# Implementació del subconjunt d'instruccions C a un processador RISC-V

Maria Tomàs Hidalgo

**Resum–** Aquest document detalla el projecte de final de Grau que consisteix a implementar un subconjunt d'instruccions C per a un processador d'arquitectura *RISC-V* descrit a nivell *RTL*. Es partirà d'un processador *in-order* prèviament dissenyat amb 5 etapes de *pipeline*. El procés de desenvolupament consisteix en un estudi previ del codi font del processador i una esquematització de les etapes ja implementades. A continuació, es realitzen les fases de disseny i implementació del nou subconjunt. Seguidament, s'efectua el procés de verificació i de forma incremental redissenyar i codificar totes les funcionalitats del circuit *RTL*, reparant possibles errors. Un cop finalitzada l'etapa de simulació, es procedeix a l'execució d'un programa senzill que barregi instruccions comprimides i instruccions de 32 bits així com instruccions de salt.

**Paraules clau–** *RISC-V*, Processador, *HDL*, Descripció *RTL*, Instruccions, *Hardware* Obert, Verilog, *Hardware*, Instruccions Comprimides.

**Abstract–** This paper contains a bachelor's degree final project that consists on developing a subset of C instructions for a RISC-V architecture processor described at RTL-level. It will be based on a pre-designed processor in-order with 5 pipeline stages. The development process consists of a previous study of the source code of the processor and a schematization of the implemented stages. Then, the design and implementation phases of the new subset are executed. You can then perform the verification process and redesign and codify incrementally all the features of the code, repairing possible errors. Once the simulation stage is over, a simple program is executed that mixes compressed instructions and 32-bit instructions as well as jump instructions.

**Keywords–** RISC-V, Processor, HDL, RTL Description, Instructions, Open Hardware, Verilog, Hardware, Compressed Instructions.

## 1 Introducció - Context del treball

En arquitectura de computadors, la base de tot sistema és el processador i, per tant, el conjunt d'instruccions que el formen. Un repertori d'instruccions o *ISA* (*Instruction Set Architecture*) és el que defineix les tasques que podrà dur a terme un processador, ja que detalla les instruccions que una *CPU* (*Central Processing Unit*) pot entendre i seguidament executar. Actualment, al mercat hi ha molts tipus de conjunts d'instruccions de diferents cases (INTEL, ARM, etc.), i a més al llarg del temps el mercat ha estat dominat per la producció de màquines de tipus *CISC (Complex Instruction Set Computer)*. Aquests conjunts es caracteritzen per tenir un gran nombre d'instruccions, les quals són més complexes i, per tant, permeten operacions de major dificultat. En canvi, els conjunts de tipus *RISC (Reduced Instruction Set Computer)*, com indica el seu nom, el nombre d'instruccions és més reduït i aquestes són més simples. Això provoca que siguin capaços de fer una única acció per instrucció, contràriament als de major complexitat.

Així doncs, tot i que les arquitectures de conjunts d'instruccions (*ISA - Instruction Set Architecture*) han estat una propietat per motius històrics o empresarials, no n'hi ha una bona raó tècnica per a la manca d'*ISA* obertes i gratuïtes. D'aquesta manera i per a trencar aquest marc limitant d'opcions propietàries, la Universitat de Califòrnia [5] va proposar un estàndard de disseny de processador alternatiu, obert i lliure utilitzant una *ISA* de tipus *RISC*. D'aquesta forma es permet un alt nivell de personalització, iniciant un camí cap al *hardware* obert i lliure que implica el desús progressiu de processadors sobredimensionats, tancats i cars per a apli-

• E-mail de contacte: 1430797@uab.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Raimon Casanova Mhor
• Curs 2021/22

cacions senzilles i especifiques. Aquesta nova arquitectura s'anomena *RISC-V*, la qual està trepitjant fort en un mercat incipient amb altes expectatives.

L'*ISA* que defineix *RISC-V* està dissenyada de manera modular, de forma que consta de diversos subconjunts. Principalment, un subconjunt bàsic, anomenat subconjunt I, i un seguit de subconjunts que doten al processador de més funcionalitats. És evident que la modularitat implica un alt grau de flexibilitat que permet el disseny de processadors específics que incloguin aquelles instruccions necessàries per a dur a terme la seva tasca concreta. En aquest cas, doncs, es pretén afegir un dels mòduls que formen la *ISA RISC-V*, concretament es vol afegir el subconjunt d'instruccions comprimides (de 16 bits, anomenat C) a un processador *RISC-V* prèviament donat. El processador actualment consta del set bàsic. L'extensió C es pot afegir a qualsevol de les bases *ISA (RV32, RV64, RV128)* i utilitza el terme genèric *RVC*. Normalment, el 50%-60% de les instruccions *RISC-V* d'un programa es poden substituir per instruccions *RVC*, el que resulta en una reducció de la mida del codi del 25% al 30%.

## 2 Motivació i Objectius

El principal objectiu del projecte és implementar el subconjunt d'Instruccions C a un processador *RISC-V* de 32 bits descrit a nivell *RTL (RegisterTransfer Level)* en llenguatge Verilog. Finalment, s'executarà un petit programa per comprovar el seu correcte funcionament. Si desglossem el projecte en subobjectius, resulta de la següent manera:

- Estudi inicial, mitjançant una recerca d'informació sobre *RISC-V* i l'estàndard de l'*ISA* a implementar.
- Estudi del codi font, del processador descrit a nivell *RTL* en llenguatge Verilog.
- Creació del Diagrama de Blocs del *Core* del qual es parteix.
- Dissenyar i codificar el *core* per al nou subconjunt d'instruccions comprimides.
- Dissenyar i codificar l'entorn de test mitjançant tasques per a cada instrucció implementada.
- Simular i verificar el funcionament del subsistema i dels anteriors.
- Modificar el codi segons els resultats de test.
- Polir el codi final, a nivell d'estil i comentaris.
- Córrer un programa senzill amb diverses instruccions d'ambdós tipus.

## 3 Metodologia, planificació i procés de desenvolupa-ment

Pel que fa a l'organització del treball, s'han dividit els objectius en tasques genèriques, les quals es duran a terme de forma esglaonada. Concretament de les tasques de codificació i verificació sorgiran diversos reptes als quals donar solució. Les solucions proposades a aquests reptes s'implementaran de forma incremental, de forma que per cada funcionalitat es realitzarà una fase de disseny, codificació i verificació. Aquest fet implica simultaneïtat entre les tasques de codificació i verificació. El diagrama Gantt de la **figura 1** mostra la projecció ideal del projecte.

## 4 Context i estat de l'art

Actualment, existeixen nombrosos dissenys de processadors basats en arquitectura RISC-V. Aquests dissenys combinen diversos sets d'instruccions definits en l'*ISA*. Avui dia, podem trobar un llistat al repositori *RISC-V International - Archive [6]*. En aquest llistat, entra d'altres, es troben cores implementats amb instruccions comprimides basats en *RV32I* i també es troben processadors que apliquen estratègies de *pipeline*. En aquest projecte es parteix d'un processador *RISCV* d'un treball de fi de màster [1]. Aquest *core*, no té implementada la lògica necessària per a l'ús d'instruccions comprimides, tasca central del projecte, que es basarà en l'estàndard de *RISCV* actual [5].

### 4.1 Processador RISC-V

El primer pas a realitzar, ha estat un estudi en profunditat del processador *RISC-V* del que es parteix. És un processador *RISC V in order* amb 5 etapes de *pipeline*, una interfície *AXI* i memòria d'instruccions i de dades (el banc de registres té 32 posicions de memòria de 32 bits cada posició). En concret, es treballen les següents etapes:

- ***Instruction Fetch (IF)***: Aquesta etapa calcula l'adreça de la següent instrucció a executar. Aquesta etapa conté *Program Counter (PC)*, el registre utilitzat com a punter a la memòria d'instruccions. Si es produeix un *branch* l'adreça de memòria calculada al *PC*, no s'ha d'executar.
- ***Instruction Decode (ID)***: inclou una unitat de control (*CU*) que rep la instrucció de l'etapa anterior i la descodifica en una sèrie de senyals de control o operands que s'utilitzaran en la següent etapa. També conté el *registerfile* i una unitat de salt que s'encarrega de comprovar la condició de salt i calcular l'adreça de salt.
- **Execució (EXE)**: calcula operacions aritmètiques entre dos operands obtinguts del registre i/o immediats mitjançant una Unitat Aritmètica-Lògica (*ALU*).
- **Accés a la memòria de dades (MEM)**: se centra en les operacions de *load/store* a la memòria de dades.
- ***Write-back (WB)***: les operacions del tipus *load* triguen un cicle de rellotge a lliurar els operands (o més per llegir/escriure dades d'una memòria SRAM connectada al bus AXI), aquesta etapa existeix per coincidir amb el temps. La sortida d'aquesta etapa torna al *registerfile* per a registrar el contingut carregat de la memòria de dades.

Per a poder modificar aquestes etapes i incloure un conjunt d'instruccions comprimit s'ha realitzat un disseny que ha ajudat a detectar els punts conflictius del problema. La **figura 12**, que es troba a l'**Apèndix A.2** es poden veure les etapes en ordre amb les sortides de cadascuna d'elles, registrades.

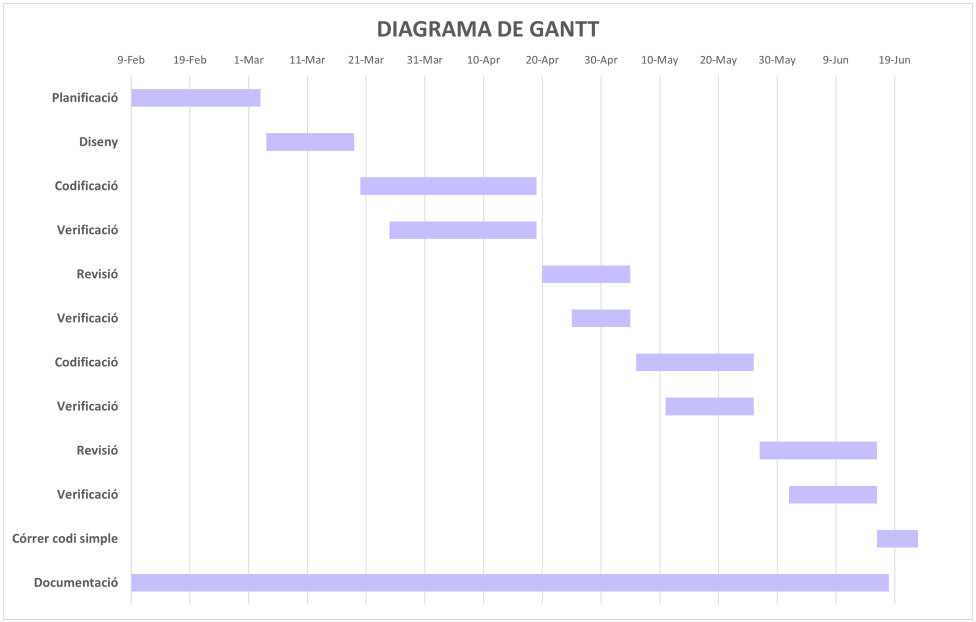


Fig. 1: Diagrama de Gantt per a la planificació.

## 4.2 Subconjunt d'instruccions Comprimides

L'estàndard de *RISC-V* pot ser de 32, 64 o 128 bits. En aquest projecte el processador és de 32 bits. Així i tot, l'estàndard, ofereix un set d'instruccions de 16 bits el qual és compatible amb totes les altres extensions d'instruccions. L'extensió C permet que les instruccions de 16 bits es barregin lliurement amb instruccions de 32 bits. D'aquest forma, les instruccions de 32 bits poden començar en qualsevol límit de 16 bits, és a dir, *IALIGN*=16. Aquest subconjunt, utilitza un esquema de compressió senzill que ofereix versions més curtes de 16 bits de les instruccions de 32 bits. S'acostumen a implementar quan:

- El desplaçament immediat o d'adreça és petit.
- Un dels registres és el registre zero (x0).
- El registre de destinació i el primer registre d'origen són idèntics.
- Els registres utilitzats són els 8 més populars.

En resum, és un subconjunt pensat per actuar com a extensió d'un subconjunt base. En el nostre cas, el subconjunt base és el set *RV32I*.

A la **figura 2** es poden veure els diferents formats en funció de la tasca que realitza cada tipus d'instrucció. D'aquest format, cal destacar que els bits que formen l'*operation code (op)*, són els que ens serveixen per identificar que les instruccions són comprimides però no per identificar el tipus d'instrucció comprimida. Per saber el tipus d'instrucció comprimida caldrà tenir en compte l'*operation code (op)* i els tres bits més significatius de la instrucció (en alguns casos també els bits que formen el codi funct4, funct6 i funct2).

# 5 Implementació del subconjunt

Segons l'estudi, realitzat, el primer pas serà adaptar cadascuna de les etapes per al tractament d'instruccions comprimides. Així doncs, caldrà modificar l'etapa d'*Instruction*

| Format | Meaning | 15 14 13 | 12 | 11 10 | 9 8 7 | 6 5 | 4 3 2 | 1 0 |
|---|---|---|---|---|---|---|---|---|
| CR | Register | funct4 | | rd/rs1 | | rs2 | | op |
| CI | Immediate | funct3 | imm | rd/rs1 | | imm | | op |
| CSS | Stack-relative Store | funct3 | imm | | | rs2 | | op |
| CIW | Wide Immediate | funct3 | imm | | | | rd′ | op |
| CL | Load | funct3 | imm | | rs1′ | imm | rd′ | op |
| CS | Store | funct3 | imm | | rs1′ | imm | rs2′ | op |
| CA | Arithmetic | funct6 | | | rd′/rs1′ | funct2 | rs2′ | op |
| CB | Branch | funct3 | offset | | rs1′ | offset | | op |
| CJ | Jump | funct3 | jump target | | | | | op |

Fig. 2: Formats de les instruccions comprimides en funció del tipus d'acció que realitzen.

*Fetch* per a obtenir les instruccions d'una memòria que pot estar desalineada, l'etapa d'*Instruction Decode* per a descodificar instruccions de 16 bits i per últim, tractar les instruccions de salt que no es tindran en compte en les primeres modificacions de les etapes d'*IF* i *ID*. Aquestes seran els reptes que s'hauran de superar de forma incremental.

## 5.1 Adaptació de l'etapa *Instruction Fetch*

La primera etapa del cicle d'execució és l'etapa d'*Instruction Fetch*. Partint de la descripció en Verilog donada s'ha realitzat enginyeria inversa per a obtenir el diagrama de blocs que defineix el circuit *RTL*, el qual implica l'obtenció de la instrucció de la memòria i l'actualització del *Program Counter (PC)*. A la **figura 3** es mostra el diagrama d'aquesta etapa.

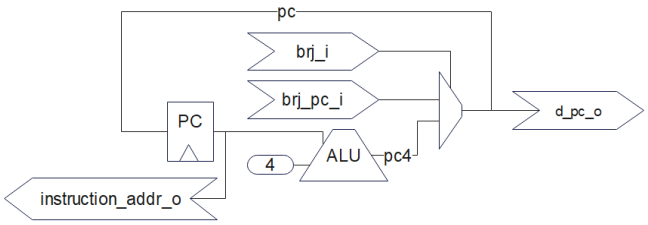


Fig. 3: Diagrama de blocs que determina el valor del PC i actualitza la instrucció llegida.

En introduir en memòria instruccions de 16 bits, ens tro-

bem amb la possibilitat que la memòria es desalinei. Com l'accés a memòria és de 32 bits, si s'introdueixen instruccions comprimides, en un sol accés a memòria ens podem trobar amb diversos casos on, per exemple, llegim una instrucció, dues o parts d'instruccions de 32 bits. Per tant, en funció del que s'estigui llegint, l'actualització del PC variarà i, en conseqüència, ja no es pot passar a la següent etapa els bits llegits sense que s'avaluïn prèviament.

A la **figura 4** es mostra una memòria que barreja instruccions de 32 i 16 bits. Per a desglossar els diferents estats en què es pot trobar la memòria a l'hora de fer una lectura de 32 bits, s'han avaluat les diferents situacions i s'han agrupat segons si la memòria que es llegeix està alineada o no, i si després de la lectura actual ho continua estant. Abans, s'ha de tenir en compte que, per a aquesta primera etapa, quan es parla de memòria desalineada, cal mencionar que és l'estat de la memòria. En cap cas es fa referència a l'accés, ja que l'accés sempre és alineat perquè llegim de 32 en 32 bits, i el PC, també estarà alineat (serà un múltiple de 4).

- **Memòria alineada:** Els primers casos que ens podem trobar és que partim d'una memòria alineada. Aquest cas passarà quan no s'hagin llegit un nombre imparell d'instruccions comprimides. Amb una lectura que garanteix partir d'una memòria alineada, es pot desalinear la memòria o no. I per tant, s'ha d'avaluar si el que es llegeix és una instrucció o dues. Així doncs, els diversos casos són els següents:
  - **Una instrucció de 32 bits:** En aquest cas es procedeix com fins ara, augmentant el *PC*. En aquest cas ens mantindrem el mateix estat de memòria alineada.
  - **2 instruccions comprimides:** en aquest cas, es voldrà guardar la segona instrucció i no augmentar el PC. Es passarà a un estat, on la memòria continuarà alineada tot i que caldrà tractar al segona instrucció sense fer un segon accés a memòria.
  - **Una instrucció comprimida i els 16 primers bits d'una instrucció de 32:** en aquest cas, es voldrà guardar la part de la instrucció de 32 bits per posteriorment concatenar-la amb els següents bits i sí que es voldrà augmentar el *PC*, per a poder llegir els bits restants. Aquest cas implica que la memòria es desalinei fins a llegir una segona instrucció de 16 bits. En aquell instant la memòria estarà alineada de nou, ja que el nombre d'instruccions comprimides llegides serà parell.
- **Memòria desalineada:** En aquest punt s'ha llegit un nombre imparell d'instruccions comprimides, per tant, en una única entrada a memòria podem tenir parts d'instruccions en lloc d'una instrucció sencera.
  - **Els 16 últims bits d'una instrucció de 32 y una instrucció comprimida:** En aquest cas, cal concatenar la instrucció de 32 bits amb el que s'ha guardat de la lectura anterior i caldrà guardar els 16 bits de la instrucció comprimida i, per tant, no s'ha d'augmentar el *PC*. Això implica, que es passarà a un estat, on la memòria tornarà a estar alineada tot i que caldrà tractar al segona instrucció sense fer un segon accés a memòria.
  - **Els 16 últims bits d'una instrucció de 32 i els 16 primers bits d'una altra instrucció de 32:** en aquest cas, de nou cal concatenar el prèviament guardat amb els primers bits d'aquesta lectura i caldrà guardar els 16 primers bits de la instrucció de 32, per a la pròxima lectura. Per tant, seguirem en un estat de memòria desalineada.
- **Memòria desalineada que s'alinea amb els bits guardats:** En el cas d'haver llegit una instrucció de 16 bits, la qual ha estat guardada, implica que aquesta instrucció es trobava en els bits més significatius d'una lectura de memòria. Per tant, implica que ens trobàvem en un punt on la memòria s'ha desalineat, però en la lectura anterior ja s'ha obtingut la instrucció que torna a alinear la memòria. En aquest cas, s'envien a la següent etapa els bits que s'han guardat a un registre auxiliar i s'augmenta el PC, per a seguir els accessos a memòria sabent que aquesta ja està alineada de nou.

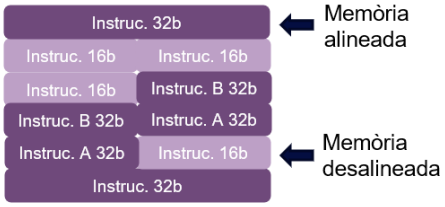


Fig. 4: Esquema amb els diversos casos d'estat de la memòria.

Comparativament amb el disseny inicial, aquest canvi implica afegir un segon multiplexor, l'entrada del qual és la sortida del multiplexor de la **figura 3** i la segona entrada és el *PC* original directe del registre, sense sumar-li 4 bytes. D'aquesta manera, es manté el *PC* un cicle de rellotge més. Aquest multiplexor afegit s'ha controlat amb la nova màquina d'estats de la **figura 5**. Aquesta màquina inclou un estat *IDLE* com a punt de partida i tres estats que solucionen les diverses situacions que es poden donar en cadascun d'ells, a partir de les quals es decideix el següent estat.

## 5.2 Adaptació de l'etapa *Instruction Decode*

Tal com es mostra al diagrama general de la **figura 12** aquesta etapa, consta de diversos mòduls. Si no tenim en compte instruccions de salt, per a avaluar instruccions comprimides, només cal modificar la Unitat de Control, la qual desglossa els 32 bits que li arriben de l'etapa anterior en diferents senyals i registres. Primer s'ha fet una anàlisi en profunditat del circuit actual per a adaptar-lo, evitant variar al màxim l'original i així mantenir les funcionalitats ja implementades, intactes.

La Unitat de Control és simplement un descodificador que a partir de la instrucció genera els senyals de control de les diferents etapes. En el cas de les instruccions de 32 bits, s'obté un *opcode* que correspon als 7 bits menys significatius de la instrucció. Per cada *opcode* s'assignen uns

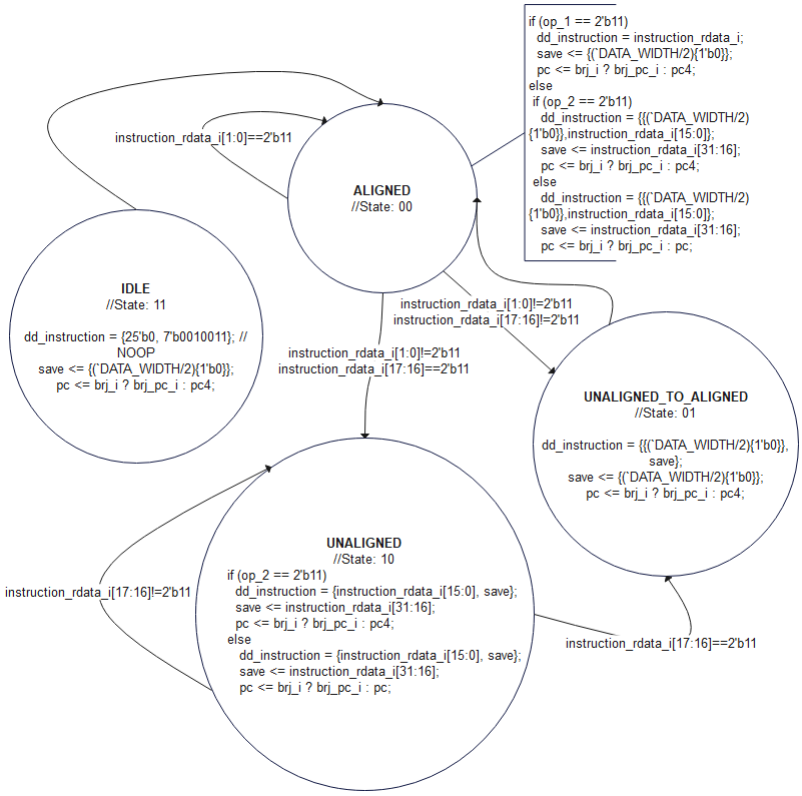


Fig. 5: Màquina d'estats que determina el nou estat del PC i actualitza la instrucció llegida adaptada a instruccions comprimides i a memòria des-alineada.

senyals de sortida, els quals determinaran les accions a realitzar a les següents etapes. A la pràctica, el que ens trobem en aquesta etapa és una taula de veritat amb *opcodes* i els senyals de sortida corresponents, que equival a un circuit combinacional descrit amb una gran estructura *case/if-else* en lleguatge *Verilog*.

Així doncs, a partir d'aquesta estructura, s'ha ampliat la taula de veritat amb un nou *opcode* (7'b0000000) per a instruccions comprimides. La descripció del circuit combinacional, ha consistit a desplegar una sèrie d'*if-elses*, per a aquest nou tipus d'instruccions, que defineixen els senyals de sortida. Al fragment de **codi 1** es pot veure l'ampliació per descodificar una instrucció *AND* comprimida.

```
assign opcode = instruction [1:0]
    == 2'b11 ? instruction[6:0]
    : {7'b0000000};
always@(*)
begin
//{...}
case(opcode)
//{...}
7'b0000000:
begin
  first_op_code <= instruction[15:13];
  case(first_op_code)
   3'b100:
   begin
   if (instruction[12:10] == {3'b011} )
    begin
     second_op_code = instruction[6:5];
     case (second_op_code)
     2'b11:
     begin
      reg_d <= {3'b00,instruction[9:7]};
      rs_2 <= {3'b000,instruction[4:2]};
      ALU_op <= `ALU_OP_AND;
      regfile_wr =
        regfile_waddr!={`REG_ADDR_WIDTH{1'b0}};
      i_r2_o = 1'b1;
      i_r1_o = 1'b1;
     end
     endcase
   end
   end
  endcase
end
//{...}
endcase
end
```

Codi 1: Exemple d'adaptació de descodificació per a la instrucció comprimida AND (arxiu font: *core_control_unit.v*).

## 5.3 Adaptació d'instruccions comprimides de tipus Salt

Aquesta adaptació s'ha tractat en dues fases. El primer subobjectiu ha estat modificar la unitat de control per tal de descodificar les instruccions de salt. El segon subojectiu serà afegir les funcions de test per verificar la correcte implementació d'aquestes. Com s'ha explicat anteriorment, el mètode de descodificació d'instruccions comprimides, serà el mateix per a totes les instruccions d'aquest tipus. Per tant, l'estructura *case* que s'ha vist al **codi 1** s'amplia també amb la descodificació de les instruccions de salt. Hi ha dos tipus d'instrucció de salt, les que efectuen el salt directament o les que depenen d'una condició. La unitat que avalua la condició i que calcula l'adreça de salt és la *Branch Unit*.

Les instruccions de *branch* actuen sobre la unitat de *branch* on s'actualitzaran els senyals de control que indiquen si s'efectua el salt o no, depenent de si es compleix la condició de salt. Així, doncs, a l'estructura *case* de la

Unitat de Control, s'activa l'accés als registres que s'han de consultar, es passa l'adreça de memòria del registre que s'ha de comparar (si cal), es guarda el valor de l'adreça de salt en cas que aquest es produeixi, i es guarda el codi de salt. Aquest codi de salt és un codi intern que indicarà a la *Branching Unit* el tipus de comparació que cal comprovar per decidir si activar el salt o no.

S'ha ampliat la *Control Unit* per efectuar salts en cas que el registre sigui zero. Seguidament, s'han definit nous codis de salt els quals són *BR_EQZ* i *BR_NEQZ*. Finalment, s'ha codificat, a la *Branching Unit*, un comparador del registre amb el valor zero, el resultat del qual determina si es fa el salt o no, fet que s'indica amb la línia de control *branch_o*. El codi per aquest cas es pot veure a **codi 2**.

```
1  assign regfile_rs1_iEqualtoZero =
2          (regfile_rs1_i == 32'b0);
3   always @*
4    case (BR_op_i)
5      //.../
6     //Compressed instruction
7     `BR_EQZ:  branch_o =
8          (regfile_rs1_iEqualtoZero);
9     `BR_NEQZ:  branch_o =
10         !(regfile_rs1_iEqualtoZero);
11
12   endcase
```

Codi 2: Comparador per a decidir l'acció de salt per a nous codis de salt d'instruccions comprimides (arxiu font: *core_branching_unit.v*).

Un cop preparada la lògica de salt cal comprovar com es fa la lectura de memòria al processador inicial. Fins ara, s'havia estat accedint a una memòria que podia estar desalineada, però el PC sempre havia estat múltiple de 4. El que pot passar en el cas de les instruccions de salt és que el PC no sigui múltiple de 4. Cal comprovar com es realitzen els accessos a memòria per, en cas que el PC estigui desalineat, abans de qualsevol canvi s'ha de saber quins bits de la memòria arriben a l'etapa d'*Instruction Fetch*.

A l'arxiu *sp_ram_instr.sv* està codificat l'accés a la memòria. El que cal destacar d'aquest, és que independentment del valor del PC llegeix els 4 bytes d'aquell registre. Per tant, en el cas que el PC indiqui la meitat d'una entrada en memòria, per exemple un PC amb valor 6, es llegirà des de la posició 4, i els 4 bytes següents. Com si el valor del PC fos 4.

En aquest cas, no cal modificar res de l'accés a memòria, ja que independentment d'un PC desalineat, l'accés continuarà sent alineat. Però si el PC està desalineat, implica que la instrucció a la qual es pretén saltar, no comença fins a 2 bytes després del començament de la lectura. A més, independentment d'on comenci, aquesta podrà ser una instrucció comprimida (de dos bytes) o no. Aquest fet ens obre nous estats a l'etapa d'*Instruction Fetch*. Així doncs, s'ha avaluat quines noves situacions es poden trobar en cas d'un salt.

- **PC alineat**: El cas en què el PC prengui un valor múltiple de 4, partim d'una memòria alineada. En conseqüència, en el cas d'un salt on el PC continuï sent múltiple de 4, s'ha d'anar a l'estat ja configurat per a memòria alineada.

- **PC no alineat**: En aquest cas cal crear un nou estat, ja que els estats que tenim fins ara, fan referència a la memòria desalineada, i en aquest cas no sabem si la lectura dels 2 bytes alinearà la memòria o no. Cal anar a un estat de salt, què anomenem *BRANCHED*, que avalua com està la memòria per saber quin tipus d'instrucció es vol llegir. El que es sap en aquest estat és que els bytes que interessen són els 2 bytes més significatius, els quals poden ser:

  - **Una instrucció comprimida**: En aquest cas, s'envia la instrucció comprimida com s'ha fet fins ara, s'augmenta el PC i es passa a un estat de memòria alineada.
  - **Els 2 bytes menys significatius d'una instrucció de 32 bits**: En aquest cas, cal fer una segona lectura, per a recuperar els bytes que falten de la instrucció. Per fer-ho, es guarden els bytes i s'augmenta el PC. Ara bé, a la següent etapa es passa la instrucció *NOOP*, ja que cal un cicle més. En aquest punt se sap que la memòria està desalineada i que s'està llegint una instrucció de 32 bits, en aquest cas ja existeix un estat que tracta aquesta condició, per tant, el següent estat serà el de memòria desalineada.

En conseqüència, s'ha actualitzat la màquina d'estats de l'etapa d'*Instruction Fetch*, la qual resulta tal com es pot veure a la **figura 6**. S'ha introduït un nou estat en el qual, en cas que el PC no estigui alineat, s'avalua en quin dels dos casos anteriors ens trobem. Per altra banda, en cas de salt, si el PC està alineat, independentment de l'estat en què es trobi la màquina, es passarà a l'estat de memòria alineada, tal com s'ha exposat al primer cas. A més, es fa un *stall* per parar el procés i s'augmenta el PC per donar un cicle de rellotge de marge per a la lectura de la instrucció a memòria.

## 6 SIMULACIÓ

Pel que fa a la verificació del sistema, s'han fet diversos tipus de test per a comprovar diverses funcionalitats. Seguint la metodologia incremental, per a cada funcionalitat implementada (és a dir dissenyada i codificada), s'ha realitzat un test per verificar-la i, en cas contrari, per a realitzar modificacions perquè funcioni correctament. Abans, però s'ha hagut d'adaptar l'entorn de verificació. Inicialment l'entorn de verificació consta de diversos arxius (un per cada tipus funcionalitat a testejar) per inicialitzar el processador i a continuació cridar diferents tasques que verifiquen els processos que realitza el *core*. Aquestes tasques estan definides a l'arxiu *testbench.v* on hi ha dos tipus de tasques necessàries:

- **Taques de codificació (*encode*):** Cada una d'aquestes tasques codifica una instrucció, per a carregar-la a memòria i ser executada. Al codi s'identifiquen com *encodeXXX*.

- **Tasques de test:** Aquestes tasques criden les tasques anteriors, en l'ordre especificat per a carregar les instruccions en l'ordre desitjat a memòria i a continuació en comprova el funcionament. En el cas de les instruccions aritmeticològiques, es comprova que el resultat de la operació es guarda al registre de destí i el valor resultant és el correcte. En el cas dels *load* es

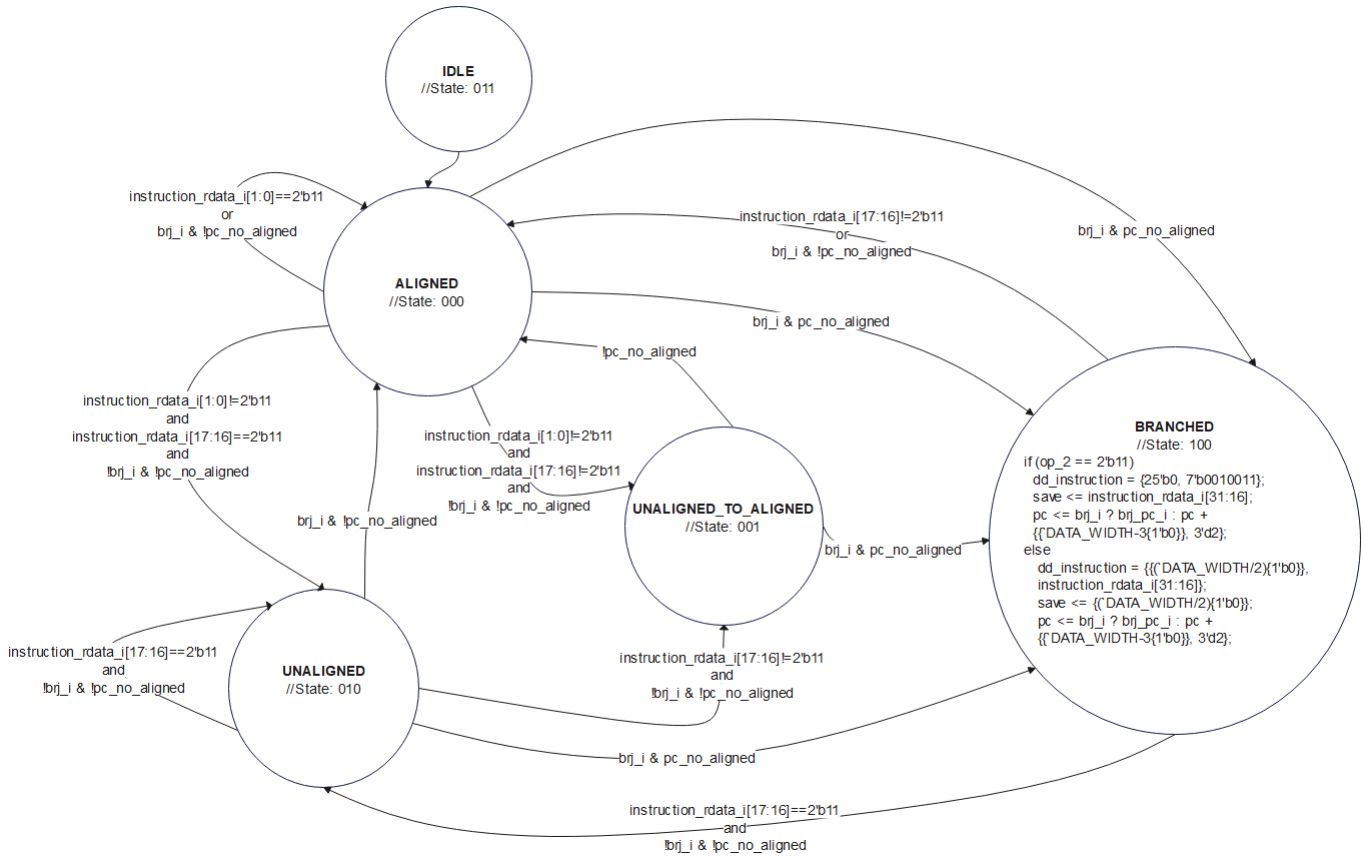


Fig. 6: Màquina d'estats que determina el nou estat del PC i actualitza la instrucció llegida adaptada a instruccions comprimides, a salts desalineats i a memòria desalineada.

comprova que la dada s'ha introduït correctament al registre i en el cas d'instruccions *store* es recupera la dada mitjançant *load* i es comprova el *registerfile*. En el cas d'instruccions de tipus salt es comprova que el *PC* prengui el valor de l'adreça de salt.

El primer pas ha estat codificar algunes tasques de tipus *encode*, per a poder carregar a memòria instruccions comprimides. Respecte a aquest punt s'ha de considerar com introduir les instruccions a la memòria, tenint en compte si aquesta està alineada o no. Per fer-ho s'ha creat un indicador per saber si el *PC* del test, que no és el mateix del *core*, està alineat (és a dir, pren valors múltiples de 4) o no ho està. En cas que no ho estigui, les instruccions comprimides se situen als bits més significatius de l'entrada corresponent a memòria. En cas que sí que estigui alineada se situen als bits menys significatius. I evidentment el *PC* s'augmenta en 2, en lloc d'en 4. Un exemple de tasca *encode*, és la tasca *encodeAndC* que es pot veure al fragment de **codi 3**.

```
task encodeAndC;
input [2:0] rs2;
input [2:0] rd;
begin
 c_instruction = {6'b100011, rd, 2'b11, rs2,
    OPCODE_Q1_COMP};
 if (pc_aux == 1'b1)
 begin
  top_CoreMem_inst.instr_mem.sp_ram_wrap_instr_i.
    sp_ram_instr_i.mem_instr[pc >> 2][2] =
    c_instruction[7:0];
  top_CoreMem_inst.instr_mem.sp_ram_wrap_instr_i.
    sp_ram_instr_i.mem_instr[pc >> 2][3] =
    c_instruction[15:8];
 end
 else
 begin
  top_CoreMem_inst.instr_mem.sp_ram_wrap_instr_i.
    sp_ram_instr_i.mem_instr[pc >> 2][0] =
    c_instruction[7:0];
  top_CoreMem_inst.instr_mem.sp_ram_wrap_instr_i.
    sp_ram_instr_i.mem_instr[pc >> 2][1] =
    c_instruction[15:8];
 end
 pc = pc + 32'd2;
 if (pc_aux == 1'b0) pc_aux = 1'b1;
 else pc_aux = 1'b0;
end
endtask
```

Codi 3: Tasca de codificació de la instrucció comprimida AND (arxiu font: *testbench.sv*).

Així doncs, seguint aquesta tasca com a referència s'han codificat la resta de tasques comprimides. A més s'han hagut de modificar les tasques que codifiquen instruccions de 32 bits, ja que si la memòria està desalineada, caldrà que els bits menys significatius de la instrucció se situïn als bits més significatius de la posició a la qual apunta el PC i l'altra meitat als bits menys significatius de la nova entrada.

Pel que fa a les tasques de test, aquestes s'han anat creant en funció de les proves que s'han volgut realitzar. De forma generalitzada, s'ha creat una tasca de test per a cada instrucció nova, de forma que es pot comprovar el funcionament aïllat de cadascuna de les instruccions comprimides.

## 6.1 Simulació d'instruccions per a avaluar l'etapa d'*Instruction Fetch*

El primer pas, doncs, ha estat comprovar que les instruccions a l'etapa d'*Instruction Fetch* un cop llegides de memòria, s'avaluen i s'envien correctament a la següent etapa. Per fer-ho s'han codificat tasques de tipus *AND* comprimides i de tipus *ADD* i *AND* de 32 bits. En la **figura 7**, es pot veure en color cian la lectura de memòria, en el primer cas és 32'h0ff00213, un cicle de rellotge més tard (quan aquesta s'ha avaluat) es tramet a la següent etapa per la sortida anomenada *d_instruction_o* en color fúcsia que, com es pot veure, pren el mateix valor. En aquest cas ha estat així perquè l'estat ens indica que ens trobem en memòria alineada (el registre *state*, amb valor 0, així ho indica) i, per tant, el que s'ha llegit de memòria és una instrucció de 32 bits. Això se sap, ja que els dos bits menys significatius de la instrucció tenen el valor 0x3, que sempre serà així per a totes les instruccions de 32 bits. Si es continua comprovant la **figura 7** es pot veure que, en el següent cicle de rellotge de l'arribada de la lectura 32'h01938ef1, no s'envia aquest mateix valor a la següent etapa, sinó que s'envia 32'h00008ef1. Això és per què es tracta d'una instrucció comprimida, ja que el valor dels dos bits menys significatius de la instrucció són 0x1. Els dos bytes més significatius es guarden al registre *save* i també s'avaluen per saber si corresponen a una instrucció de 16 bits o als bits menys significatius d'una instrucció de 32 bits. En aquest cas com correspon a una de 32 (ens ho indica *op_2* = 0x3), ens mantindrem en un estat de memòria desalineada, l'estat ens l'indica el registre *state*, el qual passa a valer 0x2.

## 6.2 Simulació d'una instrucció comprimida completa per a avaluar l'etapa d'*Instruction Decode*

L'objectiu d'aquesta simulació és comprovar el funcionament de cadascuna de les instruccions. Per tant, en aquest cas s'han executat cadascun dels tests que correspon a cadascuna de les noves instruccions comprimides implementades. Per exemple, s'ha executat el test que permet comprovar el correcte funcionament d'una instrucció *AND*, el codi corresponent es pot veure al fragment de **codi 4**.

```
task test_andC;
begin
$display("COMPRESSED␣AND␣Test");
pc = 32'b0;
pc_aux = 1'b0;

rstinstrMem();
encodeAddi(5'h0, 5'h5, 12'hFFF);
encodeAddi(5'h0, 5'h4, 12'hFF);
encodeAndC(3'h4, 3'h5);
rst_n = 1'b1;
waitNclockCycles(8);
if (top_CoreMem_inst.core_inst.
    id_stage_inst.reg_file_inst.regFile[5]
    == 32'h000000FF)
    $display("OK:␣reg5␣is␣:␣%h␣\n",
    top_CoreMem_inst.core_inst.id_stage_inst.
    reg_file_inst.regFile[5]);
else begin
  $display("ERROR:␣reg5␣has␣to␣be
␣␣h000000FF␣but␣is:␣%h␣\n", top_CoreMem_inst.
  core_inst.id_stage_inst.reg_file_inst.
  regFile[5]);
  //$fatal;
end
end
endtask
```

Codi 4: Test de la instrucció comprimida AND (arxiu font: *testbench.sv*).

El resultat d'aquest test és positiu, tot i que per a major veracitat s'ha repassat el correcte funcionament mitjançant el *Waveform*. A la **figura 8** es poden veure els senyals implicats més significatius. Sense entrar en detalls, es pot veure marcat en color groc com les instruccions passen de l'etapa d'*Instruction Fetch* a l'etapa de Descodificació. En aquesta fase, podem veure com els senyals de control, marcats en blau, prenen valors en funció del tipus d'instrucció. El cas de la instrucció comprimida es correspon al codi de la *ALU* 7, ja que la instrucció és 00008ef1. En color rosa, Es veu com s'assignen les adreces dels registres involucrats. En aquest cas l'operació és x[5] = x[5] + x[4] (x[5]=ffffffff i x[4]=000000ff), el resultat de la qual es veu al regfile_data_i ressaltat en color fúcsia.

## 6.3 Simulació d'una instrucció de salt per avaluar la correcta actualització del *PC*

En aquest cas, s'han creat les tasques *encode* pertinents a les instruccions de salt. Per exemple, s'ha avaluat la instrucció *Branch not Equal Zero* i *Branch Equal Zero*. Aquest tipus de tasca no té complicació i ja s'ha exposat anteriorment. A continuació s'ha codificat la tasca de test. En aquest cas, no es comprova l'estat d'un registre qualsevol sinó l'estat del *PC*, ja que en el cas que realitzi el salt s'ha d'actualitzar a una nova entrada de memòria.

A la **figura 9** es pot veure la simulació de la tasca de salt en el cas que el registre sigui igual a 0. En aquest cas el *PC* s'actualitza al valor que indica la instrucció. No es manté i no continua, ja que no hi ha cap instrucció a llegir en aquest cas. Quan s'està tractant la instrucció comprimida de salt, a la unitat de control s'activa a 7 el codi de salt, perquè és el codi que s'ha definit a l'arxiu *defines.vh* com a codi d'aquesta operació. Podem veure en color taronja que arriba aquest codi a la unitat de salt pel bus *BR_op_i*. Aquest codi activarà la sortida *branch_o* en color cian, ja que el comparador del registre 1 amb el valor 0 està actiu (en color groc). Finalment, al següent cicle de rellotge on es procediria a llegir de memòria, la instrucció que indica el salt, el *PC* ha de valer 252. Efectivament, es pot veure en color fúcsia que el *PC* pren el valor en hexadecimal 0xfc, que en decimal és 252.

## 6.4 Simulació d'un petit programa per avaluar el correcte funcionament

S'ha elaborat un restador en codi màquina amb 5 instruccions, tres de les quals són comprimides. Aquest restador inclou una instrucció de salt, que en cas que la condició es compleixi salta a una posició de memòria que no és múltiple de 4. Per tant, és un cas aleatori que comprova els diversos problemes que s'han enfrontat en aquest projecte (memòria desalineada, instruccions comprimides barrejades amb d'altres no comprimides, noves instruccions que codificar, instruccions comprimides de salts i PC desalineat). El codi el conformen en aquest ordre les instruccions de la **figura 1**.

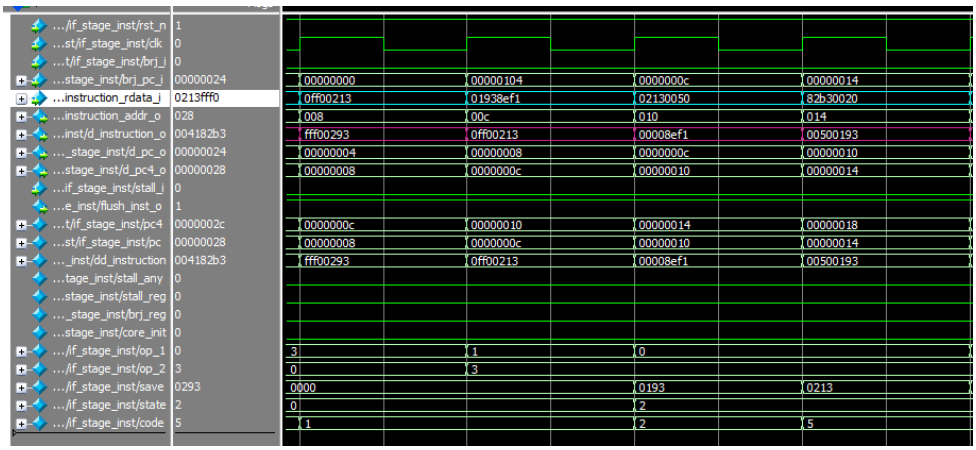

Fig. 7: Resultat de la simulació de l'execució de tasques de 32 i 16 bits barrejades.

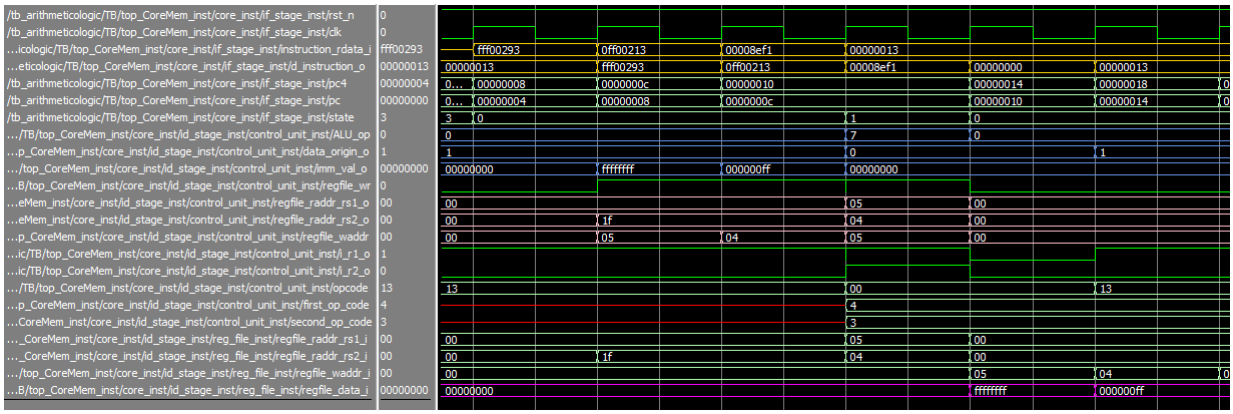

Fig. 8: Resultat de la simulació de l'execució de la instrucció AND comprimida.

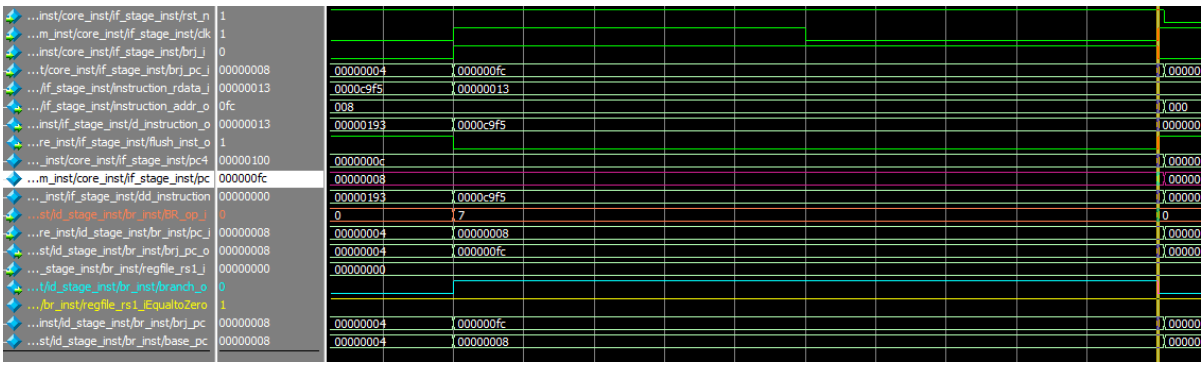

Fig. 9: Resultat de la simulació de l'execució de la instrucció comprimida de salt en cas que el registre sigui 0, a la posició 252 de memòria.

| Instrucció | Nom |
|---|---|
| 00400293 | ADDI |
| 0185 | C.ADDI |
| 8e8d | C.SUB |
| feed | C.BNE |
| 00a00293 | ADDI |

TAULA 1: Codi executat al processador RISC-V final a mode de verificació.

A la **figura 10** es pot veure l'execució sencera d'aquest programa. És interessant destacar el salt, en el moment de descodificar la instrucció de salt, brj_pc_i (en color fúcsia) pren el valor de 6, i és que en cas que es compleixi la condició de salt, s'ha de tornar a aquell punt per a executar la instrucció *C.SUB*. Com és el cas (en color rosat es pot veure com canvia el *flag* que indica si es compleix la condició de salt), al següent cicle cal fer un *stall* del processador per donar temps a llegir la instrucció de memòria i la qual es llegeix des de la posició 4 (d'aquesta manera el PC s'alinea) i es tracta el cas a l'etapa d'*IF* a l'estat *Branched* que s'havia definit per aquests casos. La finalitat del restador es compleix, ja que en aquest cas és resta de 4 a 0 tal com es veu al *registerfile* (ressaltat en color blau).

Finalment, cal mencionar que cal automatitzar l'entorn de test per realitzar una verificació més exhaustiva, d'aquesta manera es podrien executar programes amb instruccions

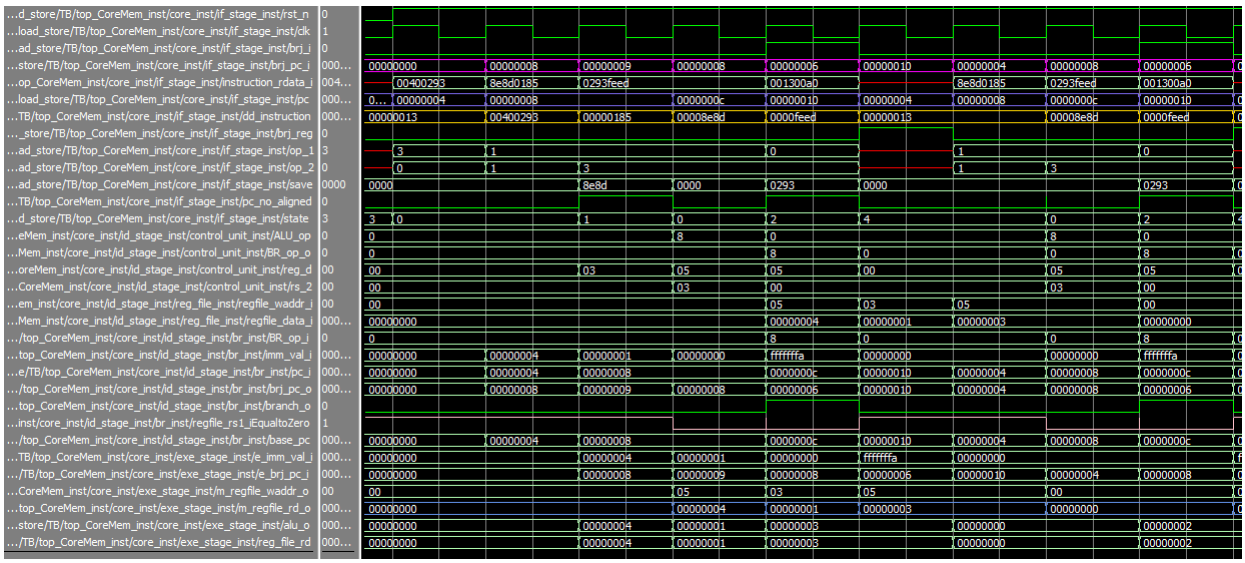

Fig. 10: Resultat de la simulació de l'execució d'un petit programa restador, que combina instruccions de 2 i 4 bytes i conté una instrucció de salt.

generades aleatòriament i treure'n la traça per comparar-la amb la traça generada per un model funcional de processador *RISCV*. Això no ha estat possible perquè actualment no es disposa d'aquest entorn de verificació.

## 7 Conclusions

La realització d'aquest projecte sorgeix de la necessitat completar un processador RISC-V existent perquè pugui executar instruccions comprimides. Durant el procés de desenvolupament del projecte s'han abordat diverses etapes. S'ha realitzat un estudi en profunditat del processador base. A continuació, s'han dut a terme les etapes de disseny, codificació i verificació o test de forma incremental, tractant primer l'etapa d'*Instruction Fetch*, després l'etapa de descodificació i finalment adaptant ambdues anteriors per a fer operacions de salt. La part de test també ha inclòs etapes de disseny i codificació per codificar les instruccions a testejar i les tasques que executen els tests. S'han obtingut bons resultats respecte als objectius plantejats. Actualment, el processador pot executar instruccions comprimides o no indistintament i s'adapta a salts alineats i desalineats. A més, constantment s'han dut a terme els testos base per a garantir que les funcionalitats anteriors es mantenen intactes.

## Agraïments

Em plau agrair aquest projecte al meu tutor Raimon Casanova per acompanyar-me en un dels moments més decisius que he tingut a la meva vida. També vull agrair al Raül Salinas, a la meva mare i als meus companys de carrera, que m'han acompanyat compartint els meus triomfs. El meu futur ha estat molt determinat per aquest projecte, i per tant, és gràcies a totes aquestes persones.

## Referències

[1] F. Fuentes. (2014, Abril ). Codi Font. [Online]. Disponible: https://github.com/FFD-UAB/RISC-V-Instruction-sets-M-and-F

[2] V. Taraate. *Digital Logic Design Using Verilog*. Tsinghua University Press, Xina, 2016.

[3] Y. Li. *Computer principles and Design in Verilog HDL*. Hosei University, Japó, 2015.

[4] D.A. Patterson, J.L. Hennessy. *Computer organization and design*. University of California, Berkley, 2018.

[5] A. Waterman, K. Asanović*The RISC-V Instruction Set Manual Volume I: Unprivileged ISA* SiFive Inc., CS Division, EECS Department, University of California, Berkley, May 2017. [Online]. Disponible: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[6] RISC V International (2021, Marzo ).*RISC-V Cores and SoC Overview* [Online]. Disponible: https://github.com/riscvarchive/riscv-cores-list

[7] D. Patterson, A Waterman. (2018, 11 de Julio).*La Guía Práctica de RISC-V* [Online]. Disponible: http://riscvbook.com/spanish/guia-practica-de-risc-v-1.0.5.pdf

[8] P. Casacuberta. (2020, Febrer).*Disseny d'un "core" RISC-V Didactic* [Online]. Disponible: https://ddd.uab.cat/pub/tfg/2020/tfg_244100/TFG_GEI_Informe_Final.pdf

# APÈNDIX

## A.1 Secció d'Apèndix

Prèviament al disseny de la màquina d'estats de l'etapa d'*Instruction Fetch* definitiva, es va procedir amb una primera versió tant de disseny com de codi,no funcional. El disseny es basava, en mantenir la màquina d'estats original i dins de l'estat REGISTRAR (màquina de la **figura ??** ) afegir una estructura *case*, que tractés els diferents casos. En aquest cas no es funcional, ja que com hem vist, no es poden donar tots els casos sempre; sinó que depenent de l'estat de la memòria es poden derivar unes situacions o d'altres. En segon lloc, el codi en sí no era funcional, ja que en tots els casos valorava els bits 0 i 1 i els bits 15 i 16, i aquesta comprovació no té sentit en el 100% dels casos. Per exemple, en el cas on la memòria estigui desalineada, si s'està llegint dues parts de dues instruccions de 32 bits, els primers 16 bits (els menys significatius) pertanyen als bits més significatius d'una primera instrucció i per tant els bits de la posició 0 i 1 que s'han llegit poden prendre qualsevol valor (ja que estaríem parlant dels bits 15 i 16 d'una instrucció de 32 bits que en la majoria dels casos formen part del codi del registre). A la **figura 11** es pot veure el diagrama de blocs que es va codificar dins l'estat REGISTRAR.

## A.2 Diagrama del processador RISC-V

En el diagrama de la **figura 12** diagrama de blocs es pot veure un esquema del processador base sobre el qual s'ha treballat al llarg del projecte. S'ha realitzat enginyeria inversa sobre el codi, per poder abstraure'n la idea global de la construcció per etapes del *core* del processador.

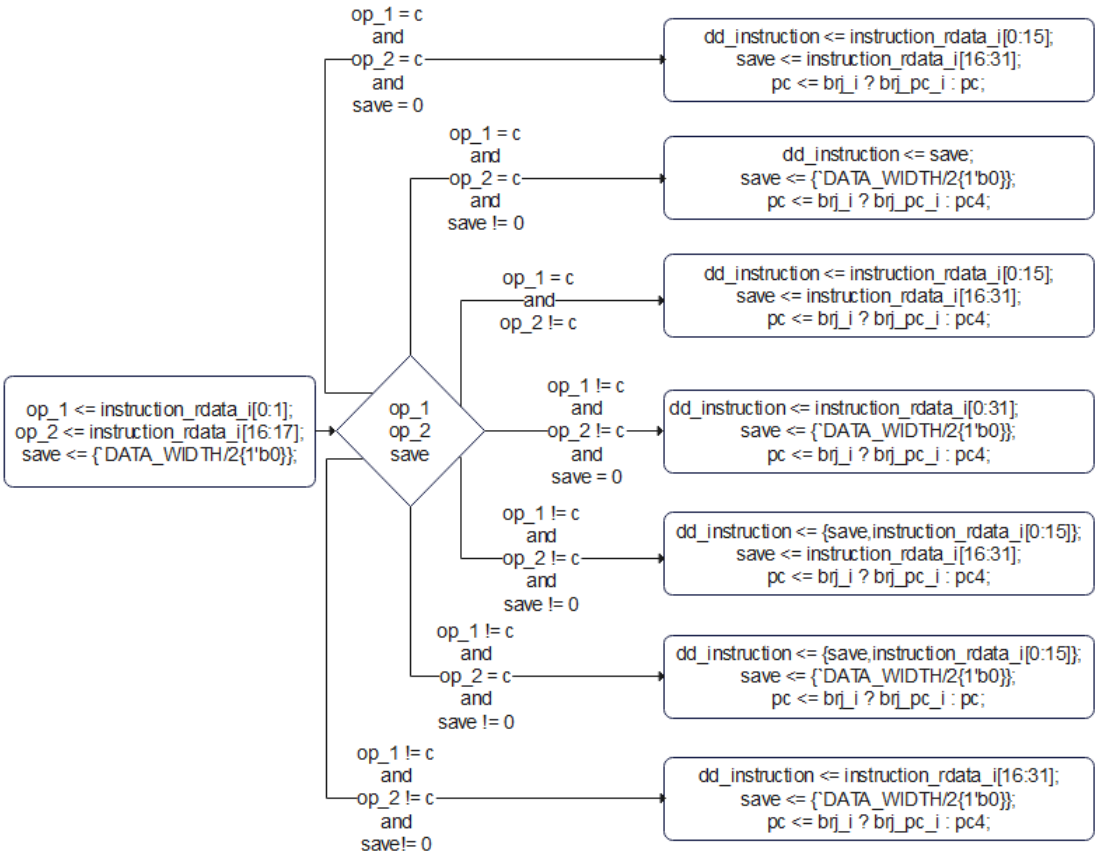


Fig. 11: Diagrama de blocs per al tractament de memòria no alineada.

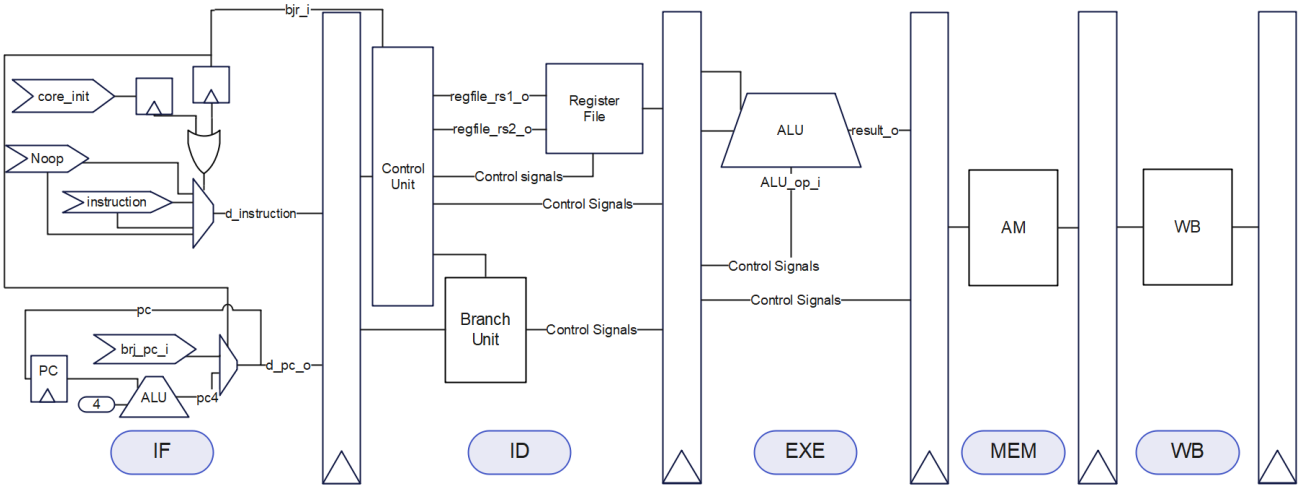


Fig. 12: Diagrama de blocs del processador base, 5 etapes *pipeline in order*.