Deliverable d3: Architecture Logical Blocks implementation

Sistemi di Elaborazione a.a. 2018/2019

Group M

Produced by	Date of Approval	Approved by	Version
Fernando Manna, Giovanni Di Prisco, Andrea Maione, Antonio Coppola, Edoardo Cossentino	04 Jan. 2019	Francesco de Pertis, Fernando Manna	1.0

Deliverable d3:
Architecture Logical blocks implementation by group M

Release Information

The following changes have been made to this document.

Date	Revision	Authors	Change Details
04 January 2019	0	Fernando Manna, Giovanni Di Prisco, Andrea Maione, Antonio Coppola, Edoardo Cossentino	Schemas and calculation of the cost amount. MMU performance evaluation by MMU sub-team.

Table 1 Change History

Contents

Deliverable d3:
Architecture Logical blocks implementation by group M

List of Tables

Deliverable d3:

Architecture Logical blocks implementation by group M

List of Figures

Deliverable d3:

Architecture Logical blocks implementation by group M

List of Acronyms

Deliverable d3:

Architecture Logical blocks implementation by group M

ASID	Address Space Identifier
AU	integer Arithmetic Unit
BHT	Branch History Table
BPB	Branch Prediction Buffer
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
BU	Branch instruction Unit
CC	Clock Cycle
EX	Execute
FU	Functional Unit
ID	Instruction Decode
IF	Instruction Fetch
L1	Level 1
L2	Level 2
LSU	Load and Store instructions Unit
MEM	Memory
MM	Main Menu
MMU	Memory Management Unit
MU	Multiplication Unit
OOO	Out-of-Order
PC	Program Counter
ROB	Re-Order Buffer
RSR	Result Shift Register
TLB	Translation Lookaside Buffer
VIPT	Virtually Indexed Physically Tagged
VM	Virtual Memory
WB	Write Back
WT	Write Through

Preface

This preface introduces the Deliverable d3: Processor Logical blocks implementation. It contains the following sections:

About this manual

The purpose of this manual is to describe the logical blocks implementation of some non-trivial combinatorial networks. In particular, we intend to analyze the logical structure of the following components:

- Branch recognizer
- Aliasing Recognizer
- Instruction Decoder
- Forward Decoder
- Acceptance Unit
- Forwarding Unit

An approximate account of the cost of the designed architecture will then be provided. Moreover, the MMU performance will be shown.

Intended audience

This manual is written for experienced hardware and software engineers who might or might not have experience of ARM products.

Using this manual

The information in this manual is organized into two chapters, as described below.

Chapter 1: Logical Blocks implementation

Chapter 1 describes the implementation of:

- Branch Recognizer and Aliasing Recognizer blocks contained within the Branch Prediction Unit Block designed by group M.
- Forward Decoder and Instruction Decoder contained within the Control Unit block designed by group M.
- Acceptance Unit contained in the Out-of-Order Block designed by group M.
- Forwarding Unit implementation designed by group M.

Chapter 2: Cost of the architecture

Chapter 2 shows an approximate account of the cost of the designed architecture.

Chapter 3: MMU Performance evaluation

Chapter 3 the performance of the MMU of the architecture designed by group M.

Chapter 4: Resources

Chapter 4 describes how work has been organized between team members and the resulting total efforts in working hours.

Conventions

Graphical

The graphical conventions for the signals within the schemes are:

⇗	Denotes aggregate bus
↗	Denotes data bus
↗	Denotes address bus
↗	Denotes control bus

References

ARM publications

- ARM Architecture Reference Manual (ARM DDI 0100I)
- ARM9TDMI Technical Reference Manual (ARM DDI 0180A)

Other sources

- Computer Organization and Design (3rd edition) - David A. Patterson, John L. Hennessy
- Lecture notes of the course of Sistemi di elaborazione a.y. 2018/2019 – Prof. Angelo Marcelli

Chapter 1: Logical Blocks implementation

This chapter introduces the Branch Prediction Unit Block implementation and contains the following sections:

- *Branch Recognizer* on page 3
- *Aliasing Recognizer* on page 3
- *Instruction Decoder* on page 4
- *Instruction Decoder*

The task of the Instruction Decoder is to assert all the relevant data and control signals needed for pipeline execution.

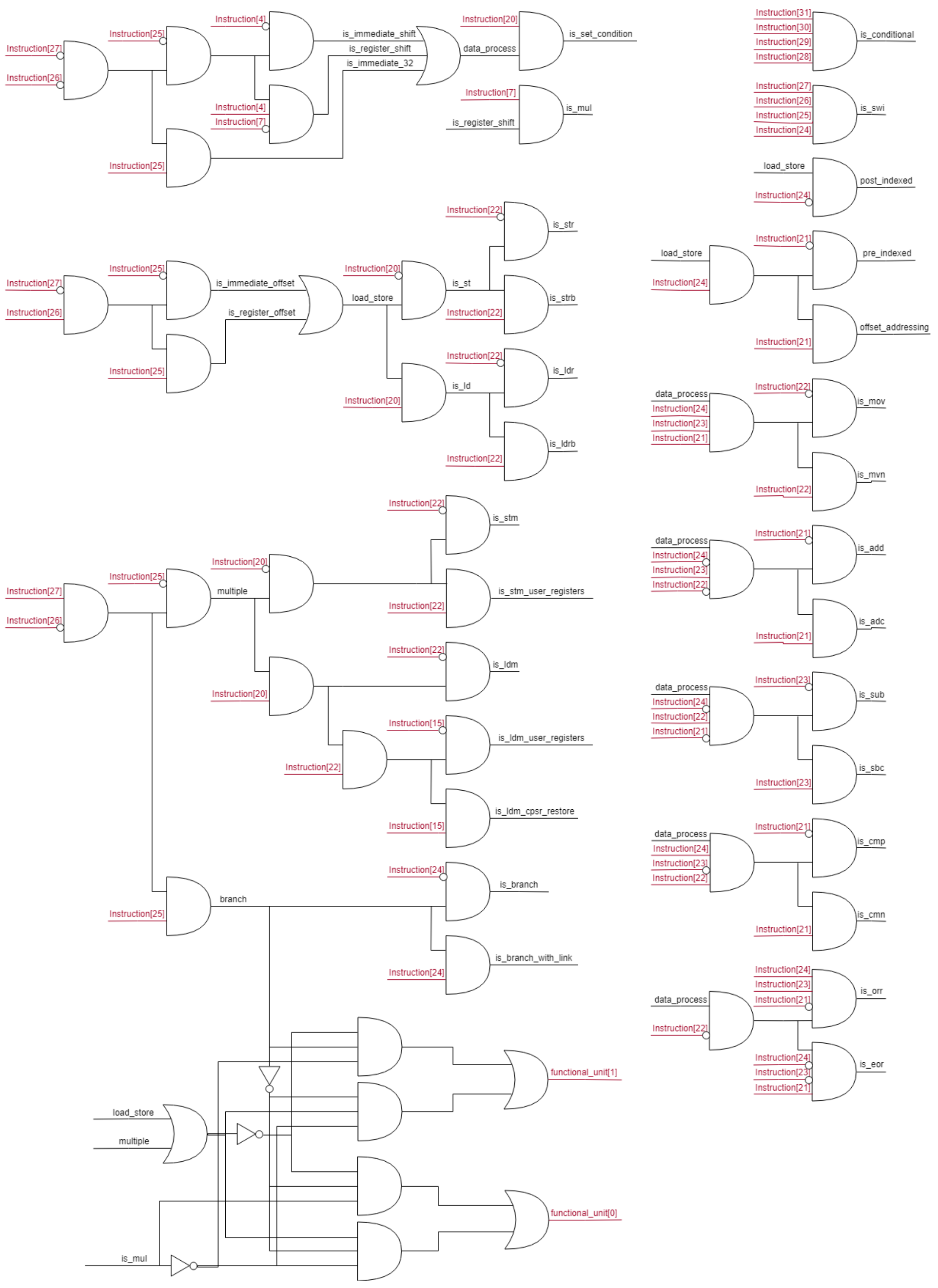

Figure 3: Instruction Decoder

- Forward Decoder on page 5
- *Acceptance Unit* on page 6
- *Forwarding Unit* on page 7

1 Branch Recognizer

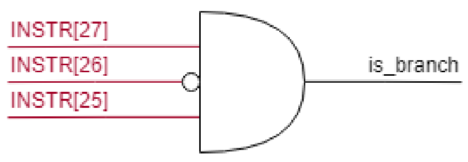

Figure 1: is_branch signal

The recognition of a branch instruction is performed by a single logic port capable of asserting the is_branch signal if INTR [27-25] has value 101.

2 Aliasing Recognizer

The miss and reset_validity signals are generated by the same combinational network within the module responsible for the aliasing recognition.

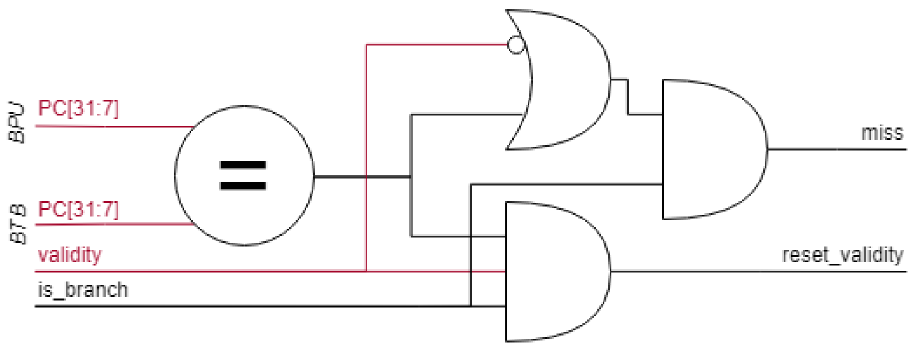

Figure 2: Miss and reset_validity signals

3 Instruction Decoder

The task of the Instruction Decoder is to assert all the relevant data and control signals needed for pipeline execution.

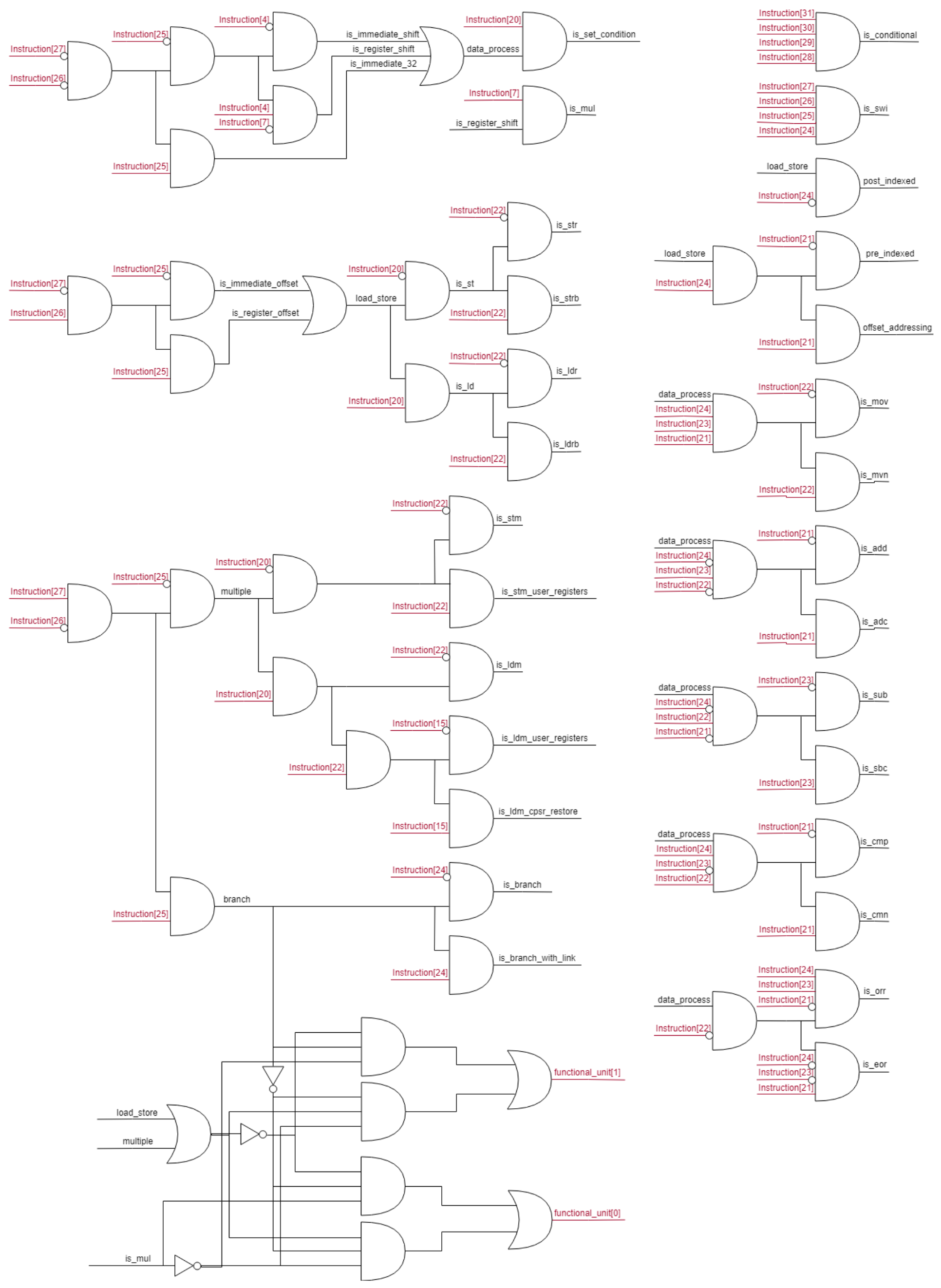

Figure 3: Instruction Decoder

4 Forward Decoder

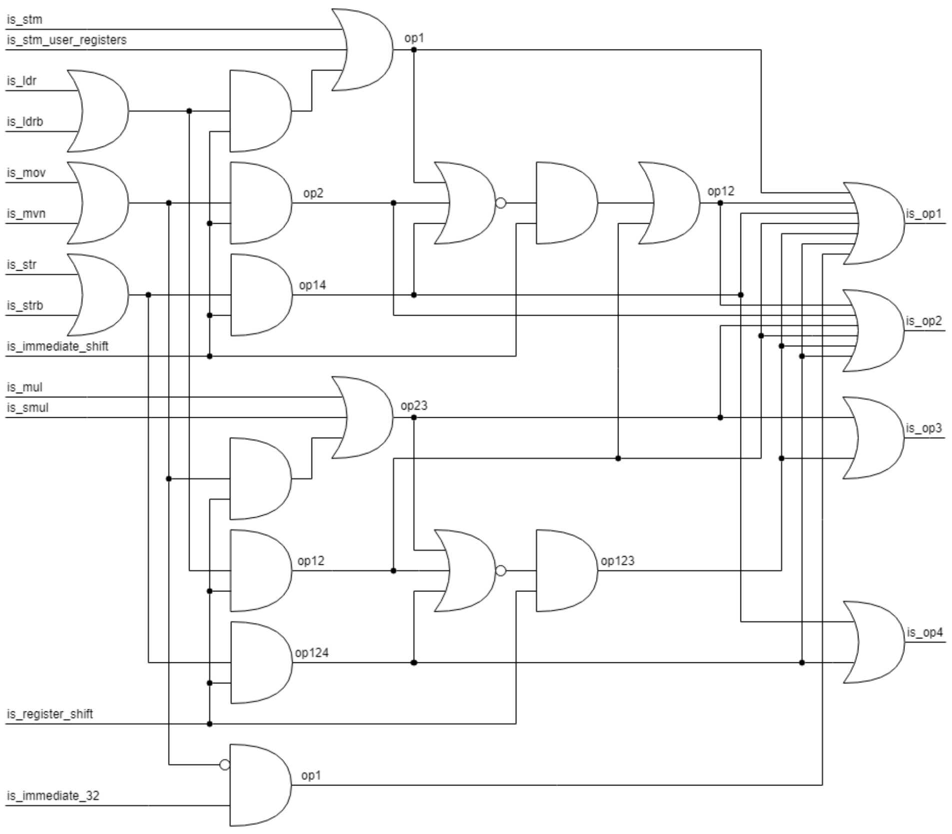

Figure 4: Forward Decoder

The task of the Forward Decoder is to identify which of the four fields predisposed to contain the operands are actually used by the current instruction. In accordance with the instruction set that has been selected, the Forward Decoder asserts the signals corresponding to the entry fields of the ID/EX Inter-stage Buffer containing the operands.

5 Acceptance Unit

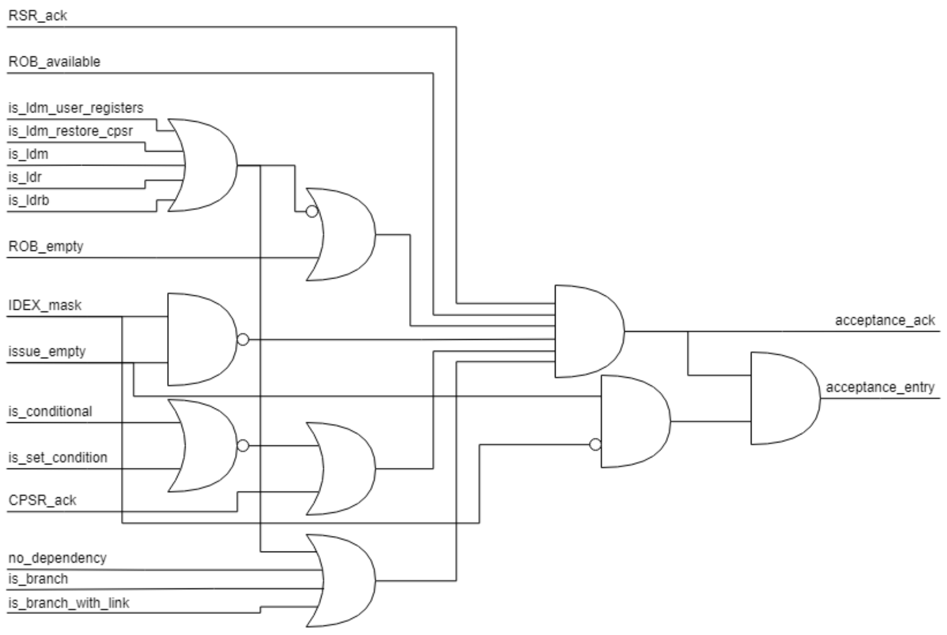

Figure 5: Acceptance Unit

The control signals issued by the Acceptance Unit concern the issue of the instruction to the Issuing Unit or to the Issue Register, respectively they are acceptance_ack and acceptance_entry. The signals take into account the status of the ROB, the RSR, the Issue Register, the dependencies due to the operands or conditional instructions. The unit also takes into account the type of operation (jump or multiple load).

6 Forwarding Unit

The logic implemented within the Forwarding Unit allows to update the fields of the Forwarding Table, to verify the dependence on the operands and to provide for their propagation.

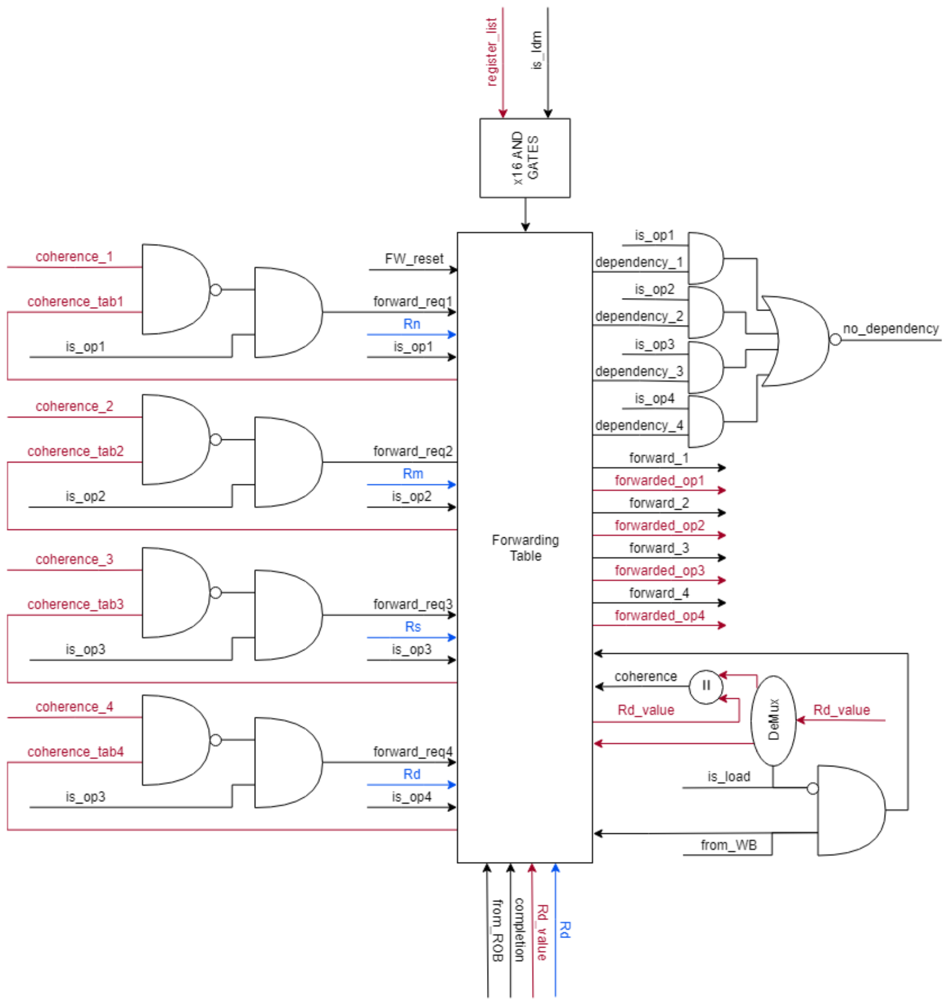

Figure 6: Forwarding Unit

Chapter 2: Cost of the architecture

This chapter introduces the resources involved in the design and contains the following section:

- *CPU Cost evaluation* on page 9
- *MMU Cost evaluation* on page 10

1 CPU Cost evaluation

For the cost evaluation of the entire architecture designed by the group M, we provide the cost in coins for the combinational networks used and the amount of memory required.

As for the combinatorial networks, the total number of inputs to the logic gates was analyzed: for example, a combinatorial network containing a 3-input AND gate and a 2-input OR gate has a total cost of 5 coins. The cost is given for the units that have requested an ad hoc design, therefore the units of Condition Verifier, Shifter and Functional Units are not evaluated, as well as the Inter-stage buffers and the Register Bank.

For units that contain memory, the cost is the total of the required storage bits.

Unit	**Combinatorial Cost (in *coins*)**	**Memory Cost (in *bits*)**
BPU	2	0
BTB *	0	1920
Branch Recognizer	4	0
Aliasing Recognizer	58	0
Priority Updater	3	0
Forward Decoder	58	0
Instruction Decoder	154	0
Issue Register *	0	3264
RSR *	0	112
Shunt Unit	205	0
Acceptance Unit	31	0
Issuing Unit	205	0
Routing Unit	205	0
ROB	32	1712
Forwarding Unit *	170	544
Exception Handler	51	2024
Wrong Prediction Handler	0	0
Full issue register and miss Handler	6	0

Table 2: Cost of the CPU architecture

(*) : A unit that contains an element of memory for which the combinatorial cost due to the management of data inside it is not counted. For example, the cost of the combinatorial network required to update the table of the RSR has not been analyzed.

2 MMU Cost evaluation

2.1 Introduction

The main purpose fixed for architecture design is to keep a good trade-off among cost, complexity and performance.

In the case of Memory Management Unit the cost is identified by extra bits added to the basic architecture. For this reason, choices made in order to achieve the above – mentioned purpose aim to minimize the extra bits usage. In some particular cases, by adding extra hardware some significative performance improvements have been observed. In that cases, the cost has been considered acceptable.

The following are the cases in which the architecture has been optimized both when extra hardware has been added and when it has avoided.

2.2 Write Buffers

The coherence management chosen policy is write-through with write buffer both for L1 cache and L2 cache. For this reason, within the memory hierarchy, two write buffers have been inserted.

In this case, extra hardware has been added but performances are optimized because the accessing latency in the underlying memory levels is hidden (see Performance evaluation paragraph).

The buffers can be full, and, in this case, the CPU has to stall the appropriate pipeline stages in order to transfer the contained data in the full buffer to his underlying memory level.

First buffer has four entries and second buffer has eight entries. Each entry can contain up to 32 bits of data and a 32-bit address, so the total cost is 768 bits.

2.3 Distiction of the stalls

The MMU implemented design provides the propagation of extra signals in order to give the possibility to the CPU to know which pipeline stages have to be stalled, in the case of the different stalls that may occur.

For this reason, the Buffer Stall Definer and Miss Stall Definer functional blocks assert two different signals in case a miss in L1 cache occurs during a single load/store or a multiple load/store. To do this, four signals have been added in the design, thus, considering that every signal is equivalent to one bit, the total cost is 4 bits.

2.4 VIPT

VIPT addressing used for L1 caches guarantees two benefits:

- Virtual address translated in physical address in a short time, because of the fact that the operations necessary for translation are performed in parallel;
- Absence of PID for each datum in L1 cache.

The second benefit is fundamental according to the fixed main purpose because, through VIPT addressing and the appropriate choice of L1 caches size is guaranteed the absence of aliasing phenomena, so PID is unnecessary.

2.5 Data of a single process in TLBs

TLBs store pages that belong to a single process.

When a context switch occurs TLBs are flushed. This operation is not costly because TLBs are relatively small and, working with an ARM-based architecture, it is supposed that the developed architecture is an embedded oriented system, so the context switches number is not too big. Such a structure allows to avoid adding the PID field because all the date belong to the same process.

Chapter 3: MMU Performance evaluation

This chapter introduces the Control Unit implementation and contains the following sections:

1 AMAT

In order to evaluate the MMU system performance the AMAT (Average Memory Access Time) is estimated.

With two cache levels, the equations for AMAT estimation are:

$$AMAT = L1HitTime + L1MissRate * L1MissPenalty$$

$$L1MissPenalty = L2HitTime + L2MissRate * L2MissPenalty$$

$$L2MissPenalty = MainMemoryLatency$$

so:

$$AMAT = L1HitTime + L1MissRate * (L2HitTime + L2MissRate * MainMemoryLatency)$$

By designing MMU, different choices have been taken in order to minimize AMAT and achieving the fixed main purpose of the architecture: to keep a good trade-off among cost, complexity and performance.

2 Hit time

VIPT addressing has been adopted to reduce hit time for the first level cache.

Through VIPT, translation of virtual address and the access in cache occur at the same time without adding extra bits in order to manage the aliasing phenomena.

Without VIPT, translation of virtual address can’t be executed in parallel with the operation of accessing in cache.

With VIPT:

$$L1HitTime = \max\{Tcache, Ttranslation\} + Tcompare$$

$Tcompare$ is the time needed to compare tag in cache and physical tag. Basically, the comparison must occur after the translation.

Since $Ttranslation$ is the time for accessing the TLB, for instance TLB miss is ignored, usually the time to access and get data in cache is bigger than time to translate virtual address (TLB is a smaller cache than L1 cache). For this reason, translation time does not impact on $L1HitTime$. Thus, in the most cases $L1HitTime$ can be considered in this way:

$$L1HitTime = Tcache + Tcompare$$

Without VIPT:

$$L1HitTime = Ttranslation + Tcache + Tcompare$$

$Tcompare$ is the time to compare the tag in cache and physical tag. The comparison occurs after the accessing in cache.

3 Miss rate

A 4-way set-associative mechanism has been adopted in L1 cache in order to minimize $L1MissRate$, preserving the constraints on dimension imposed by the nature of chosen addressing methodology and the constraints on hardware, considering that the L1 cache is on the same processor chip.

4 Miss penality

The factor that can be minimized in order to reduce $L1MissPenalty$ is $L2MissRate$.

Considering that L2 cache is not on the processor chip, there are no strict constraints on hardware usage. For this reason, an 8-way set associative mechanism is adopted, with L2 cache dimension equal to 256kB.

5 Write strategy

The write strategy adopted is write through with write buffer both for L1 cache and L2 cache.

It is not adopted write back strategy, although this strategy does not need the addition of extra hardware, because with write through has been observed an improvement on writing time and, with some considerations, the stalls due to writing are reduced.

The observations made are based on a qualitative comparison between the two strategies.

By supposing that the accessing time in L1 cache takes a unitary time and the accessing time in L2 cache and Main Memory are respectively 5 times and 50 times the accessing time in L1:

$$t_{cacheL1} = 1$$

$$t_{cacheL2} = 5 * t_{cacheL1} = 5$$

$$t_{mem} = 50 * t_{cacheL1} = 50$$

In the current architecture the block size is:

$$B_{size} = 64B$$

Through write back strategy when a block has to be replaced its corresponding dirty bit is checked, and if dirty bit is 1 the block has to be transferred into the underlying memory level.

R_1 is 1 if a replacement is requested in L1 cache, otherwise 0
R_2 is 1 if a replacement is requested in L2 cache, otherwise 0
D_1 is 1 if dirty is 1 in L1 cache, otherwise 0
D_2 is 1 if dirty is 1 in L2 cache, otherwise 0

$$T_{store} = t_{cacheL1} + R_1 * (t_{cacheL1} + D_1 * 16 * T_{writeL2})$$

$$T_{writeL2} = t_{cacheL2} + R_2 * (t_{cacheL2} + D_2 * 16 * t_{mem})$$

so:

$$T_{store} = t_{cacheL1} + R_1 * [t_{cacheL1} + D_1 * 16 * (t_{cacheL2} + R_2 \\ * (t_{cacheL2} + D_2 * 16 * t_{mem})]$$

When D = 1 is needed to access 16 times in L2 cache or main memory in order to transfer the block in the underlying memory level, because the presented architecture is a 32-bit architecture. For this reason, transferring a whole block within the system means accessing the bus 16 times.

$$(R1 = 0)$$
$$T_{store} = t_{cacheL1} = 1$$

$$(R1 = 1)$$

$$T_{store} = \ t_{cacheL1} + R_1 * [t_{cacheL1} + D_1 * 16 * (t_{cacheL2} + R_2 * (t_{cacheL2} + D_2 * 16 * t_{mem})]$$

$$(R1 = 1, D1 = 0)$$

$$T_{store} = 2 * \ t_{cacheL1} = 2$$

$$(R1 = 1, D1 = 1, R2 = 0)$$

$$T_{store} = t_{cacheL1} + R_1 * [t_{cacheL1} + D_1 * 16 * (t_{cacheL2} + R_2 * (t_{cacheL2})] = 2 + 16 * 5 = 82$$

R2 = 1 only for the first access in L2 cache, when the block is replaced R2 = 0.

$$(R1 = 1, D1 = 1, R2 = 1\ (one\ time), D2 = 0)$$

$$T_{store} = t_{cacheL1} + R_1 * [t_{cacheL1} + D_1 * 16 * (t_{cacheL2} + R_2 * (t_{cacheL2})] = \ 2 \ + \ 1 * 10 \ + \ 15 * 5 \ = \ 87$$

$$(R1 = 1, D1 = 1, R2 = 1\ (one\ time), D2 = 1)$$

$$T_{store} = t_{cacheL1} + R_1 * [t_{cacheL1} + D_1 * 16 * (t_{cacheL2} + R_2 * (t_{cacheL2} + D_2 * 16 * t_{mem})] = \ 2 \ + \ 1 * (10 + 16 * 50) + \ 15 * 5 = \ 887$$

Through write through strategy is possible to replace the block into cache without checking dirty bit value. The writing into buffer and into cache occurs in parallel.

$$T_{store} = t_{cacheL1} + R1 * t_{cacheL1}$$

$$(R1 = 0)$$

$$T_{store} = t_{cacheL1} = 1$$

$$(R1 = 1)$$

$$T_{store} = t_{cacheL1} + R1 * t_{cacheL1} = 2$$

Using write through strategy with write buffer implies a huge reduction of time for store rather than the cases in which R1 = 1 and D1 = 1 with write back strategy.

The introduction of write buffer determines an increasing of stall cycle in memory.

The stall cycle in memory can be split out into stall cycle in writing and stall cycle in reading.

Stall cycle in reading is:

$$Stalls\ Cycles\ in\ Writing = \left[\frac{Writing}{Program} * Miss\ Rate\ in\ Writing\right] + Stalls\ Number\ of\ Buffer\ in\ Writing$$

The last term of equation is linked to introduction of write buffer.

It accounts the stalls due to the filling of buffers during the writing operations.

Buffer stalls due to how much writing operations are close in time and is not possible to define a simple equation in order to count buffer stalls occurrences.

However, if the depth of buffers is enough (four or more words) and the writing rate of memory is enough bigger then the writing rate (e.g. a factor of two) the stalls occurrences are very few and it can be omitted.

In the presented architecture are used buffers with depth of four and eight, and it's supposed to work with a modern memory with small latency, so the $Stalls\ Number\ of\ Buffer\ in\ Writing$ is not significative.

Chapter 4: Resources

This chapter introduces the resources involved in the design and contains the following sections:

- *Resources allocation* on page 19
- *Resources timesheet* on page 19

Resources allocation

Resources per leadership

Leadership	Member
Team Leader	Fernando Manna
CPU Leader	Giovanni Di Prisco
MMU Leader	Edoardo Cossentino
Documentation Leader	Francesco de Pertis

Resources timesheet

Sub-team	Team members	Tot. work hours d3
CPU	Fernando Manna	8
	Giovanni Di Pisco	8
	Marco Schettini	8
	Ilaria Gigi	8
	Mario Mupo	2
	Romeo Rinaldi	2
	Michele Rescigno	2
	Ascanio Guglielmelli	2
MMU	Andrea Maione	8
	Antonio Coppola	8
	Edoardo Cossentino	8

	Giuseppe Mascolo	2
Documentation	Francesco de Pertis	2
	Giuseppe Cirillo	2
	Antonino Durazzo	2