# Processor Architecture

## Processor design

*Autumn - 2015*

Professor: Roger Espasa

David Trilla Rodríguez
Cristóbal Ortega Carrasco
Constantino Gómez Crespo

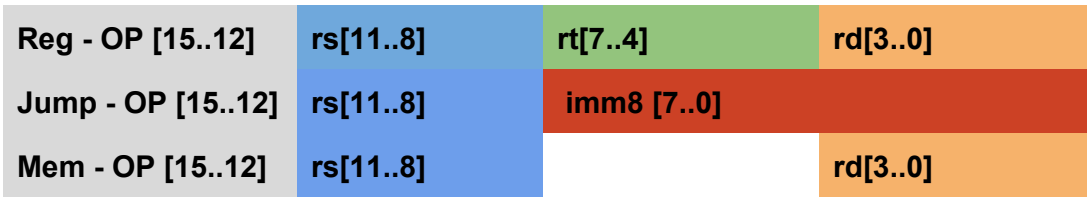
# 1 Architecture

## Summary

This is the general features of the processor we wanted to implement.

| Element | Value |
|---|---|
| Registers | #16 - 16 bits |
| LOAD & STORE | 8 / 16 bits |
| Instructions | RISC - 16 bits |
| Memory space | 16 bits |
| Virtual memory | 1 KB page size |
| Endianess | Little Endian |

## Instruction Set Architecture




**load word - lw**

---

**Opcode** 0000
**Format** lw rd, rs
**Description** rd ← MEM([rs])

**Operation:**

- virtAddr = R[rs]
- (physAddr, isCached) ← (translate(addr))
- memword ← LoadMem(physaddr, isCached)
- R[rd] ← memword

**Exceptions:**

- TLBmiss @ translate(addr)

**load byte - lb**

---

**Opcode** 0001
**Format** lb rd, rs
**Description** rd ← MEM([rs]);

**Operation:**

- virtAddr = R[rs]
- (physAddr, isCached) ← (translate(addr))
- membyte ← LoadMem(physaddr, isCached)
- R[rd] ← membyte

**Exceptions:**

- TLBmiss @ translate(addr)

**store word - sw**

---

**Opcode** 0010
**Format** sw rd, rs
**Description** MEM([rs]) ← rd;

**Operation:**

- virtAddr = R[rs]
- (physAddr, isCached) ← (translate(addr))
- StoreMem(physaddr, isCached, R[rd])

**Exceptions:**

- TLBmiss @ translate(addr)

**store byte - sb**

---

**Opcode** 0011
**Format** sb rd, rs
**Description** MEM([rs]) ← rd;

**Operation:**

- virtAddr = R[rs]
- (physAddr, isCached) ← (translate(addr))
- StoreMem(physaddr, isCached, R[rd])

**Exceptions:**

- TLBmiss @ translate(addr)

**integer add - add**

---

**Opcode** 0100
**Format** add rd, rs, rt
**Description** rd ← rs + rt;

**Operation:**

- R[rd] = R[rs] + R[rt]

**integer sub - sub**

---

**Opcode** 0101
**Format** sub rd, rs, rt
**Description** rd ← rs - rt

**Operation:**

- R[rd] = R[rs] - R[rt]

**compare equals**

---

**Opcode** 0110
**Format** ceq rd, rs, rt
**Description** (R[rs] == R[rt])? R[rd] = 1 : R[rd] = 0;

**Branch NOT Zero - BNZ s**

---

**Opcode** 0111
**Format** bnz rs, imm8
**Description** if rs != 0 then branch

**Operation:**

- Z ← (R[rs] == 0)
- if (!Z) then: PC ← PC + Extend-logic-to-16(imm8)

**Long instruction**

---

| | |
|---|---|
| **Opcode** | 1000 |
| **Format** | lnginstr rd, rs |
| **Description** | rd ← rs + 4 |

**Operation:**

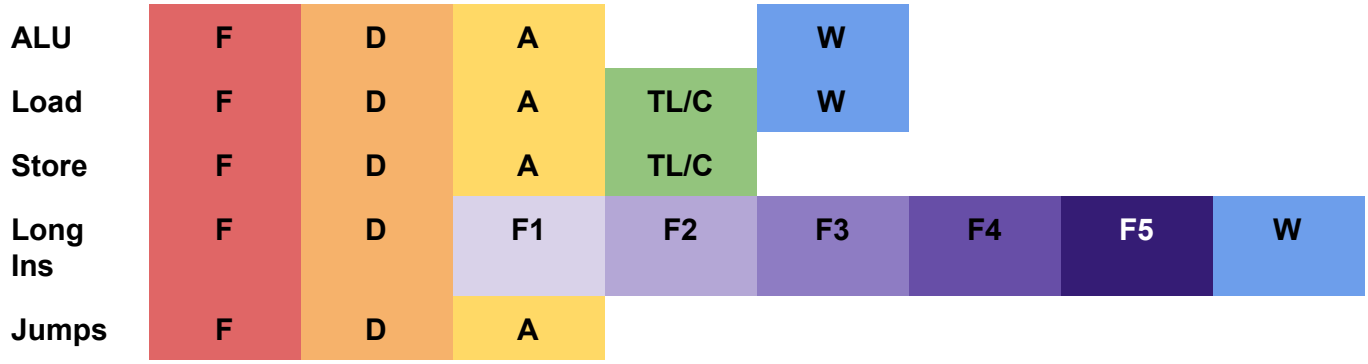
- R[rd] ← R[rs] + 1 + 1 + 1 + 1

**NOP**

---

| | |
|---|---|
| **Opcode** | 1111 |
| **Format** | nop |
| **Description** | disables |

# 3 Microarchitecture




### Structural hazards

We might find structural hazards in the writeback stage.

| decoding / pipeline | ALU | LD | ST | LONG | JMP |
|---|---|---|---|---|---|
| **ALU** | -- | -- | -- | **3** | -- |
| **LD** | -- | -- | -- | **3** | - |
| **ST** | -- | -- | -- | -- | -- |
| **JMP** | -- | -- | -- | -- | -- |
| **LONG** | -- | -- | -- | -- | -- |

### Structural hazard management

We will stall the instructions at Fetch and Decode stages whenever a Structural hazard is detected. NOP instructions are injected to fill the gaps in the pipeline.

**Operation**

```
Condition ← ( f3_inst = "0001" & decode_inst = (alu or load) )
if Condition then
      fetch_decode.stall ← 1
      fetch.stall ← 1
      decode.op ← NOP
```

```
end
```

### Data hazards

A data hazard may happen when a younger instruccion consumes data produces but an older but close instruction. The table represents the minimum distance between instructions with true dependencies.

| prod \ consum | ALU | LD | ST | JMP | LONG |
|---|---|---|---|---|---|
| **ALU** | 3 | 3 | 3 | 3 | 3 |
| **LD** | 3 | 3 | 3 | 3 | 3 |
| **ST** | -- | -- | -- | -- | -- |
| **JMP** | -- | -- | -- | -- | -- |
| **LONG** | 6 | 6 | 6 | 6 | 6 |

Keep in mind that we can write and read during the same cycle, and there's no additional bypasses.

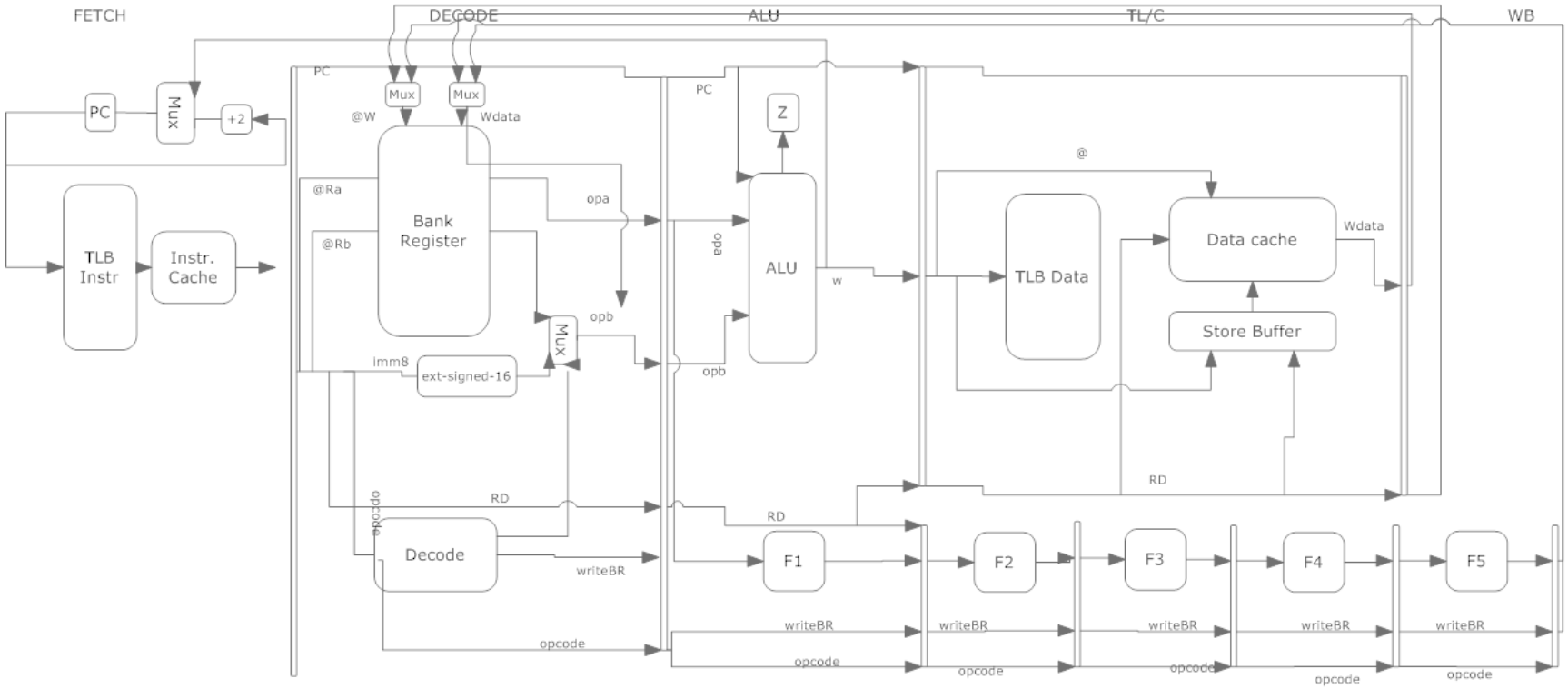
### Data hazard management

Data hazards will also be manage during the decode stage, if a hazard is found between the current instruction and another instruction forward in the pipeline, we will stall the decode and previous stages, and let the rest of the pipeline running until the hazard is solved.

**Operation**

```
Condition ← ( f3_inst = "0001" & decode_inst = (alu or load) )
if Condition then
        fetch_decode.stall ← 1
        fetch.stall ← 1
        decode.op ← NOP
end
```

### Block Diagram


FETCH
DECODE
ALU
TL/C
WB
PC
MUX
+2
TLB Instr
Instr. Cache
PC
Mux
Mux
@W
Wdata
@Ra
@Rb
Bank Register
opa
opb
MUX
imm8
ext-signed-16
opcode
RD
Decode
writeBR
opcode
PC
Z
opa
opb
ALU
w
RD
F1
writeBR
opcode
@
TLB Data
Data cache
Wdata
Store Buffer
RD
F2
writeBR
opcode
F3
writeBR
opcode
F4
writeBR
opcode
F5
writeBR
opcode

## Memory hierarchy

We have implemented 3 levels of memory. Registers, cache and external ram. Both Instruction cache and Data cache are independent.

Our cache is a direct mapping, of size 128 bytes with 16 lines of 8 bytes long. We have a write-through, no-allocate policy for stores, hence, no dirty line control is required. The instruction cache is checked during the FETCH stage and the data cache is checked during the TL/C stage.

## IRF - Integer register file

We have 16 available 16 bit registers in a Integer Register File. Two read ports A and B and one write port W.

**Writes**

During the rising edge, if writeBR is enabled we will write the value specified in w in the register given by regDST. During the lower edge writing is disabled.

**Reads**

We output by the BR ports A and B the values at the registers specified by @A and @B.

Registers are R0-R15 and are codified from 0000 tol 1111

## On miss behavior

When a miss occurs on the tags of the cache, issues a miss signal. The control then stalls the pipeline to the cache stage and backwards and a request to memory is processed. After 4 cycles we fill the cache with two burts of 4 bytes each (1 line of cache). This behaviour is applied to the Icache and Dcache.

## On TLB miss behavior (not implemented)

When a Load / Store instruction enters the TL/Cache stage the address of memory in the pipeline is a virtual address. The TLB module translates the tag of the direction to a physical address tag, then, the cache uses the translated tag to check for hit/miss.

When we miss at the TLB a TLB exception is issued. How we manage this exception is described below in exceptions section.

### Jumps

Jump instructions are commited during the ALU stage, two cycles after the instruction enters the pipeline. In that exact cycle, PC will adquire the jump address and a NOP will be injected into the DECODE stage to kill the instruction loaded after the jump that don't have to be executed.

### Information spread through the pipeline

The instruction will advance through the pipeline along with complementary information needed to control the execution.

- PC
    - Enables support for precise exceptions. Used as a ret address when a TLB exception or interruption occurs. Used to compute the branch new PC.
- Op code
    - Basic control of the pipeline.
- regDest
    - Destination register is needed to write in the register bank during the WriteBack stage. Store
- A and B values
    - Spread until Cache stage. The semantic of the value depends on the Op code.
- Write enable in BR
    - Allows us to control whether a instruction in the pipeline has to write in the BR or not.
- Exception vector (not implemented)
    - Whether the instruction produced an exception and which one.

### What we can do at stages:

- Stall  all datapath (memory misses)
- Stall fetch and decode and insert nops in the pipeline (data and structural hazards)

### Exceptions (not implemented)

In our datapath we can only get exceptions caused by:

- TLB miss in the instruction cache
- TLB miss in the data cache

- Not permitted operation in the Execution stage (divide by 0)

Given that, we do not need a Reorder Buffer or History File to offer to the user precise exceptions.

If we get an exception we save the state of the instruction that gave the exception in our special registers, then we proceed to change the PC in the Fetch stage to go to the exception routine.

### Interruptions (not implemented)

When an interruption comes, we save the state in the Decode stage (PC + 2) and the value of the interruption in the special register, and then we start to insert NOPs, meanwhile we change the PC to the Fetch stage to go to the interruption routine.