]). Measurements have been obtained using a evaluation methodology proposed in /COR87a/. We have designed a compiler adding a new back-end to the ACK /TAN83/ to generate a machine language based on the one of Berkeley RISC-II processor.

Several mechanisms have been compared with the IFSU we propose /FAR86/ /KAT83/:

a) Delayed branch
b) Squashed branch
c) Profiled Squashed branch
d) IFSU(Kat83)

In order to compare all this mechanisms with the same conditions, we have assumed that IFSU(Kat83) can execute any

---

1) Programs of UNIX Operating System (UNIX is a Trademark of AT&T Bell Labs.)
2) C front-end of the ACK.

kind of cmp&br instructions in one cycle (even non-fast comparisons).

Performance has been evaluated as the useful instruction throughput rate that EU can execute. Control transfer instructions (cmp&br, unconditional branches, call, return,...) are considered as non-useful instructions. Extra instructions executed due to branch execution are also considered non-useful.

Table I shows the performance of several mechanisms. The proposed IFSU can achieve an improvement of 6% to 21% in processor performance compared with the other mechanisms.

	Delayed	Squashed	Prof.Squashed	IFSU-Kat83	IFSU
LEX	0.59	0.68	0.69	0.70	0.77
CEM	0.60	0.63	0.67	0.74	0.75
YACC	0.68	0.74	0.76	0.78	0.86
NROFF	0.63	0.68	0.72	0.78	0.82
Average	0.63	0.68	0.71	0.75	0.80

Table I: Throughput (useful instructions per cycle) achieved by several branch execution mechanisms

Also, IFSU behavior with a dual-port instructions cache has been evaluated. The interest of evaluating it in this conditions lies in determining the extra main memory traffic that could be generated. The two-way prefetching can pollute instruction cache with some code parts that are not needed. Statistics have shown that this extra traffic is similar to the one produced by an "always prefetch" algorithm on cache misses /Smi85/.

Simulations with several queue sizes have been runned. They have shown that less than 1% of performance is lost with IQs of size 3 and TAQs and CIQs of size 1.

6. CONCLUSIONS

In this paper, we present a mechanism to keep control transfer instructions out of the pipeline of an architecture without condition codes. It is based on an IFSU with a dual-port instruction memory that executes branch instructions. The machine language has been modified to allow an advanced target address computation and a two-way instruction prefetch. Comparisons are executed in parallel with one useful instruction without spending pipeline cycles. A comparator has been introduced in the EU to compute conditions quickly.

This mechanism achieves a meaningful improvement in processor performance (6% to 21%). Also main memory traffic generated by IFSU has been evaluated. It is similar to the one produce by prefetch algorithms on cache misses.

ACKNOWLEDGEMENTS

Our appreciation to the "Facultad de Informática de Madrid" for the permission to use a PDP-11/60 with UNIX.

REFERENCES

/Kat83/ Manolis G.H. Katevenis, "Reduced Instruction Set Computer Architectures for VLSI", Ph. D. Dissertation, University of California, Berkeley, October 1983.

/Smi85/ Alan J. Smith, "Cache Evaluation and the impact of workload choice", Proc. of the 12th. Ann. Int. Symp. on Comp. Arch., June 1985, pp. 64-73.

/Far86/ Scott Mc Farling and John Hennesy, "Reducing the Cost of Branches", Proc. 13th. Ann. Symp. on Comp. Arch., June 1986, pp. 396-403.

/Tan83/ A.S. Tanenbaum et al., "A practical tool kit for making portable compilers", Comm. ACM 26:9, 1983, pp. 654-660.

/Cor87a/ J. Cortadella and J.M. Llabería, "A Low Cost Evaluation Methodology for New Architectures", 5th. Int. Symp. on Applied Informatics. Grindelwald (Switzerland), February 1987.

/Mah86/ Michael J. Mahon et al., "Hewlett-Packard Precision Architecture: The Processor", Hewlett-Packard Journal, Volume 37 Number 8, August 1986.

/Kat85/ H. Katz, editor, "Proceedings of CS292i: Implementation of VLSI Systems,"Technical Report UCB/CSD 86/259, Computer Science Division (EECS), University of California, Berkeley, September, 1985.

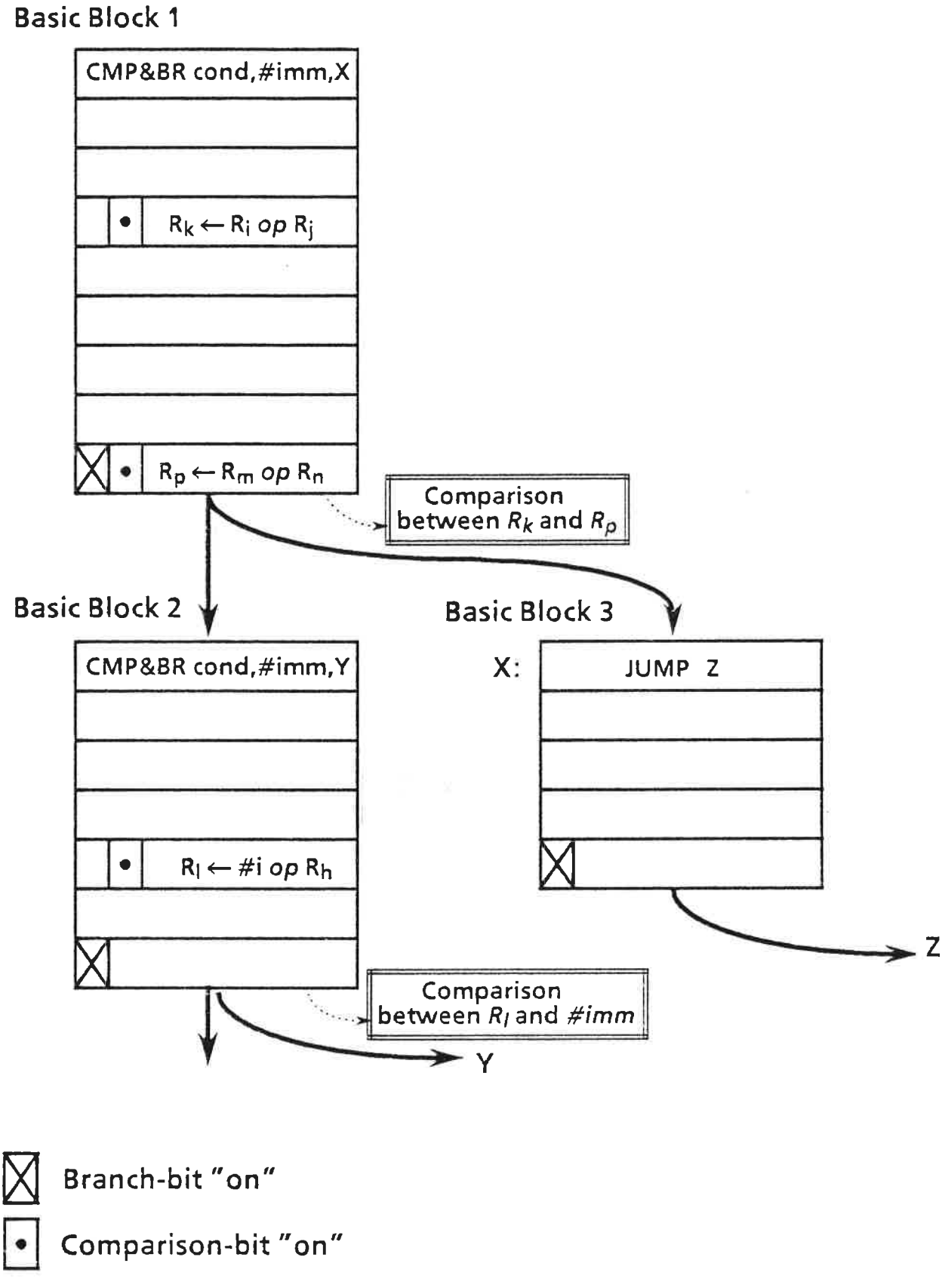

Figure 1

Basic Block structure in the proposed Machine Language

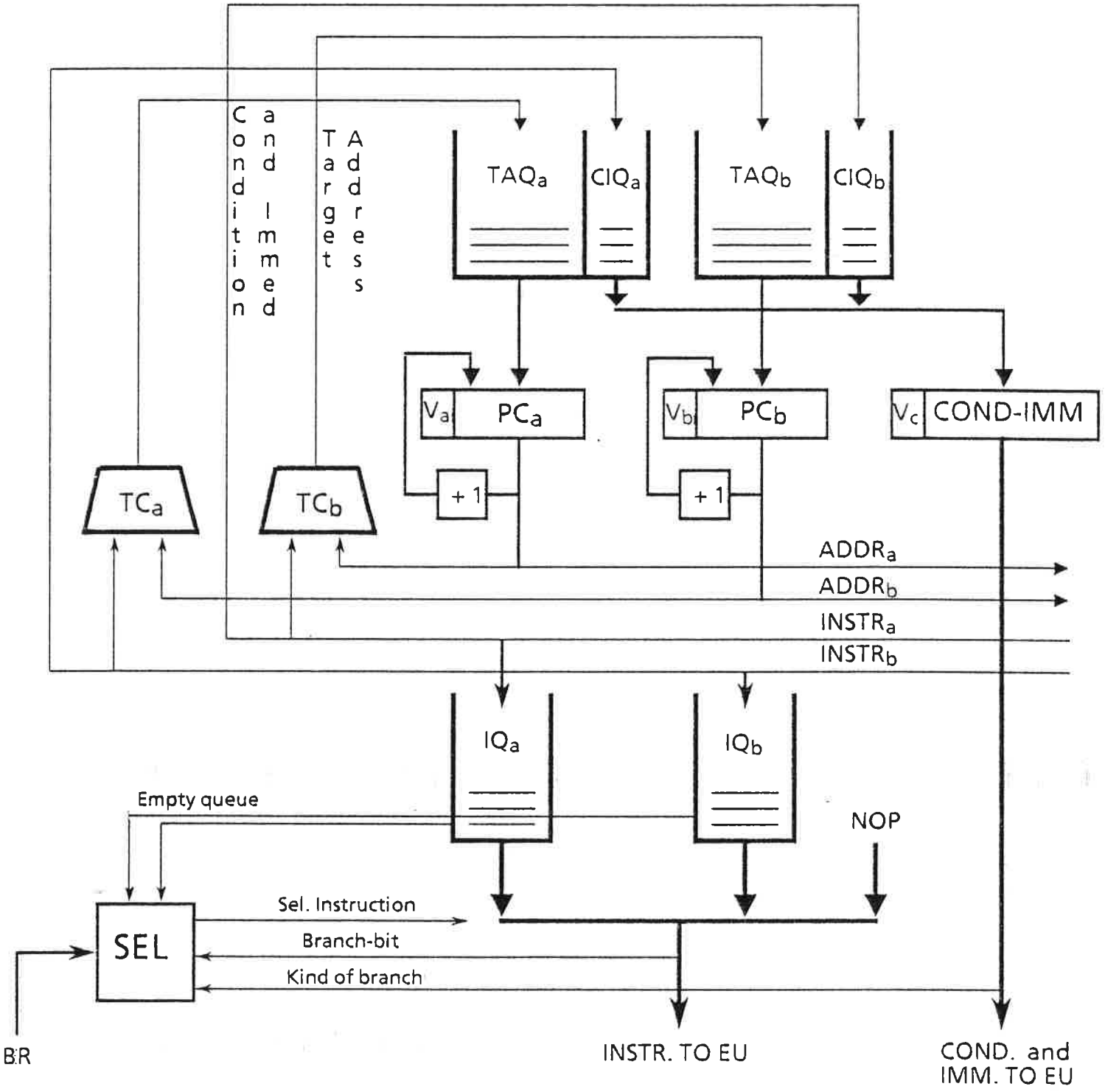
Condition and Immed
Target Address
TAQa
CIQa
TAQb
CIQb
Va
PCa
Vb
PCb
Vc
COND-IMM
+ 1
+ 1
TCa
TCb
ADDRa
ADDRb
INSTRa
INSTRb
IQa
IQb
Empty queue
NOP
SEL
Sel. Instruction
Branch-bit
Kind of branch
BR
INSTR. TO EU
COND. and IMM. TO EU

Figure 2

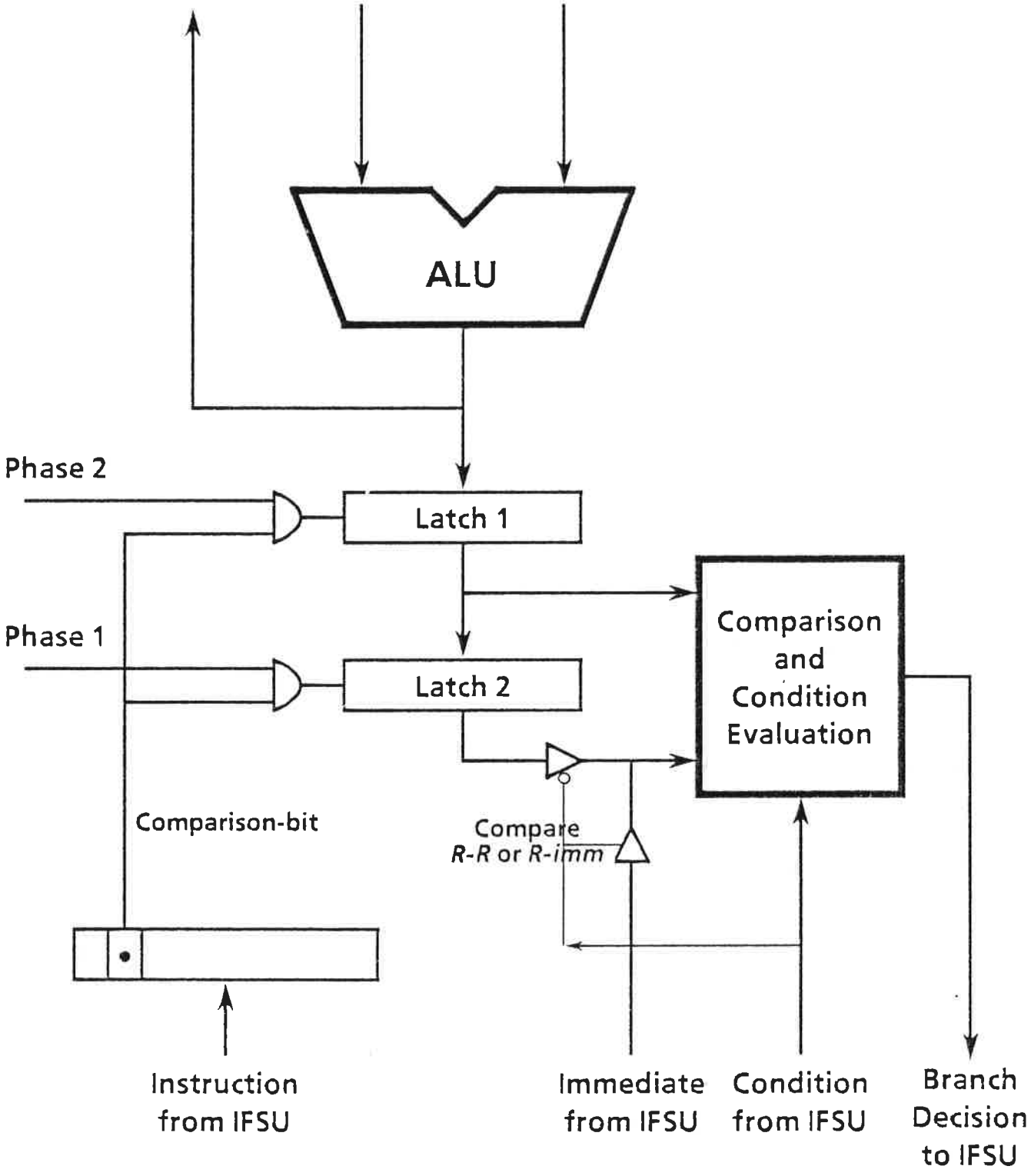

Figure 3

EU Support for comparisons

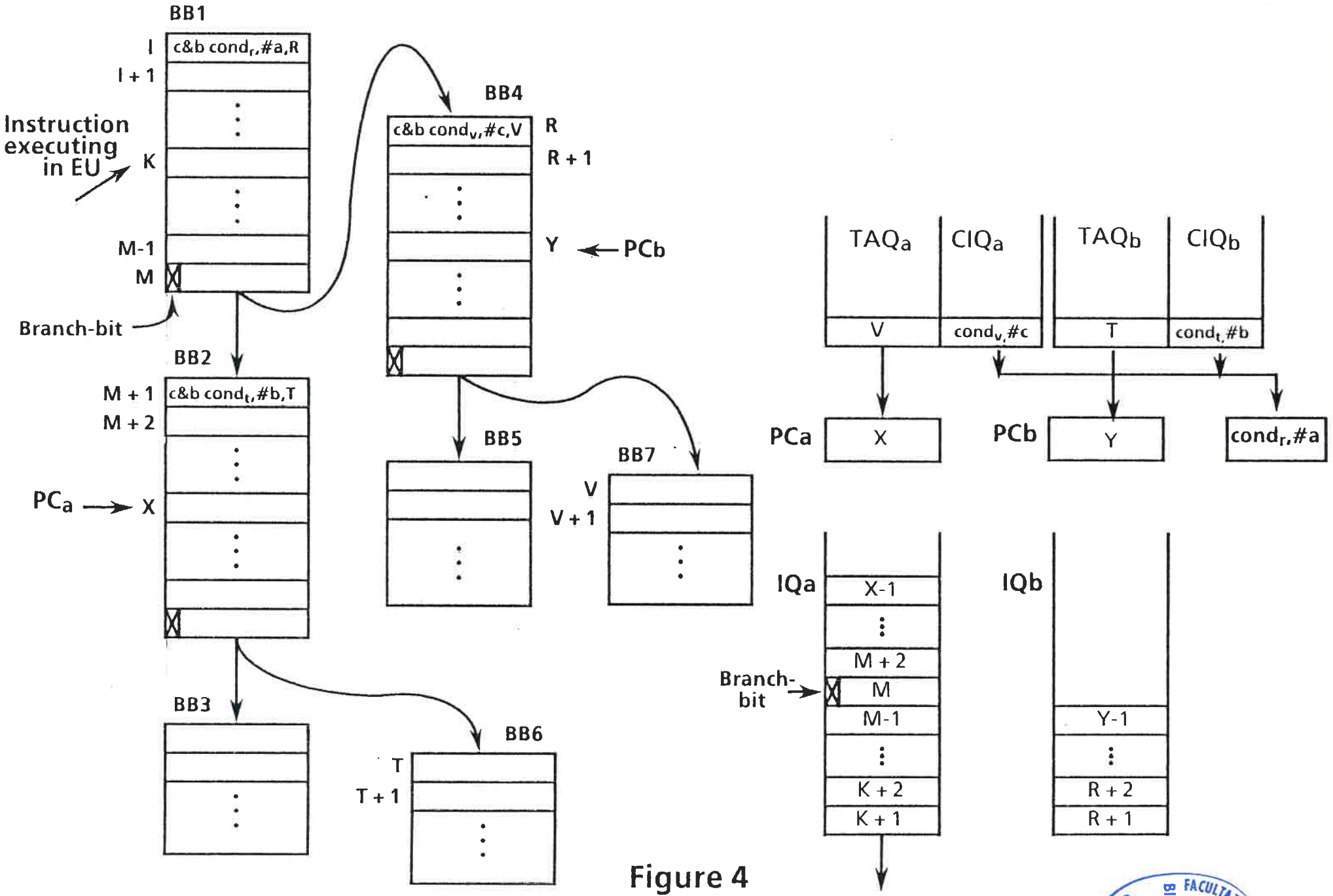

Figure 4

State of IFSU during execution

For a better understanding we have put the addresses of instructions instead the instructions themselves