**Instruction Format**

* IR register (16 bit)

|  |  |  |  |
| --- | --- | --- | --- |
| Op code  (2 bits) | Func  (3 bits) | Rsrc  (4 bits) | Rdst  (4 bits) |

* Op code

|  |  |
| --- | --- |
| 00 | One operand |
| 01 | Two operands |
| 10 | Memory |
| 11 | Branch |

* One operand function (00)

|  |  |
| --- | --- |
| 000 | No op |
| 001 | Set carry |
| 010 | Clear carry |
| 011 | Not rdst |
| 100 | Increment rdst |
| 101 | Decrement rdst |
| 110 | Out rdst |
| 111 | In rdst |

* Two operand functions (01)

|  |  |
| --- | --- |
| 000 | Mov |
| 001 | add |
| 010 | Sub |
| 011 | And |
| 100 | Or |
| 101 | Shift left |
| 110 | Shift right |

* Memory functions (10)

|  |  |
| --- | --- |
| 000 | push |
| 001 | pop |
| 010 | Load imm |
| 011 | Load from memory |
| 100 | Store to memory |

* Registers

|  |  |
| --- | --- |
| R0 | 0000 |
| R1 | 0001 |
| R2 | 0010 |
| R3 | 0011 |
| R4 | 0100 |
| R5 | 0101 |
| R6 | 0110 |
| R7 | 0111 |
| SP | 1000 |
| PC | 1001 |
| Flag | 1010 |

**Instruction details**

**One operand**

* No op

IR 🡪 all zero ( one operand – no op)

WB, MR, MW 🡪 zero

ALU 🡪 no op

* Set carry

IR 🡪 (one operand – set carry – rdst =0 – rsrc = 0)

WB, MR, MW 🡪 zero

ALU 🡪 set carry

* Clear carry

IR 🡪 (one operand – clear carry – rdst =0 – rsrc = 0)

WB, MR, MW 🡪 zero

ALU 🡪 set carry

* Not dst

IR 🡪 (one operand – not – rdst = dst - rsrc =0 )

WB ,MR 🡪 zero

WB 🡪one

ALU 🡪 not

**Pipeline register details**

* F/D Buffer (16 bit)

|  |
| --- |
| I  R |

* D/E Buffer (48 bit)

|  |
| --- |
| Rdst (3 bit ) |
| Rsrc (3 bit ) |
| Branch taken (1 bit) |
| Load use (1 bit) |
| Rsrc data (16 bit) |
| Rdst data (16 bit ) |
| Stall\_long (1 bit) |
| WB (1 bit ) |
| Memory read (1 bit) |
| Memory write (1 bit) |
| Alu op (4 bit ) |

* E/M (23 bit)

|  |
| --- |
| Rdst (3 bit ) |
| Wb (1 bit) |
| Stall\_long (1 bit) |
| Memory read (1 bit ) |
| Memory write (1 bit) |
| Alu result (16 bit) |

* M/W (38 bit)

|  |
| --- |
| Rdst (3 bits) |
| Wb (1 bit) |
| Memory read (1 bit) mux select |
| Stall\_long (1 bit) |
| Alu result (16 bit) |
| Memory result (16 bit) |

**Schematic diagram Blocks**

* **Control unit**

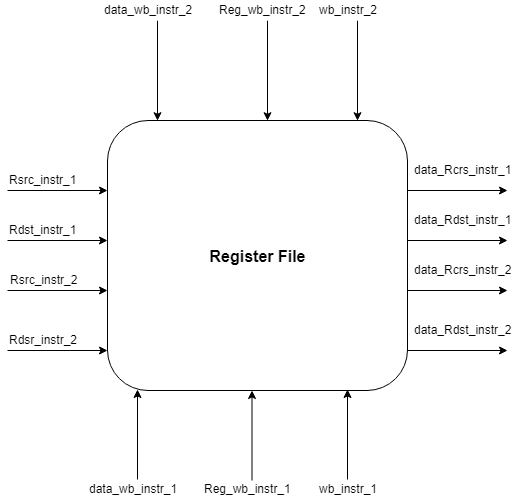
instruction alu (4 bit)

MR(1 bits)

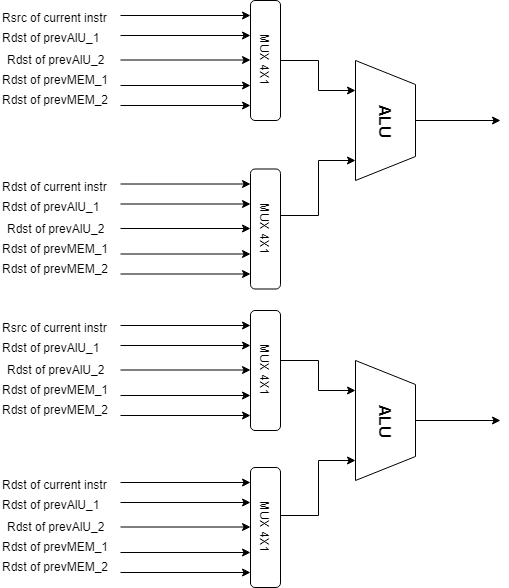
Rdst (3 bit)

Wb (1 bit) Mw (1 bit)

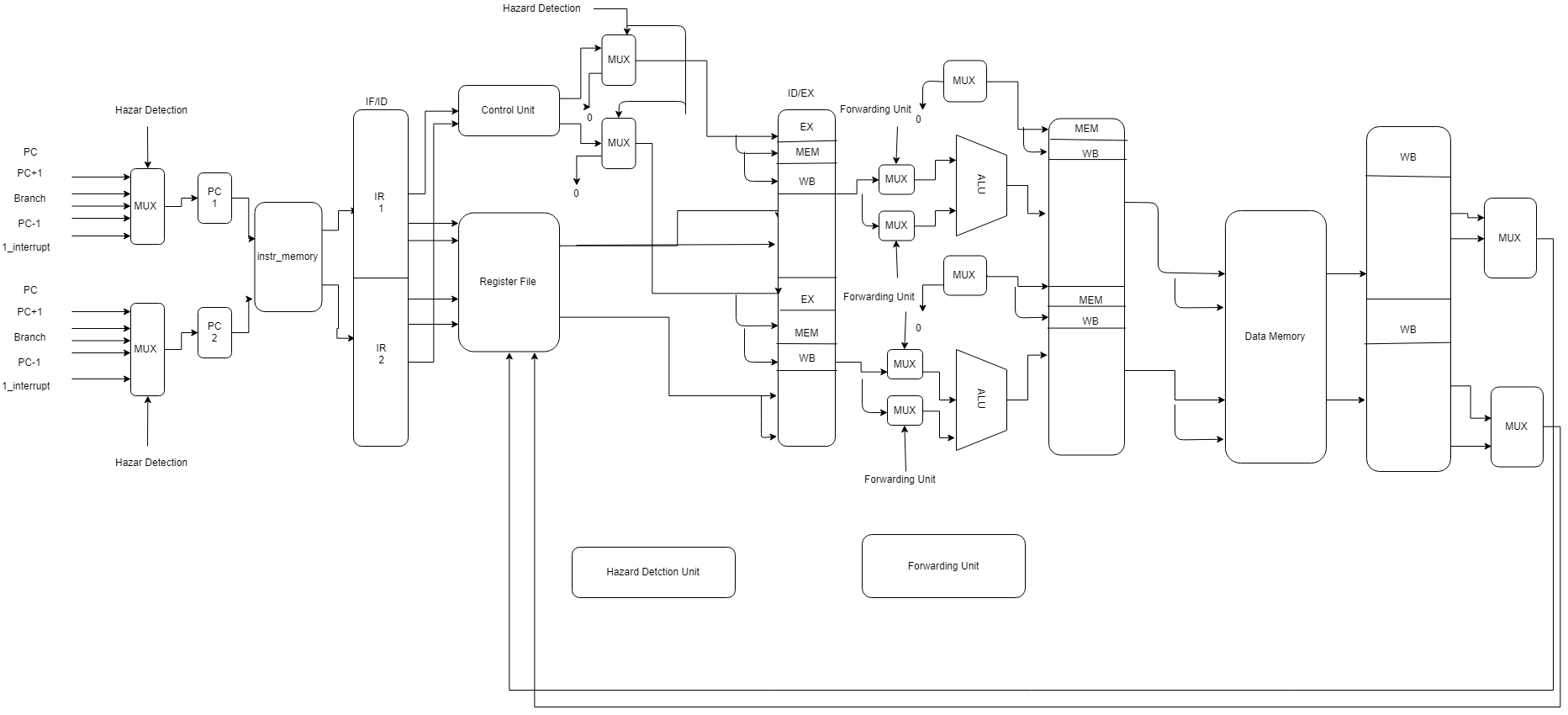
* **Register File**



* Alu



Mux selectors form forwarding unit



**Hazards Solution**

1. **Data hazard**

Solution 🡪 forwarding between two different packets

Data Forwarding unit:

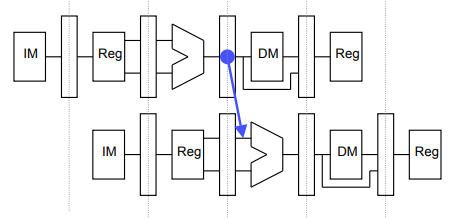
* A forwarding unit selects the correct ALU inputs for the EX stage to solve any data hazards.

1. If there is no hazard, the ALU’s operands will come from the register file (selectors = "000").
2. If there is a hazard, the operands will come from either the EX/MEM or MEM/WB pipeline registers instead.

* The ALU sources will be selected by two multiplexers, with two selectors which controlled by the forwarding unit.
* There are two kinds of data hazards.

1. EX/MEM data hazards.
2. MEM/WB data hazards.

Detecting EX/MEM data hazard



An EX/MEM hazard occurs between one of the instructions of the issue currently in its EX stage and the previous two instructions in the previous issue if:

1. One of the previous instructions will write to the register file, and
2. One of the destination registers is one of the ALU source registers in the EX stage.

example on EX/MEM data hazard:

I1. add R1, R2

I2. sub R6, R7

I3. or R5, R6

I4. and R3, R4

here there is a data hazard between l3 instruction from the second packet of instructions and l2 instruction from the first one.

The ALU source comes from the pipeline register when necessary.

if (EX/MEM.RegWrite1 = 1 and EX/MEM.RegisterRd1 = ID/EX.RegisterRs1)

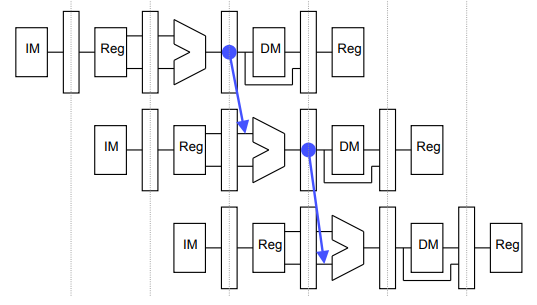
then forward\_a1 = "001" //choose Rdst of the prev. alu of the first instruction in the previous packet .

if (EX/MEM.RegWrite2 = 1 and EX/MEM.RegisterRd2 = ID/EX.RegisterRs1)

then forward\_a1 = "010" //choose Rdst of the prev. alu of the second instruction in the previous packet.

The same checks are done for the second operand of the first instruction of the packet, also for the second instruction in the packet.

Detecting MEM/WB data hazards



A MEM/WB hazard may occur between an instruction in the EX stage and

the instruction from two cycles ago.

For detecting and handling MEM/WB hazards for the first ALU source.

if (MEM/WB.RegWrite1 = 1 and MEM/WB.RegisterRd1 = ID/EX.RegisterRs1)

then forward\_a1 = "011" //choose Rdst of the alu of the first instruction in the earlier packet.

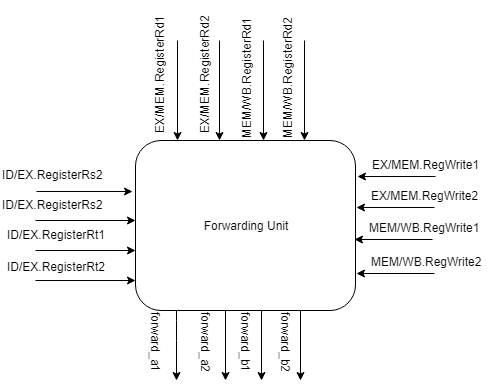
if (MEM/WB.RegWrite1 = 2 and MEM/WB.RegisterRd2 = ID/EX.RegisterRs1)

then forward\_a1 = "100" //choose Rdst of the alu of the second instruction in the earlier packet.

The same checks are done for the second operand of the first instruction of the packet, also for the second instruction in the packet.

The forwarding unit

The forwarding unit has several control signals as input and outputs 2 selectors for each instruction in the packet.



**Hazard Detection**

As we have mentioned that we will fetch packet of instructions and execute those using concept of parallelism. The packet consists of only two instructions. These instructions may depend on each other and this dependency will cause a problem called hazards.

**Hazards Types**

* Structural Hazard
* Control Hazard
* Data Hazard

**Structural Hazard**

**Structural hazard occurs in the following situation: -**

* Instruction Fetch is conflicting with data memory access.
* **Solution** is to use two separated memory (Data and Instruction).
* Register File is accessed by the instruction in the **decode** stage (reading) and at the same time other instruction in the **write-back** stage.
* **Solution** is to ensure that writing is done in the falling edge and reading is done in the rising edge.
* Packets is executed in parallel and no sufficient resources
* **Solution** is to duplicate the resources. We will have two ALUs, two MAR, two MDR, Register File has two signals **WB** (WB1 for the first instruction in the packet and WB2 for the second instruction in the packet).

**Data Hazards Types**

* Data Inner Hazard
* Data Outer Hazard

**Data Inner Hazard**

This type of hazards occurs when the packet itself is dependent. For example, if I have a load in the first instruction that uses some registers and the second instruction is using the same register to do some logic in the program. Assume code has only three instructions (**Inst1 to Inst3**) and the first, second cause data hazard because of dependency.

**Solution** is to stall the pipeline until this instruction will be written back in the register file after doing the logic of first instruction then start fetch a new packet and this new packet will contains the following **Inst2**--**Inst3.**

**How this solution actually works behind the scenes?!**

The Hazard Detection Unit **(HDU)** takes inputs from the previous, current packet. The inputs are the **IR** of the first packet instructions (Two IR) and the **IR** of the second packet instructions (Two IR). **Write after write may also make conflicts in the register.**

check\_one = (DEC\_Rsrc2 = DEC\_Rsrc1 AND DEC\_Inst1=One\_Operand)

check\_two = (DEC\_Rsrc2 = DEC\_Rdst1 AND DEC\_Inst1=Two\_Operand)

check\_waw = (DEC\_Rdst1 = DEC\_Rdst2)

stall\_long = check\_one OR check\_two or check\_waw

**stall\_long** is responsible for making the first instruction in the packet resume executing (only it) for 3 clock cycles using crawling **stall\_long** in the pipeline and take the result of it in the last buffer. How! By disabling the (IR Register) which means that we add to the pipeline **NOP** operations until the desired instruction finished and disabling the **PC** to keep the next instruction but we need after that to decrement it by 1 to fetch the correct packet.

Rst\_IR = !(EXE\_stall\_long OR MEM\_stall\_long)

enable\_PC = !(EXE\_stall\_long)

pc = pc – 1 when MEM\_stall\_long.

Then our new fetch will be correct isa.

**Outer Hazard**

This type of hazards occurs when an instruction (or more) in the packet depend on previous packet (**Load-Use**). For example we’ve 4 instruction **Inst1 to Inst4**. Assume Inst2 is LDD Rdst, Rsrc and inst3 need to use Rdst. We’ve 4 cases **(Inst3 and Inst2), (Inst3 and Inst1), (Inst4 and Inst1), (Inst4 and Inst2)**

**Solution** is to stall the pipeline one cycle (stall bit is added in the control word).

**How to stall the pipeline?!**

* Disabling The **IR** & **PC** (Don’t take the new instruction that the **PC** stored its address and don’t increment **PC**).
* Reset The Buffer between the decoder stage and execution stage (act as **NOP**).

**Stall Logic**

stall = EXE\_Memory\_Read AND ( ( EXE\_Rdst1 = DEC\_Rsrc1 )

OR ( EXE\_Rdst1 = DEC\_Rsrc2 )

OR ( EXE\_Rdst2 = DEC\_Rsrc1 )

OR ( EXE\_Rdst2 = DEC\_Rsrc2 ))

**Control Hazard**

**Control hazard occurs in the following situation:-**

* **Branch Taken (Jump)**
* Static prediction is not to take the branch and continue the program **(NOT-TAKEN).**
* We know that the instruction is branch in the decode stage and we raise one bit called **branch\_taken** if taken or not.
* In the execution if the branch\_taken is raised then we need to flush else continue the programs (no problem).
* **Load Immediate**
* Some instructions take a load value from the next instruction so I’ve to take this value and put it in the control signal then flushing the instruction (actually the instruction is a value not an instruction).

**Solution** is to flush the new instruction or (value).

**How to flush the instruction or value?!**

* Raise signal **(flush)** (acts as **NOP**), we may need to RST the IR (explained in the branch when occupies the 1st place in the packet).

**There are two cases**

* The branch instruction is in the first instruction of the packet
* The branch instruction is in the second instruction of the packet

Each case need a different handler. For the case #1 assume we have this code

1. JZ Rdst
2. Add R1, R2

So in the above example we need to flush the 2nd instruction in the packet and the next packet once we knew that the branch is taken.

For case #2

1. Add R1, R2
2. JZ Rds

But in the above example we’ve to complete the 1st instruction in the packet till the end then we back to case #1.

stall\_long = (DEC\_Inst2 = Branch)

Rst\_IR = !(EXE\_stall\_long OR MEM\_stall\_long)

enable\_PC = !(EXE\_stall\_long)

pc = pc – 1 when MEM\_stall\_long.

**Flush Logic**

Flush = immediate\_load OR branch\_taken.

PC = effective\_address when branch\_taken.

Rst\_IR = branch\_taken (flush second packet)

**Immediate Load**

**There are two cases**

* The load instruction is in the first instruction of the packet
* The load instruction is in the second instruction of the packet

We always need the load instruction (**LDM**) to be the first in the packet, so if it occupies in the second instruction in the packet we will make the slight same logic of the **data-inner-hazard**.

Each case need a different handler. For the case #1 assume we have this code

1. LDM R5, 10
2. 10

In the above example we raise only the signal of flush that will make the mux pass zeros to the alu.

For the case #2 assume we have this code

1. Add R1 , R2
2. LDM R5, 10

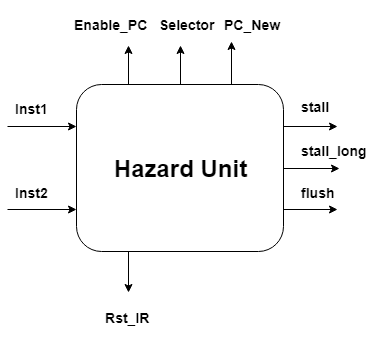
stall\_long = (EXE\_INST2 = LDM)

Rst\_IR = !(EXE\_stall\_long OR MEM\_stall\_long)

enable\_PC = !(EXE\_stall\_long)

pc = pc – 1 when MEM\_stall\_long.

**Hazard Detection (The Easy Way).**



**Control Signals**

* Alu Control

|  |  |
| --- | --- |
| **combination** | **F =** |
| 0000 | No op |
| 0001 | Set carry (one op) |
| 0010 | Clear carry (one op) |
| 0011 | Not (one op) |
| 0100 | Increment (one op ) |
| 0101 | Decrement (one op) |
| 0110 | Rdst (one op ) |
| 0111 | Add |
| 1000 | Sub |
| 1001 | And |
| 1010 | Or |
| 1011 | Shift left |
| 1100 | Shift right |
| 1101 | out |
| 1110 | in |