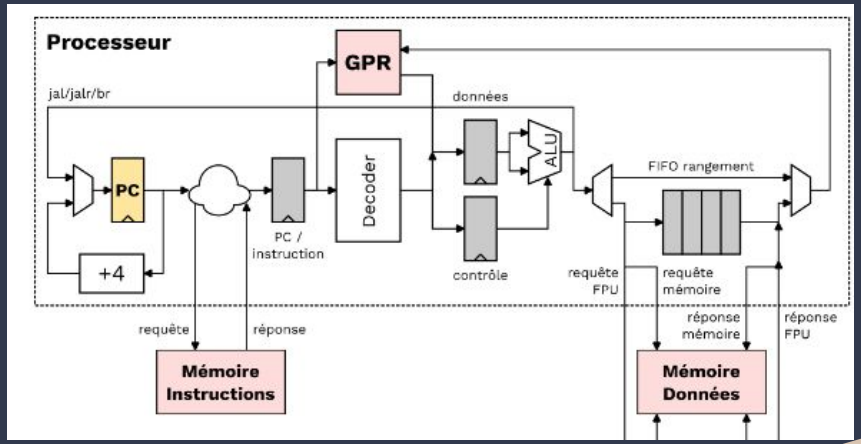
Projet RISC-V Chisel

Oscar Guillerault
Noah Milien

Sommaire

I/ Introduction



Processeur RISC-V :

- Instructions Arithmético-logiques
- Instructions de saut/branchement

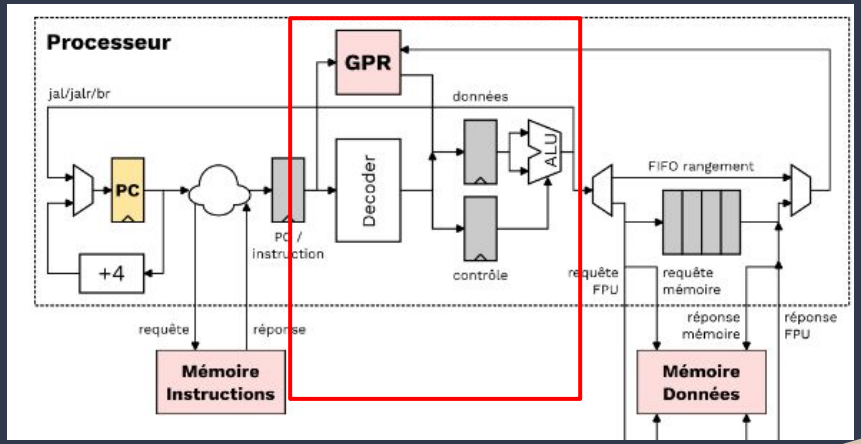
Langage Chisel :

- Langage de description matériel
- Basé sur Scala

3 parties :

- Back-end
- Front-end
- Gestion mémoire

II/ Front-end = EXEC



Trois blocs principaux fonctionnels :

- Registres généraux GPR
- Décodeur
- ALU

1. ALU

```
// Module ALU
class ALU extends Module {
  // Définit les entrées/sorties du module
  val io = IO(new Bundle {
    val i_rs1 = Input(UInt(32.W))
    val i_operande = Input(UInt(32.W))
    val funct_sel = Input(UInt(5.W))
    val o_rd = Output(UInt(32.W))
  })


val rs1 = io.i_rs1
val operande = io.i_operande
val funct = io.funct_sel
val res = WireDefault(0.U(32.W))

 switch(funct){
  is(0.U) {res := rs1 + operande}
  is(1.U) {res := rs1 - operande}
  is(2.U) {res := rs1 ^ operande}
  is(3.U) {res := rs1 | operande}
  is(4.U) {res := rs1 & operande}
  is(5.U) {res := rs1 << operande(4,0)}
  is(6.U) {res := rs1 >> operande(4,0)}
  is(7.U) {res := (rs1.asSInt >> operande(4,0)).asUInt}
 }

 io.o_rd := res
}
```

1er module décrit:

=> Prise en main du langage

Utilisation d'une variable servant à l'identification de l'opération.

Entrées :

- Registre 1
- Registre 2 / Immédiat
- Variable d'opération

Sortie :

- Registre de sortie

2. GPR

```
class GPR extends Module {
  // Définit les entrées/sorties du module
  val io = IO(new Bundle {

    //Ecriture
    val i_data = Input(UInt(32.W))      //Donnée en entrée des registres
    val i_write = Input(Bool())         //Activation de l'écriture sur le registre
    val i_sel_reg = Input(UInt(5.W))    //Numero du registre d'écriture

    //Lecture
    val i_read_reg1 = Input(UInt(5.W))  //Numero du registre entre 0 et 31 = 5 bits
    val i_read_reg2 = Input(UInt(5.W))

    val o_data_reg1 = Output(UInt(32.W))
    val o_data_reg2 = Output(UInt(32.W))

  })


  val registerFile = Reg(Vec(32,UInt(32.W)))  //File de 32 registres de 32 bits

  when(io.i_write) {        //Ecriture
    registerFile(io.i_sel_reg) := io.i_data
  }

  io.o_data_reg1 := registerFile(io.i_read_reg1)
  io.o_data_reg2 := registerFile(io.i_read_reg2)

  registerFile(0) := 0.U                    //registre x0
}
```

Registres généraux:

- 1 port d'écriture
- 2 ports de lecture (pour l'ALU)
- 32 registres de 32 bits
- Valeurs des registres de lecture en sortie

3. Décodeur

```
object OPE {
    def ADDI   = BitPat("b?????????????????000?????0010011")
    def XORI   = BitPat("b?????????????????100?????0010011")
    def ORI    = BitPat("b?????????????????110?????0010011")
    def ANDI   = BitPat("b?????????????????111?????0010011")
    def SLLI   = BitPat("b?????????????????001?????0010011")
    def SRLI   = BitPat("b?????????????????101?????0010011")
    def SRAI   = BitPat("b?1???????????????101?????0010011")
    def ADD    = BitPat("b?0???????????????000?????0110011")
    def SUB    = BitPat("b?1???????????????000?????0110011")
    def XOR    = BitPat("b?????????????????100?????0110011")
    def OR     = BitPat("b?????????????????110?????0110011")
    def AND    = BitPat("b?????????????????111?????0110011")
    def SLL    = BitPat("b?????????????????001?????0110011")
    def SRL    = BitPat("b?0???????????????101?????0110011")
    def SRA    = BitPat("b?1???????????????101?????0110011")
    def LUI    = BitPat("b???????????????????????????0110111")
    def AUIPC  = BitPat("b???????????????????????????0010111")
    def JAL    = BitPat("b???????????????????????????1101111")
    def JALR   = BitPat("b?????????????????000?????1100111")
    def BEQ    = BitPat("b?????????????????000?????1100011")
    def BNE    = BitPat("b?????????????????001?????1100011")
    def BLT    = BitPat("b?????????????????100?????1100011")
    def BGE    = BitPat("b?????????????????101?????1100011")
    def BLTU   = BitPat("b?????????????????110?????1100011")
    def BGEU   = BitPat("b?????????????????111?????1100011")
    def LB     = BitPat("b?????????????????000?????0000011")
    def LH     = BitPat("b?????????????????001?????0000011")
    def LW     = BitPat("b?????????????????010?????0000011")
    def LBU    = BitPat("b?????????????????100?????0000011")
    def LHU    = BitPat("b?????????????????101?????0000011")
    def SB     = BitPat("b?????????????????000?????0100011")
    def SH     = BitPat("b?????????????????001?????0100011")
    def SW     = BitPat("b?????????????????010?????0100011")
    def SLTI   = BitPat("b?????????????????010?????0010011")
    def SLTIU  = BitPat("b?????????????????011?????0010011")
    def SLT    = BitPat("b?????????????????010?????0110011")
    def SLTU   = BitPat("b?????????????????011?????0110011")

}
```

```
object TABLECODE{                                    //type(RISBU
    val default: List[UInt] = List[UInt]( 0.U,
    val table: Array[(BitPat, List[UInt])] = Array[(B
        OPE.ADDI          -> List(  1.U, 0.U, 0.U),
        OPE.XORI          -> List(  1.U, 2.U, 0.U),
        OPE.ORI           -> List(  1.U, 3.U, 0.U),
        OPE.ANDI          -> List(  1.U, 4.U, 0.U),
        OPE.SLLI          -> List(  0.U, 5.U, 0.U),
        OPE.SRLI          -> List(  0.U, 6.U, 0.U),
        OPE.SRAI          -> List(  0.U, 7.U, 0.U),
        OPE.ADD           -> List(  0.U, 0.U, 1.U),
        OPE.SUB           -> List(  0.U, 1.U, 1.U),
        OPE.XOR           -> List(  0.U, 2.U, 1.U),
        OPE.OR            -> List(  0.U, 3.U, 1.U),
        OPE.AND           -> List(  0.U, 4.U, 1.U),
        OPE.SLL           -> List(  0.U, 5.U, 1.U),
        OPE.SRL           -> List(  0.U, 6.U, 1.U),
        OPE.SRA           -> List(  0.U, 7.U, 1.U),
        OPE.LUI           -> List(  4.U, 0.U, 0.U),
        OPE.AUIPC         -> List(  4.U, 0.U, 0.U),
        OPE.JAL           -> List(  5.U, 0.U, 0.U),
        OPE.JALR          -> List(  1.U, 0.U, 0.U),
        OPE.BEQ           -> List(  3.U, 0.U, 0.U),
        OPE.BNE           -> List(  3.U, 0.U, 0.U),
        OPE.BLT           -> List(  3.U, 0.U, 0.U),
        OPE.BGE           -> List(  3.U, 0.U, 0.U),
        OPE.BLTU          -> List(  3.U, 0.U, 0.U),
        OPE.BGEU          -> List(  3.U, 0.U, 0.U),
        OPE.LB            -> List(  1.U, 0.U, 0.U),
        OPE.LH            -> List(  1.U, 0.U, 0.U),
        OPE.LW            -> List(  1.U, 0.U, 0.U),
        OPE.LBU           -> List(  1.U, 0.U, 0.U),
        OPE.LHU           -> List(  1.U, 0.U, 0.U),
        OPE.SB            -> List(  2.U, 0.U, 0.U),
        OPE.SH            -> List(  2.U, 0.U, 0.U),
        OPE.SW            -> List(  2.U, 0.U, 0.U),
        OPE.SLTI          -> List(  1.U, 0.U, 0.U),
        OPE.SLTIU         -> List(  1.U, 0.U, 0.U),
        OPE.SLT           -> List(  0.U, 0.U, 1.U),
        OPE.SLTU          -> List(  0.U, 0.U, 1.U),
    )
}
```

Description des paternes de chacune des instructions pour simplifier la manipulation des données.

Table d'association entre paterne et opérations :

- Type d'instruction (R/I/S/B/U/J)
- Opération (arithmético-logique)
- Avec ou sans immédiat

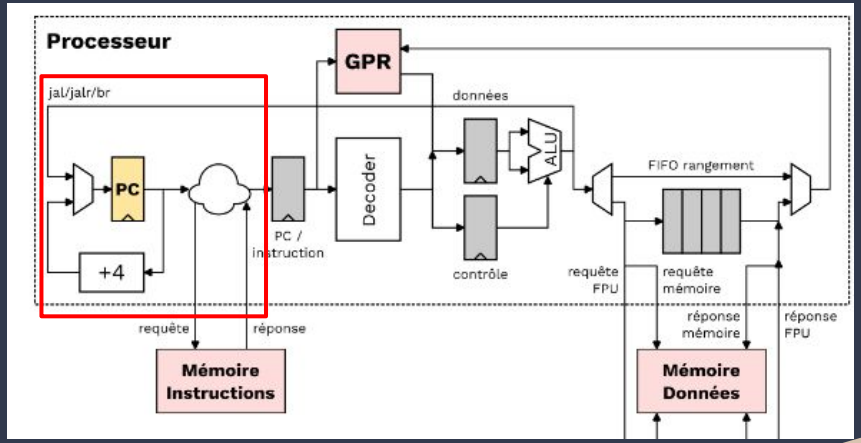
Entrée :

- Instruction

Sorties :

- Registres rs1, rs2, et rd
- Immédiat
- Signal d'écriture pour GPR
- Signaux d'activation de lecture/écriture mémoire

III/ Back-end = FETCH

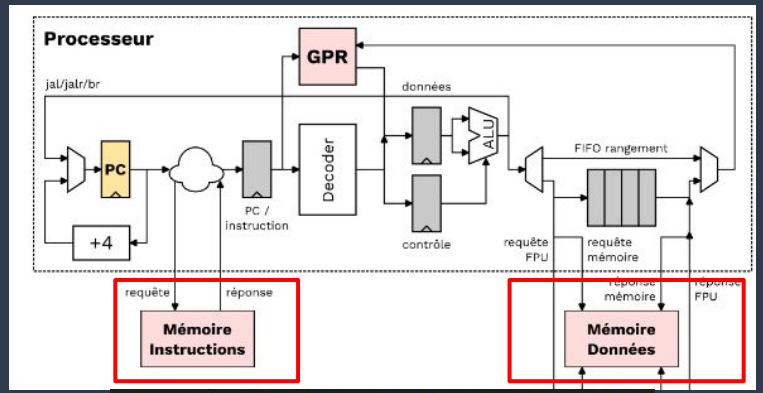


Blocs décrits et fonctionnels

- Compteur d'adresse
- Gestion des signaux de contrôle
- Saut d'adresse

```
class fetch extends Module {
    val io = IO(new Bundle{
        // Adresse du saut
        val i_jumpAdr = Input(UInt(32.W))

        // Activation du saut
        val i_jumpEnable = Input(Bool())

        // Adresse de sortie du PC
        val o_instrAdr = Output(UInt(32.W))
    })


    // ADD (+4)
    val instrAdr = io.o_instrAdr
    val addOut = instrAdr + 4.U


    // PC

    val PC_in = RegInit(0.U(32.W))

    // Si jumpEnable = true.b alors saut à jumpAdr
    // Sinon currentAdr = currentAdr + 4
    val currentAdr = Mux(io.i_jumpEnable, io.i_jumpAdr, addOut)

    PC_in := currentAdr
    instrAdr := PC_in
}
```

IV/ Mémoires




```
class InitMemInline(memoryFile: String) extends Module {
  val width: Int = 32
  val io = IO(new Bundle {
    val i_rEnable = Input(Bool())
    val i_wEnable = Input(Bool())
    val i_Adr = Input(UInt(10.W))
    val i_data = Input(UInt(width.W))
    val o_data = Output(UInt(width.W))
  })

  val memoire = SyncReadMem(1024, UInt(width.W))
  // Initialize memory
  if (memoryFile.trim().nonEmpty) {
    loadMemoryFromFileInline(memoire, memoryFile)
  }



  //Lecture
  when(io.i_rEnable){
        io.o_data := memoire.read(io.i_Adr >> 2.U)
    }.otherwise{io.o_data := DontCare}

    //Ecriture
    when(io.i_wEnable){
        memoire.write(io.i_Adr >> 2.U, io.i_data)
    }
}
```

Lecture et écriture dans un fichier mémoire

Entrées :

- Signaux d'activation de lecture/écriture
- Adresse
- Donnée à écrire

Sortie :

- Donnée lue

Mémoire initialisée depuis un fichier texte.

V/ Top level global

```
class top_fetch_exec_mem extends Module {
  val io = IO(new Bundle {
  })

    //FETCH
    val fetch = Module(new fetch)

    //EXEC
    val GPR = Module(new GPR)
    val ALU = Module(new ALU)
    val Decodeur = Module(new Decodeur)

    //MEMOIRES
    val i_mem = Module(new InitMemInline("doc_memoire/i_mem.txt"))
    val d_mem = Module(new InitMemInline("doc_memoire/d_mem.txt"))

    //CONNECTIONS MODULES

    fetch.io.i_jumpAdr := DontCare
    fetch.io.i_jumpEnable := DontCare
    i_mem.io.i_Adr := fetch.io.o_instrAdr

    i_mem.io.i_wEnable := DontCare
    i_mem.io.i_rEnable := true.B
    i_mem.io.i_data := DontCare

    d_mem.io.i_wEnable := Decodeur.io.o_wEnable
    d_mem.io.i_rEnable := Decodeur.io.o_rEnable
    d_mem.io.i_Adr := ALU.io.o_rd
    d_mem.io.i_data := GPR.io.o_data_reg2


    Decodeur.io.i_instruct := i_mem.io.o_data

    ALU.io.i_operande := Mux(Decodeur.io.o_sel_operande,GPR.io.o_data_reg2,Decodeur.io.o_imm)
    ALU.io.funct_sel := Decodeur.io.funct_sel
    ALU.io.i_rs1 := GPR.io.o_data_reg1

    GPR.io.i_data := Mux(Decodeur.io.o_rEnable, d_mem.io.o_data,ALU.io.o_rd)
    GPR.io.i_write := Decodeur.io.o_GPRwrite
    GPR.io.i_sel_reg := Decodeur.io.o_rd
    GPR.io.i_read_reg1 := Decodeur.io.o_rs1
    GPR.io.i_read_reg2 := Decodeur.io.o_rs2
}
```

Système fermé :

- Pas de réelles entrée/sortie

Utilisation de fichiers externes pour la mémoire :

- Fichier i_mem pour la mémoire d’instruction
- Fichier d_mem pour la mémoire de données

VI/ Démonstration

VII/ Conclusion

Parties finies :

- **Blocs fonctionnels**
- **Instructions arithmético-logiques**

Axes d’améliorations :

- **Ajout des instructions de jump et branchements**
- **Bypass non concrétisé**