UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TREBALL FI DE GRAU

Grau en Enginyeria Electrònica Industrial i Automàtica

DISSENY I IMPLEMENTACIÓ D'UN MICROPROCESSADOR RISC

Memòria i Annexos

Autor: Tomàs Hudson Vergara

Director: Jordi Cosp Vilella

Convocatòria: Gener 2020

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

ÍNDEX MEMÒRIA

Resum

Aquest treball de fi de grau es basa en dissenyar un microprocessador que pugui executar el set d'instruccions AVR1, documentar el seu disseny i fer-lo fàcilment modificable per aplicacions específiques futures.

Per tant, s'ha dissenyat i implementat un microprocessador RISC, dissenyant tant el *Datapath* com la Unitat de Control, s'ha convertit el disseny a codi VHDL i s'ha comprovat el funcionament del disseny mitjançant simulacions i implementant-lo en una FPGA Nexys 4 DDR.

Resumen

Este proyecto final de grado se basa en el diseño de un microprocesador que pueda ejecutar el conjunto de instrucciones AVR1, documentando su diseño y haciéndolo fácilmente modificable por futuras aplicaciones específicas.

Por lo tanto, se ha diseñado e implementado un microprocesador RISC, diseñando tanto el *Datapath* como la Unidad de Control, se ha convertido el diseño en código VHDL y se ha demostrado el funcionamiento del diseño a través de simulaciones e implementándolo en un FPGA Nexys 4 DDR.

Abstract

This Bachelor's thesis is based on the design of a microprocessor that can execute the AVR1 instruction set, documenting its design and making it easily modifiable by future specific applications.

Therefore, a RISC microprocessor has been designed and implemented, designing both its *Datapath* and Control Unit. Then, the design has been coded in VHDL and the design has been tested through simulations and implemented on a Nexys 4 DDR FPGA.

Capítol 1. Introducció

En aquest apartat es presenta el projecte que s'ha dut a terme. Es presentaran les motivacions, els objectius que es volen assolir i es farà una breu introducció del treball.

1.1. Motivació

Els microcontroladors estan presents a la major part de dispositius electrònics al món avui, i per tant és molt necessari conèixer com funcionen i com es dissenyen. A més, pels estudiants d'Enginyeria Electrònica, el microcontrolador incorpora tots els elements d'electrònica digital que s'han estudiat en la carrera, i per tant aquest projecte ofereix la possibilitat d'unir tot aquest coneixement en un sol sistema digital.

A més, amb la implementació d'aquest microcontrolador, es podrà treballar amb profunditat amb la FPGA i amb el llenguatge VHDL, eines clau pel desenvolupament de futurs projectes, tant acadèmics com laborals, d'Electrònica Digital.

1.2. Objectius

Aquest treball té com a objectiu principal la implementació en una FPGA d'una microarquitectura que sigui plenament funcional. Aquest microcontrolador no pretén ser competitiu al mercat, ni en velocitat, ni en capacitat computacional, ni en consum, sinó que busca ser una base sobre la qual es pugui construir microcontroladors personalitzats segons les necessitats de projectes futurs. Per tant, es busca dissenyar un sistema sòlid i ben documentat.

1.3. Introducció

En aquest treball es dissenyarà, simularà i implementarà un microprocessador basat en l'arquitectura del microcontrolador ATTiny11, de AVR. Aquest microcontrolador compta amb una sèrie de perifèrics que es comuniquen amb el microprocessador anomenat *Minimal AVR1 Core*, un microprocessador RISC de 8-bits amb arquitectura Harvard modificada que compta amb 90 instruccions de 16 bits.

Com que el microprocessador a dissenyar està basat en el *Minimal AVR1 Core*, el primer pas del treball serà l'estudi de la microarquitectura d'aquest microprocessador. Una vegada s'hagi fet l'estudi del seu funcionament, es procedirà a dissenyar la microarquitectura del microprocessador a realitzar. Es començarà amb el disseny general de la microarquitectura amb un diagrama de blocs, tant dels components com del *datapath*. Una vegada acabat aquest disseny preliminar, es passarà a descriure cada un dels components (registres, multiplexors, descodificadors, memòries, ALU, etc.) mitjançant VHDL (*Very High Speed Integrated Circuit Hardware Description Language*). Després, es descriurà el sistema complet, ajuntant els components del *datapath* i programant també el controlador. Finalment es simularà el sistema per comprovar el seu funcionament correcte i s'implementarà el disseny en una FPGA (*Field Programmable Gate Array*).

S'ha escollit basar el microprocessador en una arquitectura preexistent perquè així s'elimina la necessitat de desenvolupar un compilador específic per la nostra microarquitectura, donat que es podrà fer servir un compilador preexistent, en aquest cas el del *Minimal AVR1 Core* escollit per ser la versió més simplificada i reduïda dels microprocessadors d'AVR i de Microchip.

Capítol 2. Microcontroladors

En aquest apartat es farà una introducció teòrica en el tema dels microcontroladors, la microarquitectura i el seu funcionament. Es farà també un estudi de l'estat de l'art, on es compararan els microcontroladors més utilitzats en l'actualitat i els microcontroladors més similars al model sobre el qual es basa aquest treball.

2.1. Sistemes de computació

Un sistema computacional digital és un sistema que realitza tasques de forma ordenada segons les instruccions que rep. El conjunt d'instruccions que conformen la tasca a realitzar s'anomenen programes. Les instruccions del programa s'executen, una a una, de forma ordenada.

Els programes, escrits en l'idioma de programació que sigui, son finalment traduïts a l'anomenat llenguatge màquina, que es basa en vectors de números binaris que actuen com a senyals de control pel circuit electrònic que executa les instruccions. Com que la programació en llenguatge màquina resulta ser molt lenta i feixuga, s'han anat desenvolupant abstraccions sobre els diferents llenguatges, que després cal desfer per poder guardar correctament les instruccions a la memòria de la computadora. La majoria dels llenguatges de programació més populars ofereixen una programació d'un alt nivell d'abstracció, que permeten elaborar programes més senzills d' escriure, interpretar i modificar. Aquests llenguatges de nivell més alt es tradueixen després a un llenguatge d'abstracció menor, anomenat llenguatge d'assemblador, mitjançant uns programes de traducció anomenat compiladors. El codi d'assemblador, que ja és més complicat d'interpretar pels humans, es tradueix posteriorment a una llenguatge màquina mitjançant els assembladors. Els codis generats per l'assemblador dependran de com estigui dissenyat el computador en el qual es vol executar el programa.

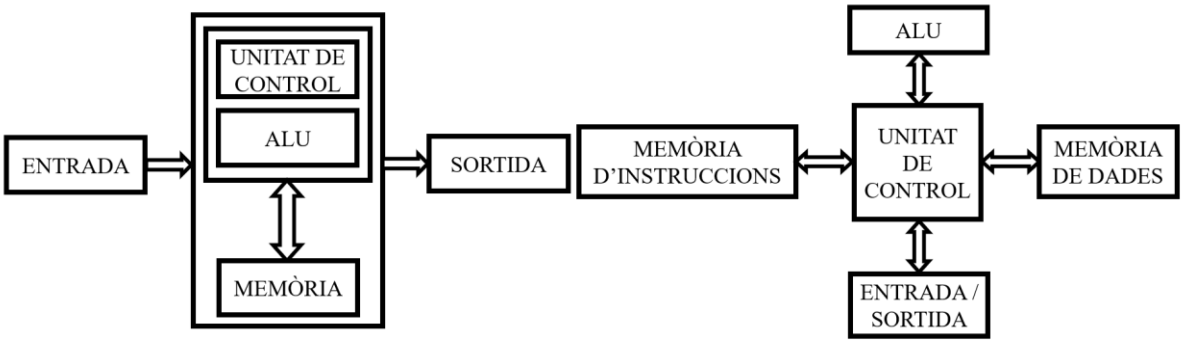


Figura 2.1 Esquema de blocs de l'arquitectura Von Neumann (esquerra) i Harvard (dreta)

L'arquitectura bàsica de les computadores modernes, anomenada Von Neumann, descriu un sistema digital que té blocs separats. En primer lloc, una CPU (Unitat Central de Processament), que està formada per una unitat de control, que entrega els senyals necessaris per l'execució d'instruccions seguint l'ordre marcat pel programa i una unitat aritmètica lògica, encarregada de dur a terme les operacions aritmètiques i lògiques necessàries per executar la instrucció. A part, inclou una memòria per emmagatzemar tant dades com

instruccions i perifèrics d'entrada i sortida, que reben i envien dades. Aquesta arquitectura ha anat evolucionant segons augmentaven les necessitats de rendiment de les computadores, fins a l'arquitectura Harvard, la més usada avui en dia. Aquesta arquitectura busca millorar l'eficiència dels sistemes de computació resolent l'anomenat coll d'ampolla Von Neumann, que és el temps d'execució perdut per utilitzar la mateixa memòria per dades i per instruccions. L'arquitectura Harvard separa les memòries de dades i instruccions per poder accedir a elles de forma paral·lela, així reduint el temps d'execució de cada instrucció.

2.2. Microprocessadors

Els microprocessadors són el els sistemes de computació més integrats. Només contenen una CPU, reben instruccions ja codificades i les executen, enviant les dades resultants a elements perifèrics en un altre *chip*. La CPU executa les instruccions fent una sèrie de passos anomenats *fetch, decode* i *execute*. El *fetch* és el pas on la CPU extreu la instrucció de la memòria d'instruccions. L'adreça on es troba la instrucció desitjada ve marcada pel *Program Counter*, que va variant el seu valor indicant quina és la següent instrucció a executar. El vector instrucció obtingut de la memòria inclou tant el codi de la instrucció a executar, o *opcode*, i els operands, siguin adreces on trobar les dades a manipular o un valor específic. Aquesta interpretació de la instrucció es fa a la fase *Decode.* Una vegada es sap quina instrucció s'ha d'executar, quins son els operands i quins son els passos necessaris per la seva execució, al pas *Execute* es connecten electrònicament diferents elements de la microarquitectura per executar la instrucció. Depenent del disseny de la CPU i de la complexitat de la instrucció, aquest pas es pot realitzar mitjançant una acció o vàries més senzilles.

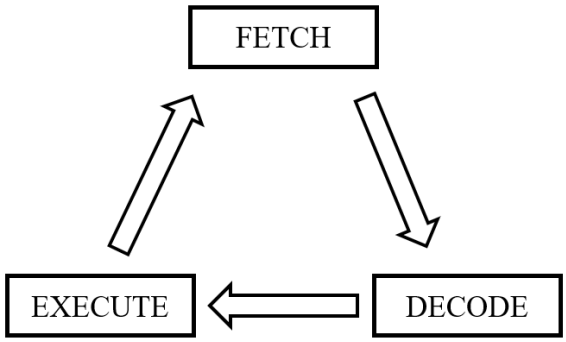


Figura 2.2 Cicle Fetch, Decode, Execute

La CPU, com s'ha explicat prèviament, conté una unitat de control i una unitat aritmètica lògica. La Unitat Aritmètica Lògica (ALU) es un sistema digital combinacional que realitza operacions aritmètiques (sumes, restes, negacions, etc.) i lògiques (AND, OR, EXOR, etc.) amb els operands que es transmeten a les seves entrades. L'operació a realitzar ve marcada per uns codis que formen part de la instrucció (*Opcode)*. Quan la operació requerida ha finalitzat, a la sortida de la ALU es troba el resultat de l'operació i informació sobre l'execució, és a dir, si el número resultant és negatiu, un zero, si el resultat és un vector més llarg del que es mostra a la sortida, etc.

La unitat de control, segons la instrucció a executar, envia senyals als diferents components del sistema per que es realitzi la instrucció desitjada. La diferenciació principal entre CPUs ve donada principalment per la seva microarquitectura, és a dir, el circuit electrònic que permet

l'execució de les instruccions. La microarquitectura, per tant, és dependent del conjunt d'instruccions que es vol poder executar. Aquest conjunt s'anomena Arquitectura de Conjunt d'Instruccions (ISA, per les seves sigles en anglès).

Les ISA es poden segregar segons la complexitat i la durada de l'execució de les instruccions. Per una banda estan les arquitectures CISC (*Complex Instruction Set Computer*), que executen instruccions més complexes, però triguen més d'un cicle de rellotge en executar-los. D'altra banda els computadors amb arquitectura RISC (*Reduced Instruction Set Computer*) executen instruccions més simples però sempre en un cicle de rellotge. Cada un dels sistemes ofereix avantatges i desavantatges. Les màquines RISC, per exemple, com divideixen instruccions complexes en una sèrie d'instruccions senzilles, executades en un cicle de rellotge, permeten un millor control de la durada de l'execució d'instruccions, permetent així el desenvolupament de tècniques avançades de processat com el *Pipeline*, on el processador comença a executar instruccions abans que hagi acabat d'executar l'anterior. Aquesta millora d'eficiència en quant a temps d'execució, però, genera la necessitat d'incloure memòries RAM de mida superior, per poder guardar valors de forma temporal entre instruccions. Les arquitectures CISC, en canvi, suposen un ús més eficient de la memòria RAM, però tenen un conjunt d'instruccions molt més gran i dificulten l'aplicació de la tècnica *Pipeline*.

2.3. Microcontroladors

Un microcontrolador és un sistema electrònic seqüencial que permet l'emmagatzematge, tractament i propagació de dades. La seva estructura és molt variada, però en general es basa en una unitat de processament central (CPU, per les seves sigles en anglès) rodejat d'una sèrie de perifèrics. Aquests perifèrics poden incloure memòries RAM o ROM, ports de comunicació (SPI, USB, etc.) i comptadors, entre molts altres. El nucli és on les dades que entren i surten pels perifèrics son tractades, ja sigui sent sotmeses a operacions aritmètiques, guardades en perifèrics de memòria o enviades als ports de comunicació.

Gran part de l'èxit del microcontrolador ve de que és la integració d'un microprocessador i tots els perifèrics necessaris per la gran majoria de les seves aplicacions en un sol chip. Això permet reduir costos significativament, ja que no s'han de dissenyar circuits específics per cada aplicació. Els microcontroladors, per tant, suposen una forma molt econòmica i integrada de controlar diferents processos de forma purament digital o combinant senyals digitals i analògiques. És per això que s'utilitzen sovint com a principal sistema de control en molts processos automatitzats.

Com s'ha dit prèviament, la majoria de microcontroladors inclouen, a part de la CPU, perifèrics com memòries, comptadors, convertidors analògic-digitals (ADC, per les seves sigles en anglès) i busos de comunicació.

Diferents empreses de microcontroladors utilitzen les mateixes ISA, per poder maximitzar la compatibilitat dels seus components. Per diferenciar-se, per tant, troben diferents formes d'executar aquell mateix conjunt d'instruccions, ja sigui posant més èmfasi en la velocitat d'execució o en el consum de potència del microcontrolador. Una vegada finalitzat el disseny, les empreses solen oferir diversos models basats en la mateixa microarquitectura, microcontroladors amb diverses combinacions de perifèrics per que s'ajustin bé a les

necessitats dels clients. Aquests conjunts de microcontroladors basats en la mateixa microarquitectura s'anomenen "famílies".

2.4. Estat de l'art

Els microprocessadors estan presents en casi qualsevol dispositiu electrònic d'avui en dia, des d'impressores fins a supercomputadors. Això fa que al mercat hi hagi una gran varietat de models del que escollir, des de microprocessadors amb unes velocitats i capacitats computacionals excepcionals fins a microprocessadors bàsics per aplicacions senzilles. El microprocessador desenvolupat en aquest treball estaria en el rang dels microprocessadors senzills, que s'utilitzen per aplicacions pràctiques d'una complexitat relativament baixa, però que ofereixen estabilitat i facilitat d'ús. Per tant, en aquest apartat, s'explicarà l'actualitat tecnològica d'ambdues bandes del ventall d'oferta de microprocessadors.

2.4.1 Intel i9

És l'últim microprocessador CISC presentat per Intel, marca líder en microprocessadors d'alt rendiment, juntament amb AMD. Presentat a finals de 2018, actua com la unitat de processament principal de diversos ordinadors d'alta gama.

El i9 compta amb 8 CPUs que poden estar processant paral·lelament 16 instruccions en *Pipeline* cada un. Això permet que treballi fins a 3.60 GHz de freqüència. Els busos de comunicació tenen la capacitat de transmissió de 8 GT/s, és a dir de transmetre 8 milions de vectors de dades per segon. Admet 128 GB de memòria externa, amb la qual es pot comunicar a 41.6 GB/s.

Es basa en l'arquitectura Intel 64, que com indica el nom consta d'instruccions de 64 bits. Les instruccions estan codificades amb un byte (8 bits) de prefix opcional, un opcode de 1, 2 o 3 bytes de llargada, una adreça de memòria (ModR/M), una adreça de memòria afegida (SIB) per les operacions de desplaçament a la memòria, 1, 2 o 4 bytes pel valor del desplaçament i 1, 2 o 4 bytes per si l'operand es un valor immediat.

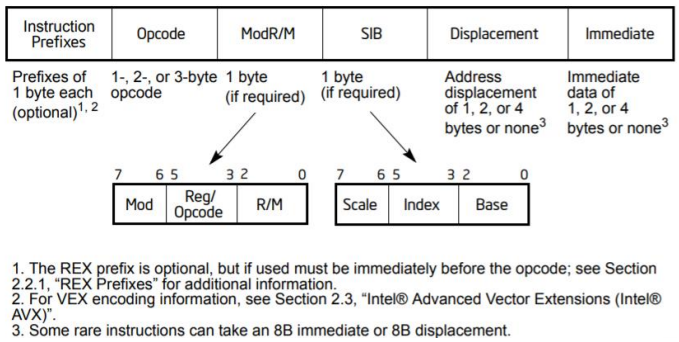


Figura 2.3 Format d'instruccions de l'arquitectura Intel 64 (Font: [6])

2.4.2 ATmega328P

Aquest microcontrolador, presentat al 2013, és el que usa el Arduino Uno, la versió més popular de les plaques de desenvolupament de la coneguda marca italiana Arduino. És un microcontrolador de 8 bits d'alt rendiment i baix consum de la marca Atmel. És de la família de microcontroladors megaAVR.

Té una arquitectura RISC de 131 instruccions, que inclou instruccions de multiplicació. Inclou 32 registres de 8 bits, 32K bytes de memòria programable *flash*, 1K byte de memòria EEPROM, 3 comptadors, canals PWM, un ADC, interfície SPI i comparador analògic.

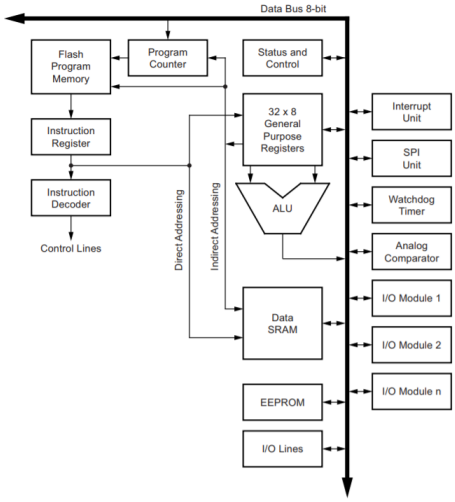


Figura 2.4 Diagrama de blocs del microcontrolador ATMeg328P (Font: [8])

2.4.3 Cortex-M3

El microprocessador Cortex-M3, de l'empresa britànica ARM, s'utilitza en una gran quantitat de microcontroladors. Els processadors Cortex-M3 es basen en l'arquitectura ARMv7-M, que té instruccions de 32 bits i de 16 bits, que poden ser usades de forma conjunta en un mateix programa. La majoria d'instruccions de 16 bits només tenen accés als registres inferiors (R0 a R7), tot i que algunes instruccions de 16 bits també poden accedir als registres superiors (R0-R15).

L'arquitectura ARMv7-M està dissenyada per executar el conjunt d'instruccions Thumb, que poden ser de 16 o 32 bits. La codificació de les instruccions depèn de la seva llargada. Les instruccions de 16 bits, per exemple, estan codificades amb un *Opcode* en els 6 bits de més pes de la instrucció (b15-b10). Aquest opcode diferencia entre operacions aritmètiques i lògiques, transferència de dades, carregar o guardar bits a memòria, fer canvis al PC, carregar o guardar múltiples registres, salts condicionals, salts incondicionals i instruccions miscel·lànies.

Després, un segon *Opcode*, posicionat segons el subgrup d'instruccions marcat per *l'Opcode* anterior, determina la instrucció específica a executar. Els operands venen representats al final de la instrucció, el seu format depenent d'aquesta. Per poder fer operacions amb 3 operands (2 registres d'origen i un de destí), posen automàticament a 0 el bit de més pes de l'adreça del registre, limitant així els operands als registres R0 a R7.

ADD (register)

ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,<Rm>` Outside IT block.

`ADD<c> <Rd>,<Rn>,<Rm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Figura 2.5 Exemple de la codificació d'instruccions del Cortex M3 (Font: [7])

2.4.4 ATTiny 11

El microcontrolador ATTiny11 és part de la subfamília TinyAVR de la marca Atmel. Aquesta subfamília de microcontroladors es caracteritzen per incloure menys perifèrics, menys memòria, menys pins d'entrada i sortida i menys instruccions que les demés famílies. Els microcontroladors ATtiny funcionen amb una varietat de processadors, o *cores*. Alguns models fan servir nuclis més complets, com el ATtiny24, i d'altres usen el AVR1, el *core* més bàsic que ofereix AVR.

L'AVR1 és una arquitectura RISC simple, de 8-bits, amb 90 instruccions i 32 Registres. El ATtiny11 té també 1K byte de memòria *flash*, un comptador de 8 bits, un comparador analògic i modes d'estalvi energètic.

L'arquitectura d'aquest projecte s'ha basat bastant en l'AVR1, tant per la seva simplicitat com per poder aprofitar els compiladors preexistents i no haver de programar en llenguatge d'assemblador.

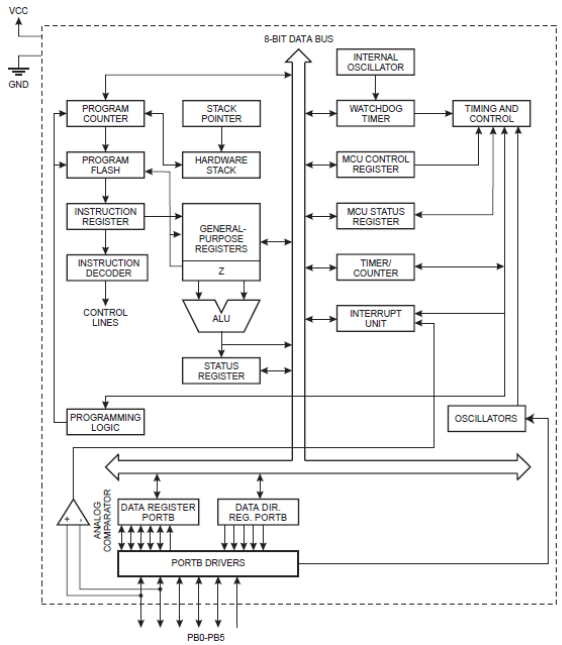


Figura 2.6 Diagrama de blocs del microcontrolador ATtiny 11(Font: [4])

Capítol 3. Disseny del microprocessador

3.1 Set d'instruccions

El *core* AVR1 consta de 90 instruccions, que es separen en instruccions aritmètiques, de branca, de transferència i de bit. Les instruccions aritmètiques executen operacions aritmètiques i lògiques amb registres i números Immediats, i les instruccions de bit fan canvis lògics a bits específics dels registres. Les instruccions de branca inclouen els salts condicionals i incondicionals dins de les línies del programa, mentre que les de transferència s'encarreguen del moviment de dades entre registres i a la memòria.

3.1.1 Instruccions Aritmètiques

Aquestes instruccions fan servir la ALU per dur a terme operacions aritmètiques. Els operands poden ser tant registres com números Immediats. Hi han altres grups de instruccions que fan servir la ALU, però el que distingeix a les operacions aritmètiques de les demès és que l'operació és la fi de la instrucció en si, a diferencia de, per exemple, una instrucció que resti 2 números per saber si són iguals.

Després de cada operació aritmètica, la ALU actualitza el valor del registre d'estat (SREG), donant informació sobre l'operació que ha executat. El registre SREG i els seus components s'expliquen en més detall a l'apartat 3.1.4.

La llista de les instruccions aritmètiques es troba a la Taula 3.1.

Taula 3.1 Conjunt d'instruccions aritmètiques i lògiques

INSTRUCCIONS ARITMÈTIQUES			
Nom	**Operands**	**Descripció**	**Operació**
ADD	Rd, Rr	Afegir dos registres sense portar	Rd ← Rd + Rr
ADC	Rd, Rr	Afegir dos registres amb *Carry*	Rd ← Rd + Rr + C
SUB	Rd, Rr	Restar dos registres	Rd ← Rd-Rr
SUBI	Rd, K	Restar constant del registre	Rd ← Rd-K
SBC	Rd, Rr	Restar dos registres amb *Carry*	Rd ← Rd-Rr-C
SBCI	Rd, K	Restar constant del registre amb Carry.	Rd ← Rd-K-C
AND	Rd, Rr	Operació lògica AND entre registres	Rd ← Rd • Rr
ANDI	Rd, K	Operació lògica AND entre registre i constant	Rd ← Rd • K
OR	Rd, Rr	Operació lògica OR entre registres	Rd ← Rd v Rr
ORI	Rd, K	Operació lògica OR entre registre i constant	Rd ← Rd v K
EOR	Rd, Rr	Operació lògica XOR entre registres	Rd ← Rd ⊕ Rr
COM	Rd	El complement a 1 d'un registre	Rd ← 0xFF-Rd
NEG	Rd	Complement a 2 d'un registre	Rd ← 0x00-Rd
SBR	Rd, K	Posa a '1' alguns bits del registre	Rd ← Rd v K
CBR	Rd, K	Posa a '0' alguns bits del registre	Rd ← Rd • (0xFF-K)
INC	Rd	Incrementa el registre	Rd ← Rd + 1
DEC	Rd	Decrementa el registre	Rd ← Rd-1

TST	Rd	Prova per a zero o menys	Rd ← Rd • Rd
CLR	Rd	Posa tot el registre a '0'	Rd ← Rd ⊕ Rd
SER	Rd	Posa tot el registre a '1'	Rd ← 0xFF

3.1.2 Operacions de Branca

Les operacions de branca modifiquen el comptador de programa (PC), regulant així l'ordre en el que s'executen les instruccions. Serveixen per crear iteracions, subrutines o arbres de decisió dins del programa. Els salts poden ser condicionals, que depenen de l'estat d'un bit o d'un registre per executar el salt, o incondicionals, que fan el salt sempre que s'executa la instrucció.

Dins les instruccions de salt incondicional, trobem el salt normal i el salt a subrutina. Ambdós salts són relatius, que vol dir que s'indica a la instrucció quantes línies cal moure el PC, no a quina línia cal que vagi, però la diferencia entre els dos tipus de salt prové de que les instruccions RCALL, de salt a subrutina, guarden l'adreça del PC prèvia al salt a la *Stack*, l'element de memòria encarregat de guardar valors de forma temporal. Aquest registre s'anomena *stack*, que significa "pila" en anglès, perquè carrega els valors d'entrada al registre superior, baixant les adreces guardades anteriorment a registres inferiors, eliminant l'adreça guardada a l'últim registre, com es pot veure a la il·lustració de la Figura 3.1.

La llista de les instruccions de branca es troba a la Taula 3.2.

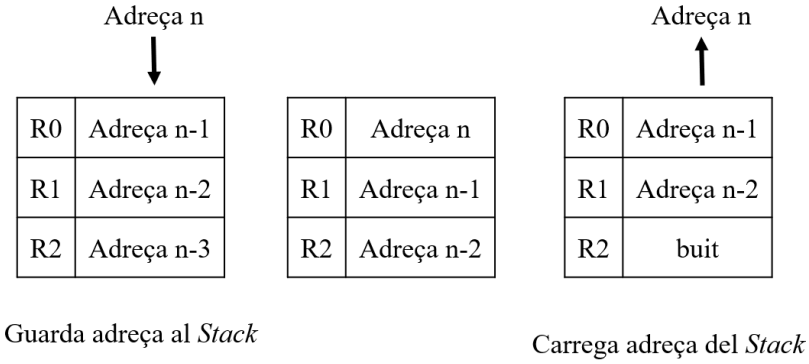


Figura 3.1 Esquema del funcionament de la memòria Stack

Taula 3.2 Conjunt d'instruccions de branca

INSTRUCCIONS DE BRANCA			
Nom	**Operands**	**Descripció**	**Operació**
RJMP	k	Salt Relatiu	PC←PC+k+1
RCALL	k	Salta relatiu a subrutina	PC←PC+k+1
RET	-	Retorn de la subrutina	PC ← STACK
RETI		Retorn de la interrupció	PC ← STACK
CPSE	Rd, Rr	Compara, omet si son iguals	si (Rd = Rr) : PC ← PC + 2
CP	Rd, Rr	Compara	Rd – Rr
CPC	Rd, Rr	Compara amb *Carry*	Rd - Rr – C
CPI	Rd, K	Compara registre amb constant	Rd – K
SBRC	Rr, b	Omet si el bit en el Registre és ‘0’	si (Rr(b)=0) : PC ← PC + 2
SBRS	Rr, b	Omet si el bit en el Registre és ‘1’	si (Rr(b)=1) : PC ← PC + 2
SBIC	A, b	Omet si el bit en el Reg. I/O és ‘0’	si (I/O(A,b)=1) : PC ← PC + 2
SBIS	A, b	Omet si el bit en el Reg. I/O és ‘1’	si (I/O(A,b)=1) : PC ← PC + 2
BRBS	s, k	Salta si el bit de SREG és ‘1’	si (SREG(s) = 1) : PC←PC+k + 1
BRBC	s, k	Salta si el bit de SREG és ‘0’	si (SREG(s) = 0) : PC←PC+k + 1
BREQ	k	Salta si la bandera Z és ‘1’	si (Z = 1) : PC ← PC + k + 1
BRNE	k	Salta si la bandera Z és ‘0’	si (Z = 0) : PC ← PC + k + 1
BRCS	k	Salta si la bandera C és ‘1’	si (C = 1) : PC ← PC + k + 1
BRCC	k	Salta si la bandera C és ‘0’	si (C = 0) : PC ← PC + k + 1
BRSH	k	Salta si Rr ≥ Rd	Salta (C = 0) : PC ← PC + k + 1
BRLO	k	Salta si Rr < Rd	si (C = 1) : PC ← PC + k + 1
BRMI	k	Salta si Rd < 0	si (N = 1) : PC ← PC + k + 1
BRPL	k	Salta si Rd > 0	si (N = 0) : PC ← PC + k + 1
BRGE	k	Salta si Rr ≥ Rd, amb signe	si (N ⊕ V= 0) : PC ← PC + k + 1
BRLT	k	Salta si Rd < 0, amb signe	si (N ⊕ V= 1) : PC ← PC + k + 1
BRHS	k	Salta si la bandera H és ‘1’	si (H = 1) : PC ← PC + k + 1
BRHC	k	Salta si la bandera H és ‘0’	si (H = 0) : PC ← PC + k + 1
BRTS	k	Salta si la bandera T és ‘1’	si (T = 1) : PC ← PC + k + 1
BRTC	k	Salta si la bandera T és ‘0’	si (T = 0) : PC ← PC + k + 1
BRVS	k	Salta si la bandera V és ‘1’	si (V = 1) : PC ← PC + k + 1
BRVC	k	Salta si la bandera V és ‘0’	si (V = 0) : PC ← PC + k + 1
BRIE	k	Salta si la bandera I és ‘1’	si (I = 1) : PC ← PC + k + 1
BRID	k	Salta si la bandera H és ‘0’	si (I = 0) : PC ← PC + k + 1

3.1.3 Operacions de transferència

Les operacions de transferència mouen informació entre els diferents registres del microprocessador. Algunes instruccions, com MOV o LDI, mouen dades entre registres del banc de registres, d’altres, com LD o ST, mouen dades entre el banc de registres i la memòria

RAM. Finalment, les instruccions IN i OUT reben o envien dades a l'exterior del microprocessador, mitjançant els pins de sortida o entrada.

Les operacions de moviment a la RAM poden tenir 3 variants de càrrega, siguin amb desplaçament, post incrementals o post decrementals. Per simplificar, s'ha marcat el desplaçament com a constant a 0, i l'adreça a la qual guardar o carregar les dades es marca com a registre Z, un registre de 16 bits compost pels últims 2 nivells del banc de registres R30 i R31, anomenats també ZLo i ZHi, respectivament. Com que la RAM que s'ha dissenyat en aquest treball és de només 64 bytes, el registre ZLo ja pot apuntar a qualsevol adreça de la RAM. De totes formes, s'ha implementat el microprocessador amb el potencial de poder funcionar amb memòries RAM de 64K bytes.

Amb es operacions de transferència de dades a l'exterior trobem una situació similar. En l'arquitectura AVR1, es dona la possibilitat d'escollir entre vàries sortides/entrades, usant els bits etiquetats amb una A al codi d'instrucció. Tot i això, s'ha decidit, per motius de simplicitat, posar automàticament A a 0 i només implementar una sortida i una entrada de dades.

La llista de les instruccions de transferència es troba a la Taula 3.3.

Taula 3.3 conjunt d'instruccions de transferència

INSTRUCCIONS DE TRANSFERÈNCIA			
Nom	**Operands**	**Descripció**	**Operació**
MOV	Rd, Rr	Mou dades entre registres	Rd ← Rr
LDI	Rd, K	Carrega un valor Immediat a un registre	Rd ← K
LD	Rd, Z	Carrega el contingut d'un registre a la RAM	Rd ← (Z)
ST	Z, Rr	Guarda el contingut de la RAM a un registre	(Z) ← Rr
IN	Rd, A (0)	Rep informació dels pins d'entrada	Rd ← I/O
OUT	A (0), Rr	Treu informació pels pins de sortida	I/O ← Rr

3.1.4 Operacions de bit

Les operacions de bit operen sobre el contingut dels registres, tant del registre d'estat (SREG) com dels registres R0 – R31, del banc de registres. Algunes operacions posen a '1' o a '0' bits específics dels registres, i d'altres fan alteracions a l'ordre dels bits.

Algunes de les operacions, com els *shifts* tenen com a conseqüència operacions aritmètiques. Per exemple, un *shift* lògic cap a l'esquerra significa que el valor decimal del vector es veu doblat de valor. És per això que algunes de les operacions, específicament les rotacions i *shifts* cap a l'esquerra, es fan executant aquestes operacions matemàtiques, així reduint el nombre d'operacions que pot fer la ALU i, per tant, la quantitat de recursos lògics consumits en la seva implementació a la FPGA.

El registre SREG és un registre de 8 bits que conté indicadors, o *flags*, que aporten informació sobre l'operació que s'ha executat a l'ALU. Els *flags* que composen el registre *SREG* i la seva funció es troben resumits a la Taula 3.4 i s'expliquen a continuació:

Taula 3.4 Resum dels bits de SREG i la seva funció

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
I	T	H	S	V	N	Z	C
Interrupcions	Guarda Bit	*Half-carry*	Signe	*Overflow*	Negatiu (C2)	Zero	*Carry*

7. I: Indica que s'ha activat una interrupció. En aquest projecte és innecessari, però s'ha incorporat igualment per facilitar el desenvolupament futur d'interrupcions.

6. T: Emmagatzematge de bit: Les instruccions de còpia de bit BLD (càrrega de bits) i BST (bit STore) utilitzen el bit T com a font i la destinació del bit operat. Un bit d'un registre en el banc de registres es pot copiar a T per la instrucció del BST, i un bit en T es pot copiar a un bit en un registre del banc de registres amb la instrucció BLD.

5. H: *Half-Carry*: El bit H indica un *half-carry* en algunes operacions aritmètiques. Si aquest *flag* està activat significa que hi ha hagut un *carry* al bit 3, el bit més significatiu de la meitat menys significativa de la paraula de 8 bits. Això és útil quan es realitzen operacions aritmètiques amb números amb format BCD. Per exemple, si es produeix una suma de dos números de 8 bits en format BCD, cal ajustar el resultat per que sigui correcte. A la suma A de la Figura 3.2, com que el resultat de la suma inclou un número no decimal (C) a la segona xifra BCD, cal sumar-hi un 06_{16} per corregir el valor. A la suma B, el número no decimal és a la primera xifra BCD (C), així que, per corregir el resultat, cal sumar-hi 60_{16}. A la suma C, en canvi, no hi ha cap xifra no decimal, però cal fer una correcció. Això és perquè hi ha hagut un *carry*, és a dir que s'ha "portat" un '1' del bit 3 al bit 4. Per tant, cal sumar-li 06_{16} per que el resultat sigui correcte. Aquesta situació no seria perceptible pel sistema si no fos pel *flag* de *half-carry*.

24	0010 0100		72	0111 0010		39	0011 1001	
+48	+0100 1000		+63	+0110 0011		+29	+0010 1001	
6C	0110 1100		C5	1100 0101		62	0110 0010	
+06	+0000 0110		+60	+0110 0000		+06	+0000 0110	
72	0111 0010		135	1 0011 0101		68	0110 1000	
A)			B)			C)		

Figura 3.2 Exemples d'operacions amb números en format BCD i l'efecte sobre el flag H

4. S: Signe: El bit és el resultat de l'operació lògica XOR del bit N i bit V ($S = N \oplus V$), els bits de negació i el bit d'*overflow*, respectivament, que s'explicaran a continuació.

3. V: Desbordament en el complement a 2: Aquest *flag* s'activa quan, en una operació aritmètica, el resultat expressat en Complement a 2 no es podria mostrar amb els 8 bits, és a dir que es requeririen més bits, com es mostra en els exemples mostrats a la Figura 3.3, on es veu que quan una operació de suma de dos números positius dona un resultat que, representat en 8 bits, és aparentment negatiu (el seu bit de més pes es '1'), ha ocorregut un desbordament, o *overflow*. En el segon cas, de suma de valors negatius en complement a 2, passa el mateix quan el resultat és aparentment positiu (bit de més pes a '0'). En ambdós casos el bit V es posaria a '1' per que el sistema fos conscient d'aquesta situació.

$$\begin{array}{rr} 127 & 0111\ 1111 \\ +127 & +\ \ 0111\ 1111 \\ \hline 254 & (0)\ 1111\ 1110 \end{array} \qquad \begin{array}{rr} -81 & 1010\ 1111 \\ +\ -119 & +\ \ 1000\ 1001 \\ \hline -200 & (1)\ 0011\ 1000 \end{array}$$

Figura 3.3 Exemples de l'activació del flag de Overflow

2. N: Negatiu: S'activa quan el bit més significatiu del resultat està a '1'. Indica que el número en complement a 2 és un número negatiu.

1. Z: Zero: Indica un resultat nul després d'una operació aritmètica o lògica.

0. C: *Carry:* Aquest bit indica que hi ha hagut un *carry*, o que s'ha "portat" un '1' en la suma de l'últim bit en una operació de suma o que s'ha fet un *borrow*, o préstec, en una operació de resta. En la suma, el carry serveix quan es sumen dos valors sense signe i el resultat de la suma no es pot expressar correctament amb els bits disponibles (8 en aquest cas). En la resta, expressada com $a - b$, indica que $a<b$, o que el resultat hauria d'interpretar-se com a negatiu. També serveix per les instruccions aritmètiques de suma amb *carry* o resta amb *carry*, on s'executa l'operació amb els dos operands i el valor de carry.

La llista de les instruccions de bit es troba a la Taula 3.5.

Taula 3.5 Conjunt d'instruccions de bit

INSTRUCCIONS DE BIT			
Nom	**Ops.**	**Descripció**	**Operació**
SEC		Posa el *flag* C a nivell alt	C ← 1
CLC		Posa el *flag* C a nivell baix	C ← 0
SEN		Posa el *flag N* a nivell alt	N ← 1
CLN		Posa el *flag N* a nivell baix	N ← 0
SEZ		Posa el *flag Z* a nivell alt	Z ← 0
CLZ		Posa el *flag Z* a nivell baix	Z— 0
SEI		Posa el *flag I* a nivell alt	I ← 1
CLI		Posa el *flag I* a nivell baix	I ← 0
SES		Posa el *flag S* a nivell alt	S ← 1
CLS		Posa el *flag S* a nivell baix	S ← 0

SEV		Posa el *flag V* a nivell alt	V ← 1
CLV		Posa el *flag V* a nivell baix	V ← 0
SET		Posa el *flag T* a nivell alt	T— 1
CLT		Posa el *flag T* a nivell baix	T— 0
SEH		Posa el *flag H* a nivell alt	H ← 1
CLH		Posa el *flag H* a nivell baix	H ← 0
LSL	Rd	*Shift* Lògic a l'esquerra	Rd(n+1) ← Rd(n), Rd(0) ← 0
LSR	Rd	*Shift* Lògic a la dreta	Rd(n) ← Rd(n+1), Rd(7) ← 0
ROL	Rd	*Shift* per *Carry* a l'esquerra	Rd(0)←C,Rd(n+1)← Rd(n), C←Rd(7)
ROR	Rd	*Shift* per *Carry* a la dreta	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)
ASR	Rd	*Shift* aritmètic a la dreta	Rd(n) ← Rd(n+1), n=0...6
SWAP	Rd	Intercanvia *nibbles*	Rd(3...0)←Rd(7...4),Rd(7...4)←Rd(3...0)
BSET	s	Posa *flag* a nivell alt	SREG(s) ← 1
BCLR	s	Posa *flag* a nivell baix	SREG(s) ← 0
SBI	A, b	Posa un bit del port I/O a nivell alt	I/O(A, b) ← 1
CBI	A, b	Posa un bit del port I/O a nivell baix	I/O(A, b) ← 0
BST	Rr, b	Guarda un bit al *flag* T	T ← Rr(b)
BLD	Rr, b	Carrega el bit del *flag* T	Rd(b) ← T

3.2 Codificació de les instruccions

Com s'ha explicat a l'apartat 2.1, per poder implementar les instruccions que conformen els programes a executar cal traduir les instruccions a codi màquina. Cada arquitectura codifica aquestes instruccions de codi màquina de forma diferent, però cada instrucció ha d'incloure: l'operació a executar i els operands amb els quals executar-la. Quan el microprocessador està en marxa, rep aquest codi de la memòria de programa i la descodifica per extreure la informació necessària per la seva execució correcta.

Per fer això, es divideixen les instruccions en 16 formats, segons les operacions que fan, els operands que utilitzen o el tipus d'adreçament que usen. El resum dels diferents tipus de format es poden veure a la Figura 3.4, i els formats de les instruccions es poden trobar en més detall a l'Annex A.

CODIFICACIÓ DE LES INSTRUCCIONS AVR1

Format 1 | 00nn | nnrd | dddd | rrrr |
Instruccions aritmètiques entre 2 registres

Format 2 | 1001 | 010d | dddd | nnnn |
Instruccions aritmètiques amb 1 registre

Format 3 | 01nn | KKKK | dddd | KKKK |
Instruccions aritmètiques entre un registre i una constant

Format 4 | 10q0 | qqnd | dddd | nqqq |
Instruccions de transferència amb la memòria RAM

Format 5 | 1111 | 1nnd | dddd | 0bbb |
Instruccions de bit i de salt respecte un registre

Format 6 | 1011 | nAAd | dddd | AAAA |
Instruccions de transferència amb els ports d'entrada i sortida

Format 7 | 1001 | 10nn | AAAA | Abbb |
Instruccions de bit amb els ports d'entrada i sortida

Format 8 | 1001 | 0100 | nsss | 1000 |
Instruccions de bit respecte el registre SREG

Format 9 | 1110 | KKKK | dddd | KKKK |
Instruccions de transferència amb un registre i una constant

Format 10 | 0011 | KKKK | dddd | KKKK |
Instrucció de comparació entre un registre i una constant

Format 11 | 1001 | 000d | dddd | 0100 |
Instrucció de transferència sobre la memòria de programa

Format 12 | 110n | KKKK | KKKK | KKKK |
Instruccions de branca de salt incondicional

Format 13 | 1001 | 010n | 000n | 100n |
Instruccions de retorn de salt

Format 14 | 1111 | 0nKK | KKKK | Ksss |
Instruccions branca de salt condicional

Format 15 | 1001 | 0101 | 10nn | 1000 |
Instruccions de control del processador

Format 16 | 0000 | 0000 | 0000 | 0000 |
Cap operació

Figura 3.4 Resum dels diferents formats de codificació d'instruccions del microprocessador

3.3 Diagrama de Blocs

En aquest pas del procés de desenvolupament, cal elaborar un diagrama que mostri com les dades es mouran de component en component per poder executar les instruccions de forma correcta. En aquest apartat s'explicarà el procediment seguit en la creació del diagrama. Per facilitar el seu seguiment, les parts ja afegides seran mostrades en gris, i les noves incorporacions seran mostrades de color negre.

3.3.1 Operacions Fetch i Decode

El primer element a tenir en compte és el *Program Counter*. És un registre que guarda l'adreça de la instrucció a executar. Aquesta adreça entra a la memòria Flash, i a la sortida RD es troba el codi de la instrucció pertinent.

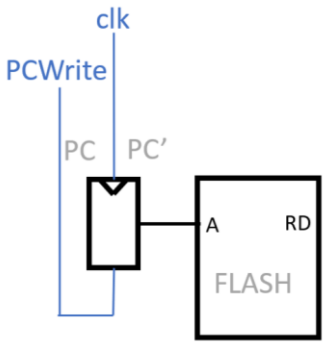


Figura 3.5 Program Counter i memòria Flash, elements bàsics de qualsevol processador

Una vegada s'obté el codi de la instrucció, caldrà descodificar-la com s'ha explicat prèviament. Per què el codi de la instrucció no canviï inesperadament, es posa un registre entre la memòria

Flash i el descodificador. Amb això, ja hi ha prou per poder executar les dues instruccions base: *Fetch* i *Decode*.

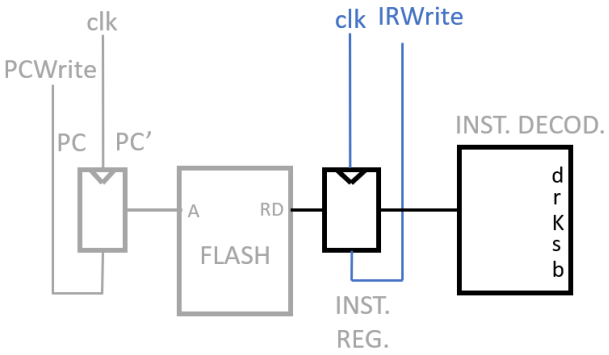


Figura 3.6 Registre i Descodificador d'instruccions

3.3.2 Instruccions aritmètiques: ADD

Per seguir amb el desenvolupament del diagrama de blocs, es tractarà el cas de voler executar una instrucció simple, com la suma de dos registres (*ADD*). Aquesta operació, en llenguatge d'assemblador es representaria com `ADD Rd, Rr`, sent Rd i Rr, respectivament, el registre de destí i el registre d'origen. En codi màquina, que és el codi que permetrà el funcionament del processador en la seva execució, prendrà la forma **"`000 11rd dddd rrrr`"**, on d i r són les adreces dels registres Rd i Rr.

Primer, caldrà accedir al contingut de cada un dels registres, per tant de la sortida *d* i *r* sortiran les adreces dels registres operands. Aquestes adreces aniran a parar al banc de registres, un element que conté 32 registres de 8 bits, que es poden adreçar mitjançant vectors de 5 bits, entre els valors 0 i 31. Les adreces entraran pels ports d'adreça A1 i A2, i el contingut dels registres indicats per *r* i *d* s'entregaran a la sortida *RD1* i *RD2*, els ports de sortida de la lectura de dades del banc de registre. Aquests valors continguts dins el banc de registres seran els operands de l'operació. Aquests operands entren a cada banda de la ALU, i mitjançant *ALUControl* es selecciona quina operació es durà a terme.

A la sortida de la ALU es troba el resultat, un vector de 8 bits que cal carregar al registre de destí pertinent. Per això es connecta l'adreça del registre destí (*Rd*) a l'entrada d'escriptura *A3* i la sortida de la ALU a l'entrada de dades d'escriptura *WD3*, i s'activa el senyal *RegWrite* per habilitar l'escriptura de dades. Amb això, ja s'iniciaria la següent instrucció, però no s'ha carregat el nou valor al *PC*, així que quedaria parat el processador.

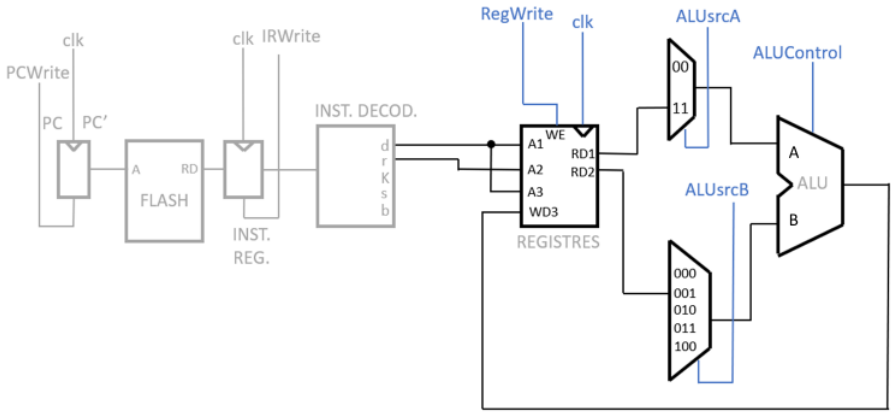


Figura 3.7 Banc de Registres, multiplexors de selecció d'operands i ALU

Per actualitzar el valor del PC, simplement cal sumar '1' al valor actual. Per millorar el temps d'execució, aquest procés es duu a terme durant la instrucció *fetch*, així preparant el nou valor abans d'haver acabat la instrucció present.

Per incrementar el valor del PC, cal portar el valor de PC a la ALU, i seleccionar per l'altre operand l'entrada del multiplexor amb valor '1'. A la sortida de la ALU, s'envia el resultat cap al PC, activant el senyal *PCWrite* per actualitzar el valor del PC.

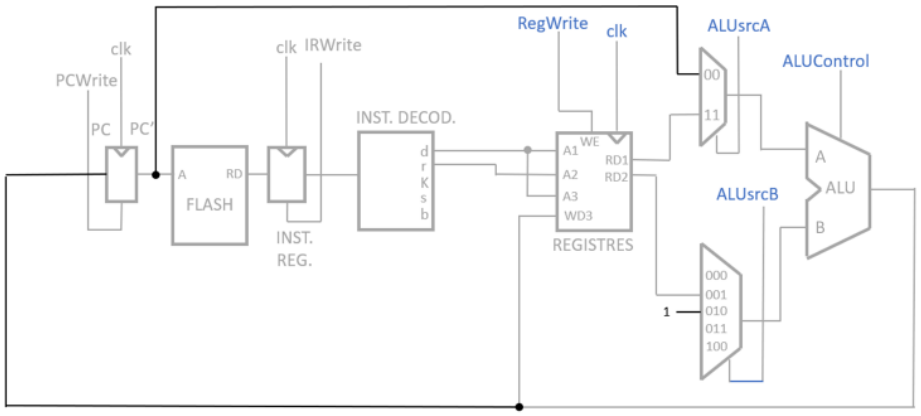


Figura 3.8 Càrrega de PC a la ALU i la realimentació posterior

3.3.3 Instruccions de transferència: ST, LD, MOV, IN, LDI i OUT

Ara es suposarà que la següent instrucció és ST, escriure el valor d'un registre a la memòria RAM. LA RAM és un banc de memòria de 64 adreces de 8 bits, que es poden modificar de forma independent mitjançant adreçament directe. Aquesta instrucció en llenguatge d'assemblador pren la forma `ST Z, Rr`, i en llenguatge màquina `"1000 001r rrrr 0000"` on 'r' de nou és l'adreça del registre d'origen i 'Z' fa referència al registre Z del banc de registres, un vector de 16 bits que serveix per adreçar la memòria RAM. Com que els registres

del banc de registres només són de 8 bits de llargada, el registre Z ocupa els registres R30 i R31, on R30 conté la meitat menys significativa, anomenada ZLo, i R31 guarda el byte de més pes, ZHi.

Els ports d'entrada i sortida de la memòria RAM son A, l'entrada de l'adreça de la memòria que es vol manipular, WD, per on entren les dades que es volen carregar a l'adreça de l'entrada A i RD, la sortida per on s'emeten les dades guardades en l'adreça de la memòria A. Finalment, l'entrada d'habilitació WE, *Write Enable*, permet, a nivell alt, l'escriptura de les dades entrades per l'entrada WD a l'adreça A.

Per carregar el valor de la memòria RAM al registre Rd, es seguiran els mateixos passos de *fetch* i *decode*, s'extraurà el valor de ZLo i Rd per les sortides RD1 i RD2 respectivament, i es portarà RD2 a l'entrada de dades de la RAM. Per fer que l'adreça a la sortida RD1 arribi a la RAM, caldrà fer-la passar per la ALU, amb l'operació *Pass*. A la sortida de la ALU, es carregarà a un registre, que mantindrà l'adreça a l'entrada estable mentre es produeix la càrrega de dades a la RAM, que s'habilitarà mitjançant la posada a '1' del senyal de control *MemWrite*.

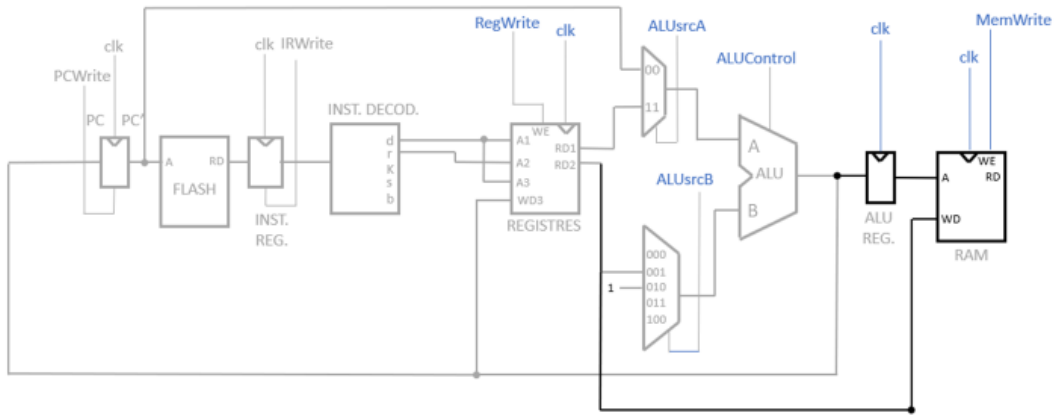


Figura 3.9 Càrrega de dades a la memòria RAM

Si en canvi de carregar valors a la RAM es vol llegir valors de la memòria i guardar-los als registres, caldrà executar la instrucció LD. Aquesta instrucció prendrà el format `LD Rd, Z` i el seu codi màquina serà `"1000 001d dddd 0000"`, on 'd' contindrà l'adreça del banc de registres on guardar les dades i Z, de nou, serà l'adreça de la memòria RAM a la que cal accedir.

El procediment d'execució d'aquesta instrucció serà més o menys similar a l'anterior. L'adreça (ZLo) seguirà sortint de la sortida RD1 i passant per la ALU i el registre de la seva sortida abans d'arribar a la RAM. Ara, amb *MemWrite* desactivat, no escriurà dades a la memòria, sinó que les llegirà i traurà el seu contingut per la sortida RD.

Per poder escriure dades de la memòria RAM a un registre del banc de registres caldrà seguir un procediment similar al de l'escriptura del resultat de la ALU, amb la diferència de que l'entrada d'informació serà la sortida de la RAM. Per això s'incorpora un multiplexor que, depenent del valor de *REGSrc*, portarà la sortida de la RAM o de la ALU a l'entrada de dades d'escriptura del banc de registres.

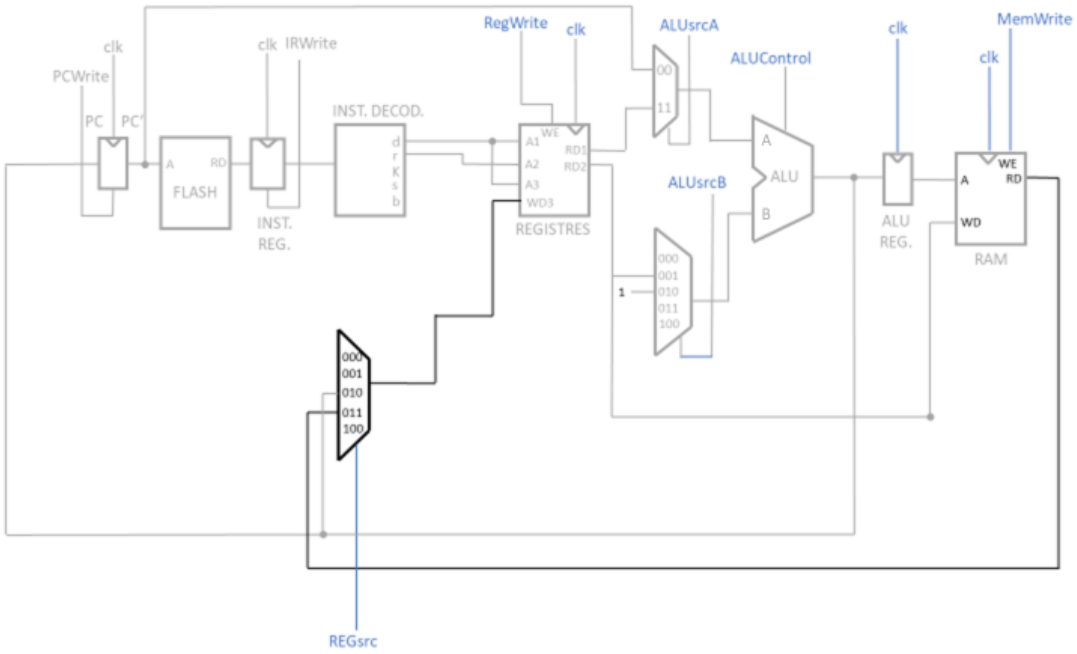


Figura 3.10 Càrrega de dades de la RAM al banc de registres

Per continuar amb les operacions de transferència de dades, es suposarà que es busca moure dades d'un registre del banc de registres a un altre. Aquesta instrucció, MOV, prendrà la forma `MOV Rd, Rr` en llenguatge d'assemblador i `"0010 11rd dddd rrrr"` en codi màquina, on Rd i Rr seran els registres de destí i origen.

L'execució de la instrucció MOV requerirà portar el contingut de l'adreça A2, present la sortida RD2 del banc de registres, a l'entrada WD3 del mateix banc de registres, passant pel multiplexor prèviament incorporat i activant el senyal de *REGWrite* amb el valor adequat.

Si es vol guardar informació provinent de l'exterior, mitjançant els pins d'entrada de dades, caldrà executar l'operació IN. Aquesta operació es representa en llenguatge assemblador com `IN Rd, 0` i en codi màquina pren la forma `"1011 0AAd dddd AAAA"`, on 'd' és l'adreça del registre de destí, i A indica quin port utilitzar per rebre dades de l'exterior. En aquest cas, com que només hi ha un port d'entrada de dades, A sempre valdrà zero.

En l'execució de la instrucció IN caldrà incorporar un registre que guardi el valor primer que entri pel port d'entrada, que s'anomenarà *IREG* i després, mitjançant el multiplexor, portar el contingut d'aquest registre a l'entrada de dades del banc de registres, WD3. La sortida del descodificador d'instruccions 'd' portarà l'adreça d'escriptura de dades al port A3 del banc de registres, on s'indica el registre on guardar les dades de IREG.

Si el que es requereix és carregar una constant a un registre, amb la instrucció LDI (`LDI Rd, K, "1110 KKKK dddd KKKK"`, on K és la constant a guardar al registre Rd) s'haurà de connectar la sortida K del descodificador d'instruccions al multiplexor de selecció de l'entrada del banc de registres WD3. Així, depenent del valor de *REGsrc* es podrà escollir quina informació carregar al banc de registres.

Finalment, si es desitja treure informació del banc de registres a través dels pins de sortida (Operació OUT: `OUT A, Rr; "1011 1AAr rrrr AAAA"`) caldrà un nou registre, anomenat OREG, que guardi els valors de sortida de RD2 i els aboqui als pins de sortida.

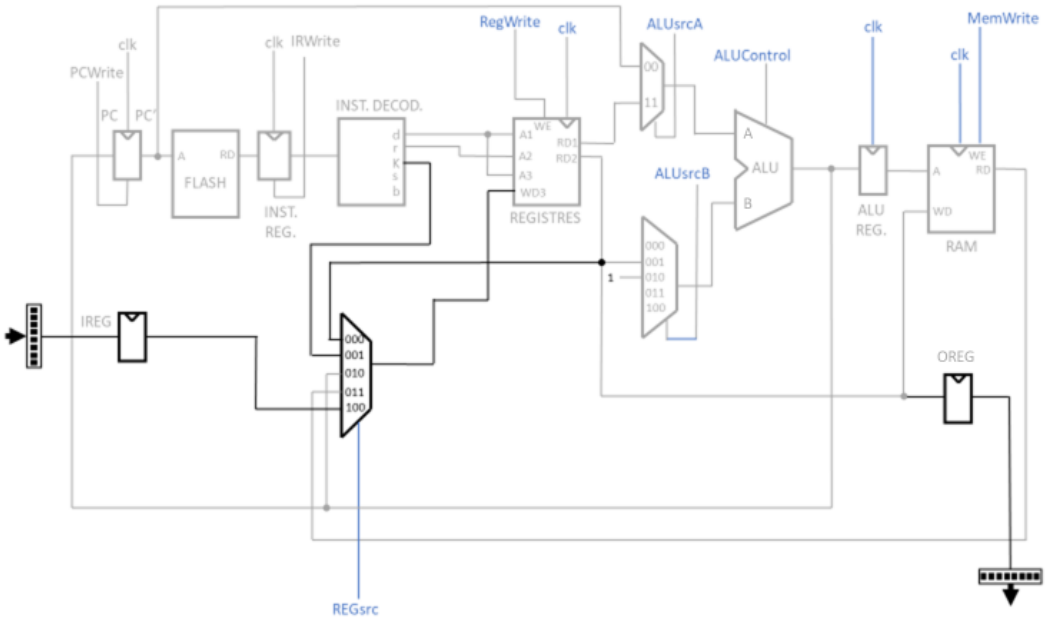


Figura 3.11 Entrada i sortida de dades per IREG i OREG i càrrega de la constant K i de la sortida RD2 al banc de registres

Amb aquest esquema del datapath ja es podrien executar casi totes les instruccions aritmètiques i de transferència.

3.3.4 Datapath de les operacions de Bit Set i Bit Clear

El següent pas en el desenvolupament del datapath serà habilitar l'execució de les instruccions de *Bit Set* i *Bit Clear*. Aquestes instruccions modifiquen bits tant en el banc de registres com en el registre d'estat, *SREG*. El registre *SREG* es carrega amb dades de la segona sortida de la ALU, que activa els *flags* pertinents després de cada operació. Encara que la càrrega dels valors dels senyals de *SREG* es faci de forma automàtica, amb les operacions de bit es poden modificar aquests *flags*, per exemple per activar el *carry* abans d'una rotació per carry, on el valor que s'afegirà a la punta del vector serà el valor del bit C de *SREG*. Amb la incorporació del registre SREG, també es podran executar les instruccions aritmètiques amb carry. Per carregar els nous valors a SREG, caldrà activar el senyal *SREGLoad*.

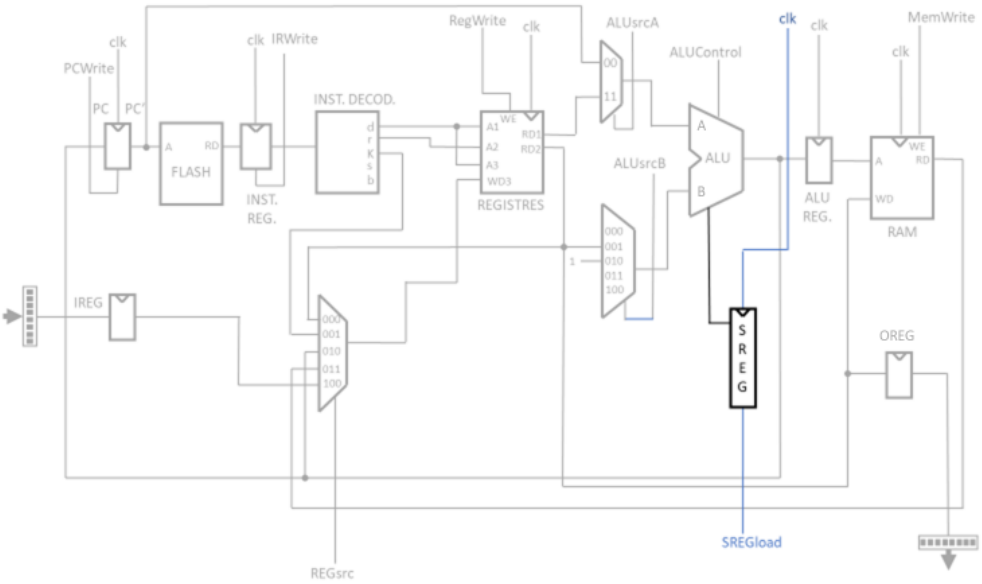


Figura 3.12 Registre d'estat, SREG

Per poder executar una instrucció de bit que actuï sobre *SREG*, caldrà saber el valor actual del registre, quin bit modificar i el valor desitjat del bit. Per saber el valor actual de *SREG*, s'afegeix una sortida sèrie al registre. El bit a modificar ve marcat per la instrucció, així que la informació de quin bit modificar està a la sortida 's' del descodificador, i el valor del bit desitjat està vinculat al valor de *BitSet*, que en torn depèn del component 'n' de la instrucció.

Aquestes dades van a un component, el codificador de *SREG*, que modifica el bit 's' i el posa a l'estat que dicta *BitSet*. La sortida, *SREG'*, ha de ser carregada al registre, així que s'incorpora un multiplexor per escollir quin valor de *SREG* carregar. El senyal que controlarà el multiplexor s'anomena *SREGsrcC*.

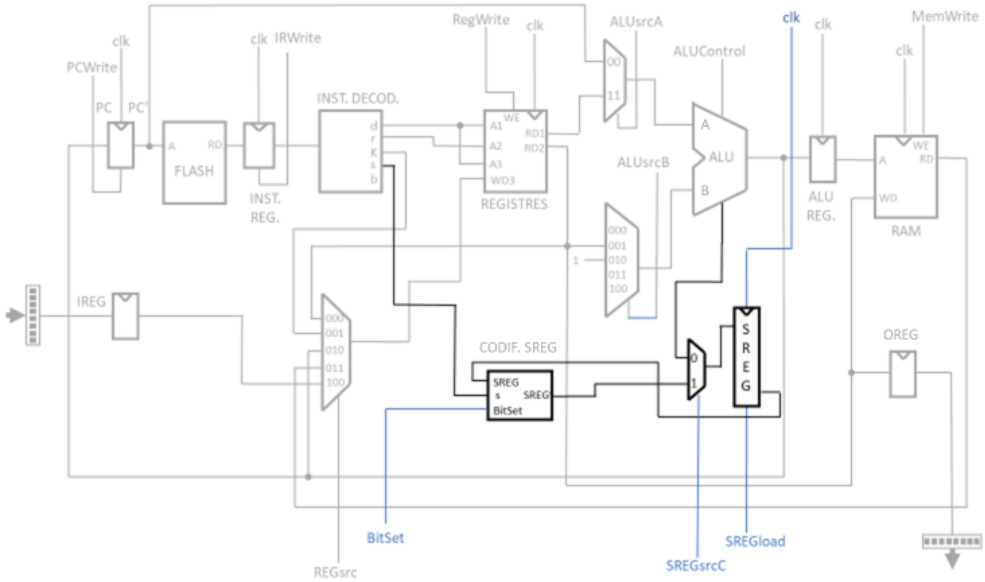


Figura 3.13 Descodificador de SREG i el multiplexor SREGsrcC

Le següents instruccions a analitzar seran BLD i BST, que quan s'executen guarden o carreguen el valor d'un bit d'un registre del banc de registres al bit T de *SREG*. Es suposa que primer caldrà guardar el bit a T, així que es comença amb la instrucció BST.

Per executar BST caldrà saber quin bit copiar, de quin registre i el seu valor. Aquest valor, que s'anomena Rr(b), serà el valor que es guardi a T. Per tant, si aquest valor és '1', T ha de ser posat a '1', i de forma igual si és '0'. Això significa que Rr(b) farà la mateixa funció que *BitSet*, així que es connectarà a la mateixa entrada del codificador de *SREG*, mitjançant un multiplexor controlat pel senyal *SREGsrcB*.

A l'entrada 's' del codificador de *SREG* caldrà indicar quin bit és el que s'ha de modificar. Com que aquest bit sempre serà el bit T, que te la sisena posició a SREG, es connectarà a l'entrada 's' un valor fix de '6', mitjançant un multiplexor controlat pel senyal *SREGsrcA*.

Així, es guardarà el nou valor de T a SREG i es carregarà al registre com s'ha explicat prèviament, activant la càrrega del registre amb *SREGLoad* i escollint la sortida del descodificador de *SREG* com a font del nou vector.

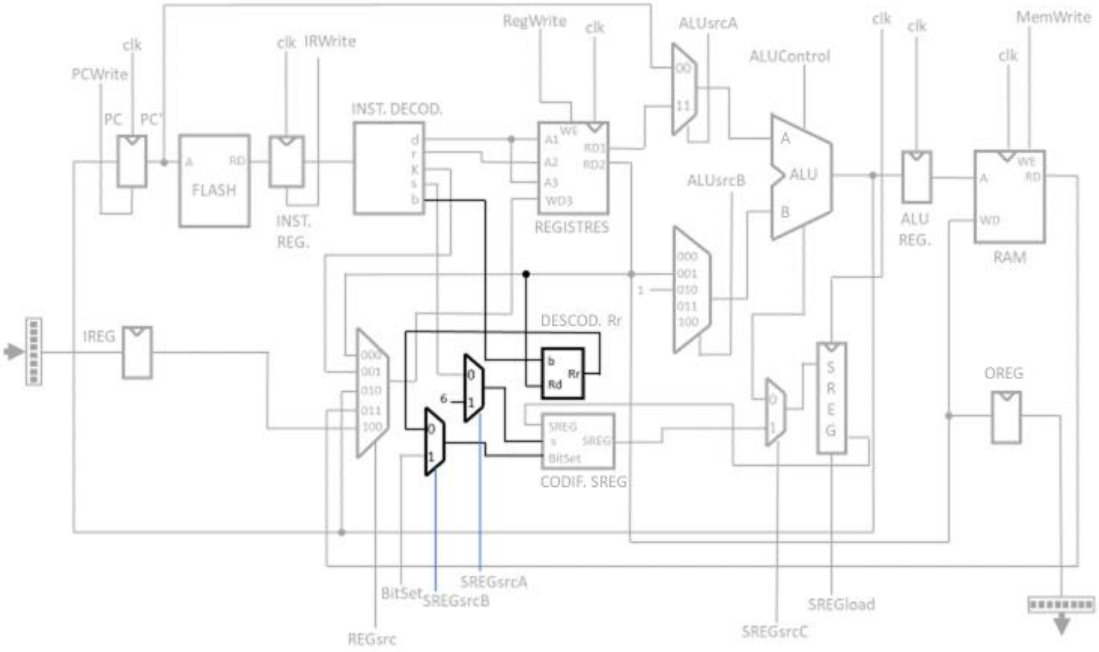


Figura 3.14 Descodificador Rr, multiplexors SREGsrcA i SREGsrcB

Una vegada guardat el valor de T a SREG, si es vogués tornar a carregar a un bit d'un registre, caldria aplicar la instrucció BLD. Per realitzar-ho, cal saber el valor de T, generar un vector amb el que modificar el contingut del registre destinatari i fer l'operació pertinent (AND o OR) segons el bit a escriure sigui un '0' o un '1'.

Del descodificador s'obté 'b', el bit del registre a modificar, i de *SREG* s'obté el valor de T mitjançant la sortida en paral·lel del registre i un multiplexor controlat pel senyal *FlagSelect*. Aquestes entrades modificaran el vector de sortida del codificador, ja que si T, el bit a escriure, val '1', el vector serà un vector de '0's amb un '1' on marqui 'b', i si T val '0' serà un vector ple de '1' excepte el bit 'b', que serà '0'.

Aquest vector després, mitjançant el multiplexor, arribarà a la ALU al port de l'operand B. Per poder escriure el valor del bit 'b' al registre s'executarà l'operació AND en cas de voler posar el bit 'b' a '0' o l'operació OR en cas de voler escriure un '1'.

$$b = 4 \qquad Rd = \text{"01001101"}$$
$$T =' 1' \qquad B = \text{"00010000"}$$

$$\begin{array}{rr} & 01001101 \\ OR & 00010000 \\ \hline & 01011101 \end{array}$$

$$b = 3 \qquad Rd = \text{"01001101"}$$
$$T =' 0' \qquad B = \text{"11110111"}$$

$$\begin{array}{rr} & 01001101 \\ AND & 11110111 \\ \hline & 01000101 \end{array}$$

Figura 3.15 Funcionament del descodificador B i com es posa un bit del registre a '1' o '0'

Una vegada surt el vector resultat de la ALU, aquest es tornarà a guardar a la mateixa adreça, com s'ha explicat prèviament, mitjançant el multiplexor d'entrada al banc de registres i l'activació del senyal *RegWrite*.

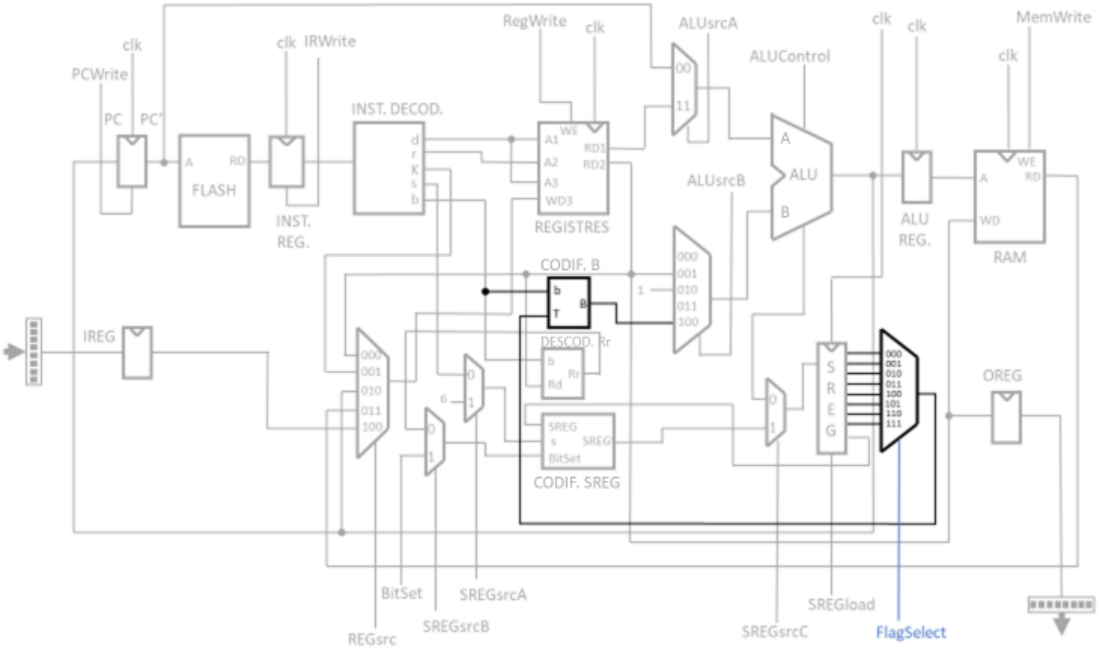


Figura 3.16 Multiplexor de selecció de flags i el descodificador B

Amb les instruccions de bit finalitzades, finalment tocarà executar les instruccions de salt o de branca. Per començar, s'estudiaran els salts més senzills, els salts relatius. Aquets salts s'executen sumant un valor immediat (K) al valor actual de PC. Per poder executar aquest salt, caldrà obtenir el següent valor de PC, mitjançant el multiplexor de l'operand A de la ALU, el valor de K, que sortirà del descodificador d'instruccions, i sumar-los a la ALU. Després, caldrà carregar el nou valor de PC. Aquest valor de PC no s'extraurà de la sortida de la ALU, com s'havia fet prèviament, sinó que s'extraurà després del registre, per reduir el camí lògic que fan les dades en un sol cicle de rellotge i disminuir el *delay* associat a la instrucció. Per escollir d'on es vol extreure el valor de PC + K cal afegir un nou multiplexor.

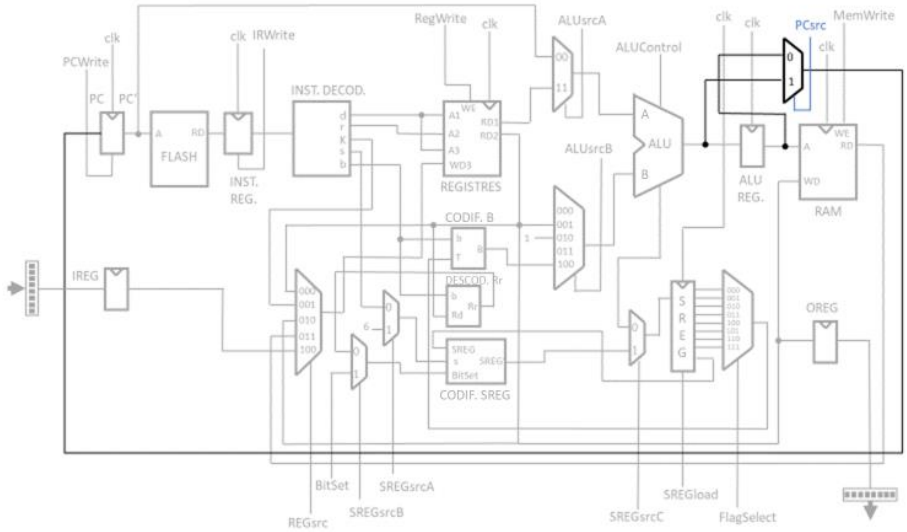


Figura 3.17 Multiplexor de selecció d'entrada de PC

L'altre format de salt incondicional que existeix és la crida a subrutina, RCALL. La particularitat d'aquest salt és que guarda l'adreça del moment de saltar per tornar-hi després d'executar el codi de la subrutina. Per fer-ho, executa un salt relatiu normal, però afegint l'escriptura de l'adreça actual a la memòria *stack*. Per fer-ho s'habilita la memòria *stack* amb el senyal *StackEN* i la seva escriptura amb el senyal *StackWR*.

Una vegada executada la subrutina, caldrà tornar a l'adreça anterior al salt. Per realitzar-ho, es carrega el valor de PC guardat a la *stack* mitjançant un nou multiplexor anomenat PC2Mux. Per obtenir l'adreça de la *stack* caldrà activar el senyal *StackEN*, i per carregar el valor al PC caldrà habilitar la senyal *PCWrite*.

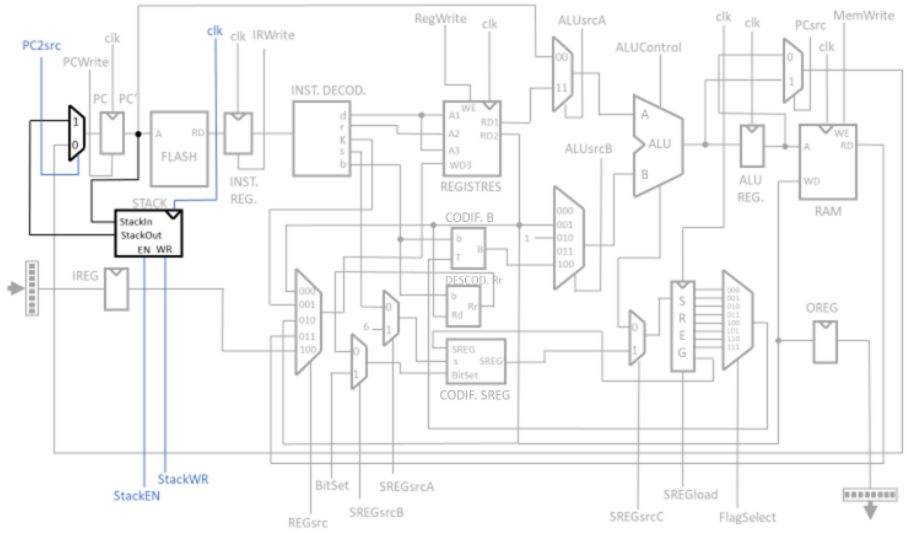


Figura 3.18 Memòria Stack i multiplexor d'entrada a PC, PC2Mux

Una vegada acabat el datapath per l'execució dels salts incondicionals, tocarà el disseny de la implementació dels salts condicionals, que representen la majoria de les instruccions de branca. Aquests salts depenen del valor que tingui el bit associat al salt condicional, que en tots els

casos és un bit del registre d'estat, *SREG*. El procediment del salt és el mateix que els incondicionals, afegint el un valor constant a l'adreça del PC. El problema és que, com que s'ha de fer la comprovació de que el bit del registre SREG té el valor correcte, el PC ja s'haurà actualitzat al següent valor, pel funcionament multicíclic del processador. Per poder executar bé la instrucció, cal agafar doncs el valor previ de PC, el valor que representa la instrucció que s'està executant en aquell moment. Per això es connecta la sortida del multiplexor *PC2MUX* a una nova entrada del multiplexor que selecciona l'operand A de la ALU.

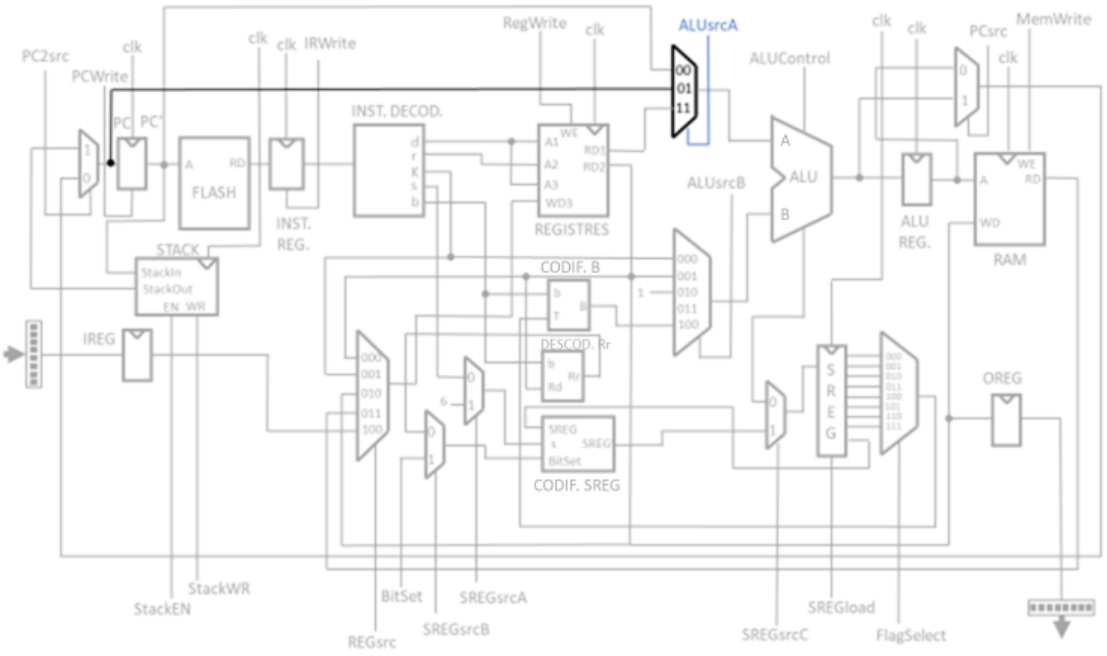


Figura 3.19 Extracció del valor de PC abans de la finalització de fetch

Finalment, els salts d'omissió son salts condicionals que salten dues posicions del PC, ometent només la instrucció següent al salt. Aquesta instrucció s'executa de forma molt similar a les instruccions de salt condicionals vistes abans: es pren el valor de PC que encara no ha estat modificat, i se li suma 2, mitjançant el multiplexor que selecciona l'operand B de la ALU.

Un problema afegit que tenen algunes instruccions de salt, específicament la instrucció de retorn del salt incondicional a subrutina i el salt incondicional, és que, com que són instruccions llargues d'executar, la següent instrucció al salt ja ha estat carregada al registre d'instruccions i està a punt per ser executada. El processador triga un cicle de rellotge en adonar-se d'aquest error, però això pot descompassar tota l'execució del programa.

Per resoldre aquest problema es generen dos estats nous, que actuen com un *Fetch* i *Decode* alternatius, que prenen la instrucció de la memòria, que canvia de valor de forma immediata, i no del registre d'instruccions, que encara te el valor de la instrucció que caldria executar si no s'hagués produït el salt. Per fer això, s'afegeix un nou multiplexor entre el registre d'instruccions i el descodificador.

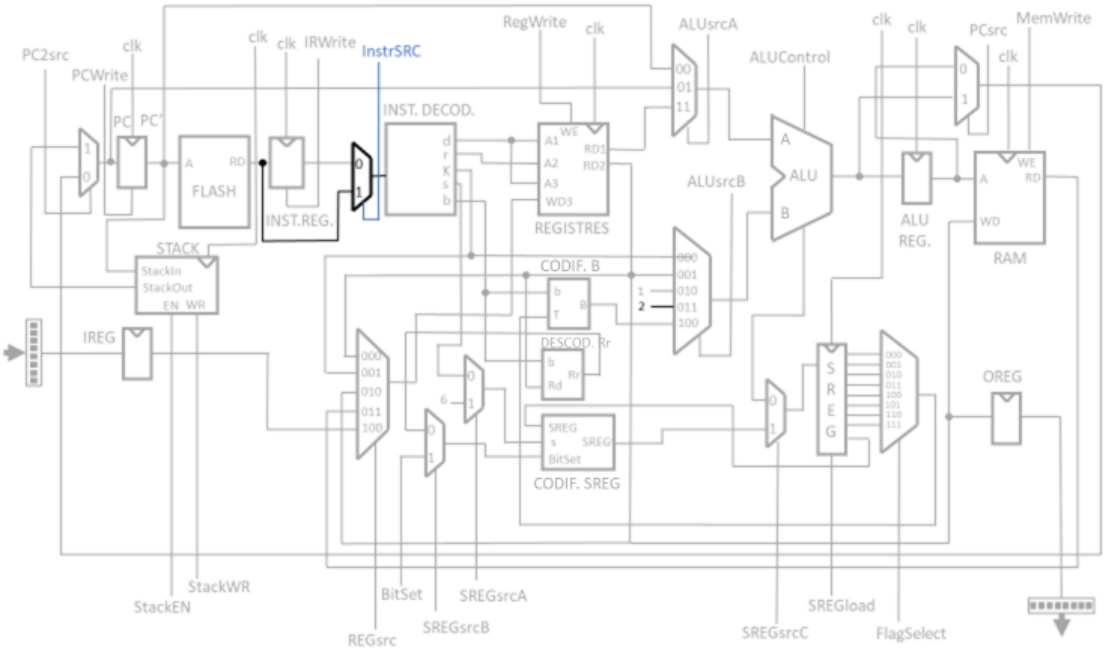


Figura 3.20 Multiplexor de selecció del codi d'instrucció

Amb això, el processador hauria de poder executar satisfactòriament totes les instruccions que inclou el set d'instruccions. De totes formes, és possible que es produís un error, per culpa d'un *glitch*, o una mala connexió. Això podria fer que el processador es bloquegés, i això podria suposar un malbaratament de potència significatiu. Per tant, en cas de que existís una paralització en l'execució del programa, la solució seria automàticament *resetejar* el sistema.

En aquest disseny del microprocessador s'inclouen 2 opcions de *reset*. La primera, el *reset* provinent de la *FPGA*, vinculat a un botó de la placa, que s'activarà de forma manual per l'usuari. La segona, que serveix com a protocol d'emergència en cas de que es produeixi un error en el processament, serà controlat pel Comptador *Watchdog*. Aquest comptador anirà incrementant el seu valor amb el cada oscil·lació del rellotge, i es posa a zero cada vegada que es comença una nova instrucció. Està programat per activar la senyal de *reset* en cas d'arribar a un valor equivalent a que hagis transcorregut 5ms, és a dir, que una instrucció trigui més de 5 ms en executar-se.

Aquests dos senyals de *reset* es combinen amb una porta lògica OR, que significa que s'activarà la senyal *reset* dels components en cas de que s'activi qualsevol dels dos senyals de *reset* del sistema. Per simplificar el diagrama i facilitar la seva lectura, no s'han inclòs les senyals de *reset* a cada component.

Amb això queda finalitzat l'esquemàtic de com s'executaran les diferents instruccions d'aquest microprocessador. Es pot veure el *datapath* complet a l'Annex B.

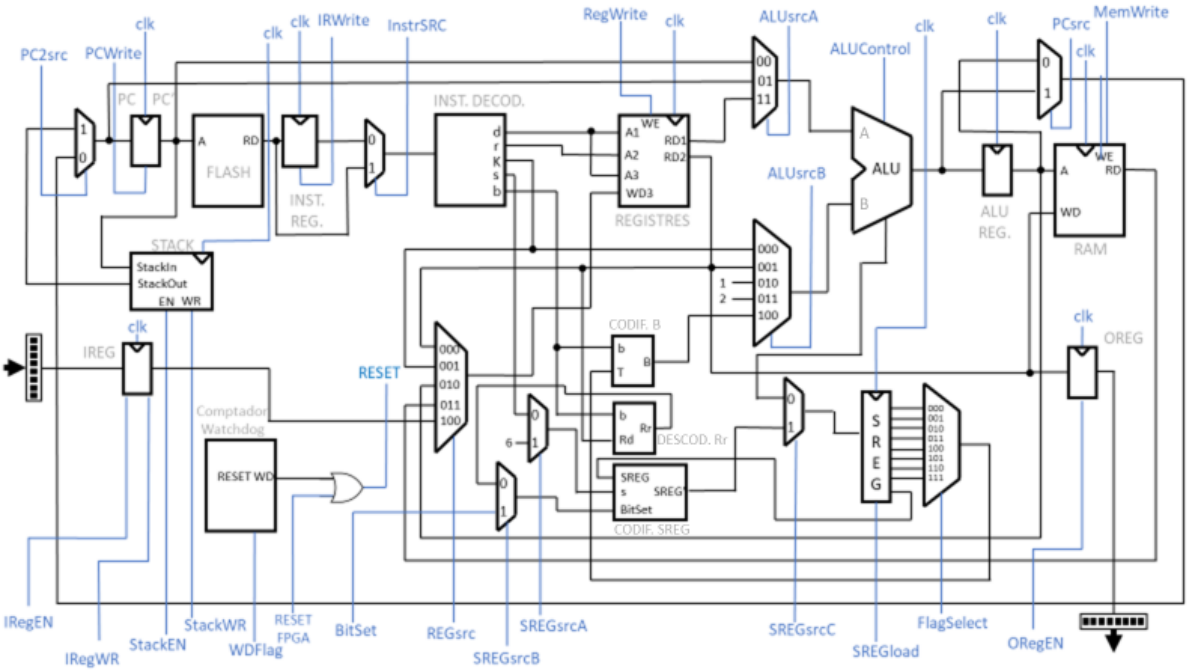


Figura 3.21 Datapath complet

3.4 Unitat de Control

Com s'ha vist en l'apartat anterior, moltes instruccions s'executen de forma casi idèntica, l'única diferència sent quines son les dades a fer servir. Per poder controlar el moviment de dades en l'execució del programa, cal controlar l'estat de les senyals de control del datapath. L'element encarregat d'això s'anomena Unitat de Control. Pel disseny d'aquest component, es divideixen les instruccions en accions encara més simples i es crea una màquina d'estats que va passant pels diferents estats segons la instrucció a executar, canviant els senyals de control dels circuits. Les senyals usades en aquesta unitat de control i una breu explicació de la seva funció es troba a la Taula 2.5 Per seguir un ordre, es començarà dissenyant els estats bàsics de qualsevol instrucció: *Fetch* i *Decode*.

Taula 3.6 Resum de senyals de control

Flag / Senyal	**Funció**
IREGEN	Habilita la transferència de dades del registre d'entrada, *IREG*.
IREGWR	Habilita l'escriptura del registre d'entrada, *IREG*.
OREGEN	Habilita l'escriptura de dades del registre de sortida, OREG.
FLAGWD	Reinicia el comptador *Watchdog*.
INSTRSRC	Selecciona l'entrada a transmetre a la sortida del multiplexor d'instruccions, *MUXINSTR.*
IRWRITE	Habilita l'escriptura del registre d'instruccions, *INSTREG.*
PCWRITE	Habilita l'escriptura del registre PC, *PCREG*.
PCSRC	Selecciona l'entrada a transmetre a la sortida del multiplexor de PC de la sortida de la ALU, *MUXPC1*.
REGWRITE	Habilita l'escriptura del banc de registres, *BANCREG.*
REGSRC	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada de dades al banc de registres, *MUXREG.*
ALUSRCA	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada a l'operand A de la ALU, MUXALUSRCA
ALUSRCB	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada a l'operand B de la ALU, *MUXALUSRCB.*
ALUCONTROL	Indica a la ALU si l'operació a realitzar depèn de la instrucció o si és una operació específica
SREGSRCA	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada al port 's' del codificador de *SREG, MUXSREGA.*
SREGSRCB	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada al port 'BitSet' del codificador de *SREG, MUXSREGB.*
SREGSRCC	Selecciona l'entrada a transmetre a la sortida del multiplexor d'entrada al registre *SREG, MUXSREGC.*
SREGLOAD	Habilita la càrrega del nou valor de *SREG* al registre d'estat, *SREG*
BITSET	Indica si el codificador de SREG, *SREGDECODE,* cal posar el flag indicat per 's' a nivell alt o baix.
MEMWRITE	Juntament amb *RAMEN, h*abilita l'escriptura de la memòria RAM
FLAGSELECT	Selecciona l'entrada a transmetre a la sortida del multiplexor de flags a la sortida de *SREG, MUXFLAGS.*
STACKEN	Habilita la lectura de dades de la memòria de *stack, STACKMEM*
STACKWR	Juntament amb *STACKEN,* habilita l'escriptura de dades a la memòria *STACK.*
RAMEN	Habilita la lectura de dades de la memòria RAM, *RAMMEM*
PC2SRC	Selecciona l'entrada a transmetre a la sortida del multiplexor de PC *MUXPC2,* entre la STACK, la ALU i el PC.

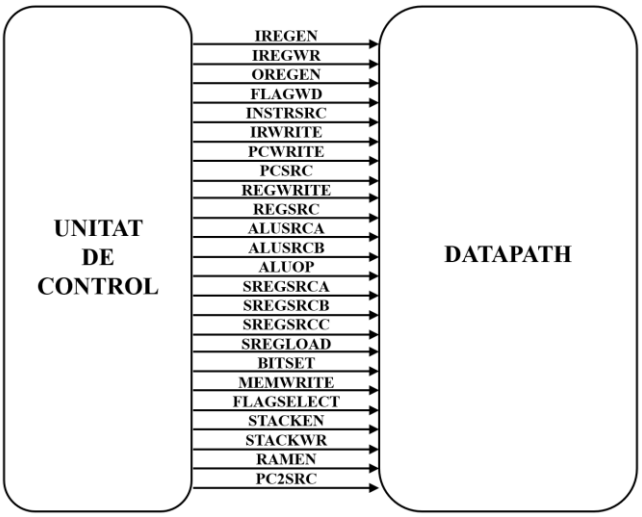


Figura 3.22 Esquema de la relació entre la Unitat de Control i el Datapath

3.4.1 Fetch i Decode

Aquestes dues accions, com s'ha explicat prèviament, carreguen la instrucció a executar de la memòria *Flash*, la descodifiquen i actualitzen el valor del PC. Per l'actualització de PC cal, primer, escollir correctament els operands (PC i '1') amb els senyals *ALUsrcA* i *ALUsrcB,* i cal indicar-li a la ALU que cal realitzar una operació de suma.

El problema que presenta la ALU és que moltes instruccions utilitzen les mateixes operacions. Les instruccions aritmètiques i lògiques, que pertanyen als formats d'instrucció 1, 2 i 3, usen la ALU per executar les operacions aritmètiques i lògiques pertinents, i per tant requereixen l'execució de diferents operacions segons quina instrucció d'aquests formats s'estigui executant. Altres, usen la ALU per operacions específiques, com sumar, realitzar una operació AND o OR, o simplement passar l'operand A a l'altre banda de la ALU.

Per poder saber quina operació s'ha d'executar amb cada instrucció, existeixen 2 senyals. La primera, *ALUControl*, indica a la ALU si s'ha d'executar una operació específica, com en aquest cas (Suma), o si l'operació depèn de la instrucció que s'executi. Dins de la ALU, hi ha un descodificador que, segons *ALUControl* i el segment 'n' del codi de la instrucció, indica quina operació cal dur a terme.

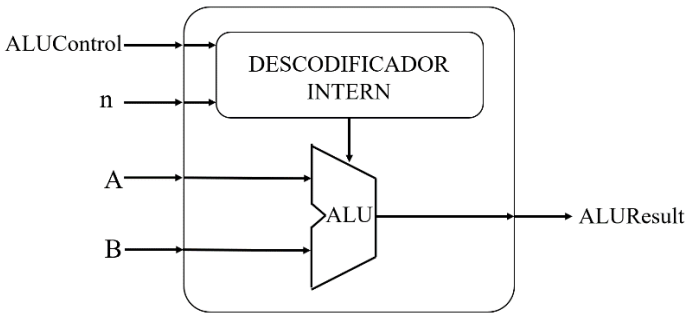


Figura 3.23 Esquema del funcionament intern de la ALU

ALUControl	Instrucció
000	Format 1
001	Format 2
010	Format 3
011	Pass A
100	AND
101	OR
110	ADD

Taula 3.7 Taula de resum de la senyal ALUControl

En aquest cas, com que cal una suma, el valor de *ALUControl* serà "110", que indica una suma.

El següent pas serà carregar el nou valor al PC. Per fer-ho, cal escollir la sortida de la ALU, abans del registre, com a entrada al multiplexor *PCMUX*, amb el senyal *PCsrc* posat a '1', i el *PC2src* posat a '0', pel segon multiplexor en el camí a PC. Per que el registre PC actualitzi el valor, cal que s'habiliti la seva escriptura, activant *PCWrite*.

Mentre es prepara el futur valor de PC, el valor actual de PC entra a la memòria d'instruccions com a adreça i a la sortida de la memòria *Flash* es pot trobar la instrucció a executar. Per que aquesta instrucció arribi al descodificador, caldrà activar l'escriptura del registre d'instruccions, amb *IRWrite*, i connectar la seva sortida al descodificador amb el *flag InstrSRC* al multiplexor *MUXInst*. Cal tenir en compte que, en els casos on es substitueix l'estat S0 per l'estat S0N, cal canviar el valor de *InstrSRC* per '1'.

Finalment, cal activar el flag WDFlag per reiniciar el comptador de Watchdog, ja que l'estat S0 és l'inici de totes les instruccions.

L'estat S1 i per tant S1N, simplement serveixen d'estat d'espera mentre es descodifica la instrucció, i per tant mantenen el valor del flag del multiplexor d'entrada al descodificador, *InstrSRC*.

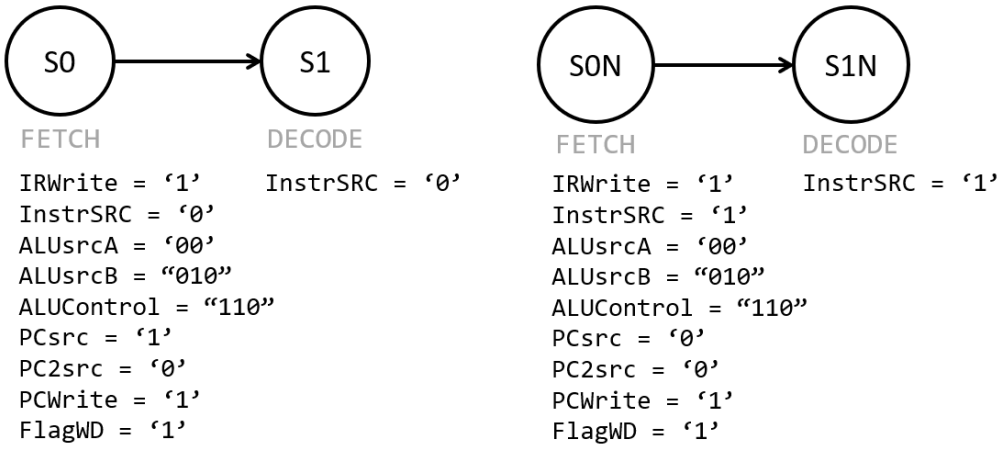


Figura 3.24 Estats S0 i S1 i les seves variants, S0N i S1N

3.4.2 Instruccions de transferència: ST i LD

Les següents instruccions a analitzar en clau de la Unitat de Control seran les de càrrega i escriptura de la RAM. Per escriure dades a la memòria RAM, després de la descodificació la màquina d'estats anirà a l'estat S2, on es passarà la sortida RD1 per la ALU fins l'entrada d'adreces de la RAM. Per fer això, caldrà que, amb el *flag ALUsrcA* es seleccioni la sortida del banc de registres com a operand A de la ALU. Amb *ALUControl* = "011" s'indica a la ALU que executi l'operació *PASS A*, que copia l'operand A a la sortida de la ALU. Com en totes les operacions de la ALU, es carregarà a *SREG* el valor actualitzat dels *flags* de l'operació. Per això, cal activar la càrrega de SREG, amb el *flag SREGLoad*.

Per carregar les dades de la RAM, a l'estat 3 s'habilita la lectura de la memòria RAM, que al final del cicle de rellotge treu per la seva sortida les dades guardades a l'adreça indicada. Una

vegada obtingudes les dades, es carregaran al banc de registres usant els *flags REGsrc* i *REGWrite*, el primer per escollir la sortida de la RAM com a entrada de dades del banc de registres i la segona per habilitar l'escriptura del registre indicat per l'entrada A3 del banc de registres. Una vegada finalitzada aquesta acció, la màquina torna a l'estat S0, per iniciar l'execució de la següent instrucció.

Per guardar dades a la RAM, el procediment és similar. Primer, es passa per l'estat S2, i es porta l'adreça guardada al registre ZLo a l'entrada d'adreces de la memòria RAM. Després, a l'estat S5, s'habilita la RAM i també la seva escriptura, amb *MEMWrite*. Una vegada guardades les dades, es torna a S0.

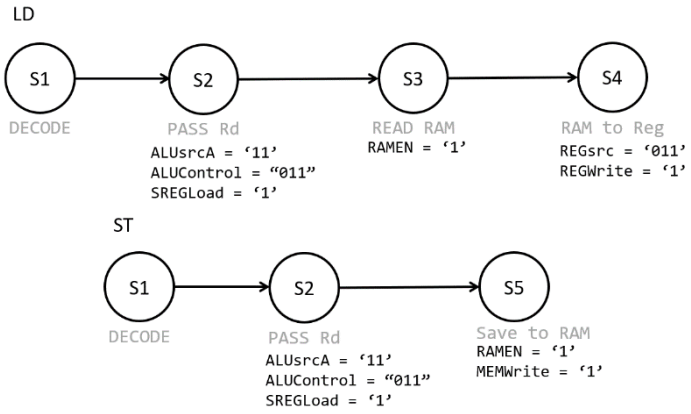


Figura 3.25 Estats de la Unitat de Control per l'execució de la instrucció LD (superior) i ST (inferior)

3.4.3 Format 1: Instruccions aritmètiques entre dos registres

A continuació, es discutiran els estats necessaris per l'execució de les instruccions del format 1, les instruccions amb adreçament directe entre dos registres. La majoria de les instruccions del format 1 són instruccions aritmètiques i lògiques. Aquestes instruccions es redueixen bàsicament a una operació entre 2 registres i la càrrega del resultat al registre de destí.

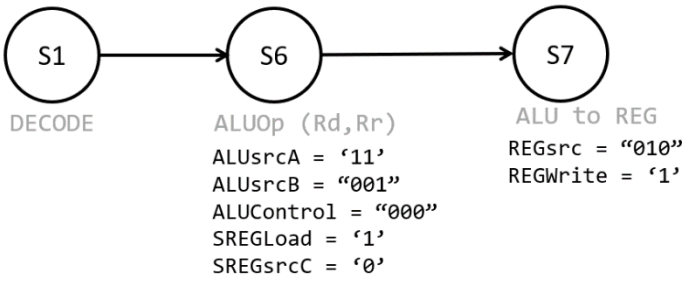


Figura 3.26 Estats de la Unitat de Control per l'execució de les instruccions del Format 1

Per tant, en cas de que després de l'estat S1 resulti que s'ha d'executar una d'aquestes instruccions, s'anirà a l'estat S6, on s'executarà l'operació entre els registres Rd, seleccionat com a operand A pel *flag ALUsrcA* = "11", i Rr, seleccionat com a operand B pel *flag ALUsrcB*. *ALUControl* tindrà el valor "000", que indica al descodificador intern de la ALU que l'operació a executar dependrà del segment 'n' del codi de la instrucció. El resum de les operacions a

realitzar en funció de la instrucció es pot trobar a la Taula 3.8. Com sempre que es fa servir la ALU, cal carregar els nous valors dels *flags* a *SREG*, així que cal primer escollir la sortida de *SREG* de la ALU com a font dels flags actualitzats, amb el multiplexor controlat per *SREGsrcC*. Després, cal activar l'escriptura de *SREG* amb *SREGLoad*.

Taula 3.8 Resum dels Opcodes de les instruccions del Format 1 i el valor de ALUOp per executar-les

nnnn	Instrucció	Operació	ALUOp
0001	CPC	Resta amb Carry	10001
0010	SBC	Resta amb Carry	10001
0011	ADD	Suma	00000
0100	CPSE	Resta	00001
0101	CP	Resta	00001
0110	SUB	Resta	00001
0111	ADC	Suma amb Carry	10000
1000	AND	AND	00010
1001	EOR	XOR	00100
1010	OR	OR	00011

Una vegada realitzada l'operació pertinent, el resultat de la ALU es guardarà al banc de registres. Aquesta operació es controlarà amb l'estat S7, que seleccionarà la sortida de la ALU com a origen de les dades mitjançant *REGsrc*, i activarà l'escriptura del banc de registres amb *REGWrite*. Una vegada finalitzada l'escriptura de les dades al banc de registres, la màquina d'estats retornarà a l'estat S0.

Com s'ha dit abans en aquesta secció, aquesta sèrie d'estats controla la majoria de les instruccions del Format 1. No obstant, dues instruccions queden fora de l'esquema de la Figura 3.27: MOV i CSPE[1].

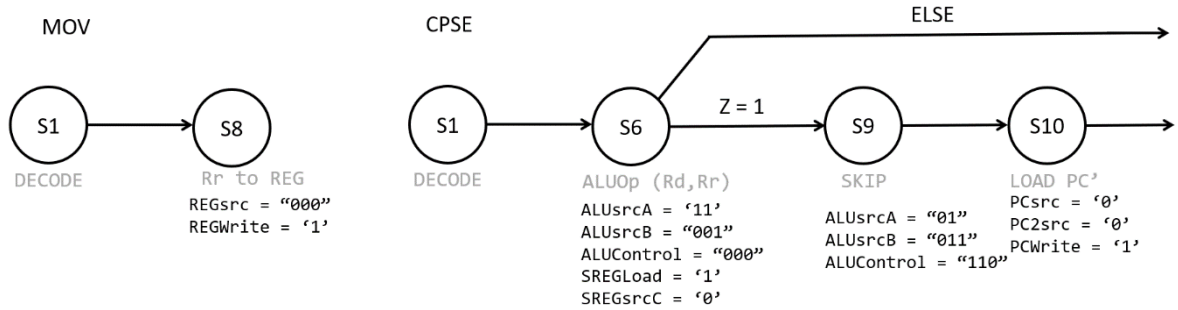


Figura 3.27 Estats de la Unitat de Control per l'execució de les instruccions del Format 1, MOV i CSPE

La instrucció MOV és bastant simple. A la sortida de l'estat S1 es troba S8, que porta la sortida del banc de registres RD2 a l'entrada de dades del banc de registres, WD3, mitjançant el

[1] MOV: Instrucció que mou el contingut d'un registre del banc de registres a un altre adreça del banc
CSPE: Instrucció que compara el valor de dos registres i, si tenen el mateix valor, fa un salt d'omissió

multiplexor controlat per *REGsrc*. Escriu les dades de la sortida RD2 al registre marcat per A3 amb el *flag REGWrite*. Una vegada ha acabat, sorna a S0.

CPSE és una instrucció més complicada, ja que és una operació aritmètica seguida d'un salt condicional. Primer, fa servir l'estat S6 per calcular la resta entre Rd i Rr. Després, depenent de l'estat del flag de *SREG* Z, que indica si l'operació executada ha resultat en un zero, fa un salt d'omissió o no. En cas de que Z no estigui activat, no es produirà el salt, ja que els valors de Rd i Rr no són iguals. En cas de que si que ho siguin, i Z = '1', es produirà el salt.

L'estat S9, mitjançant *ALUsrcA, ALUsrcB* i *ALUControl*, seleccionarà el valor previ a l'actualització de PC com a operand A, '2' com a operand B i els sumarà. EL resultat es carregarà al PC per l'estat S10, que activarà els *flags PCsrc* i *PC2src* per portar el nou valor al PC, i *PCwrite* per habilitar l'escriptura d'aquest. SI es produeix el salt, caldrà que la màquina d'estats implementi l'estat S0N, per que la següent instrucció s'executi correctament.

3.4.4 Format 2: Instruccions aritmètiques sobre un registre

Les instruccions del Format 2 són instruccions que, igual que les del Format 1, executen operacions aritmètiques i lògiques, però en aquest cas només s'apliquen a un registre. Aquest format d'instruccions, tot i incloure una varietat d'operacions, es pot controlar sencer amb només un camí d'estats. Primer, l'estat S11, de forma molt similar a l'estat S6, executarà l'operació pertinent, marcada pel *flag ALUControl*, que quan marca "001" és per indicar que, com passava en les operacions del Format 1, cal mirar 'n' per saber quina instrucció cal executar. El resum de les operacions a realitzar en funció de la instrucció es pot trobar a la Taula 3.9. Després es carrega el resultat al banc de registres mitjançant l'estat S7, i es torna a S0.

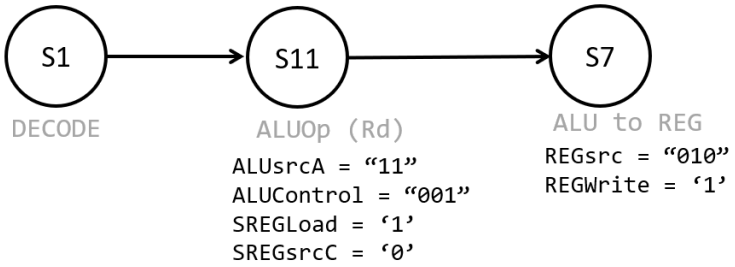


Figura 3.28 Estats de la Unitat de Control per l'execució de les instruccions del Format 2

Taula 3.9 Resum dels Opcodes del Format 2 i el valor de ALUOp necessari per executar-les

nnnn	Instrucció	Operació	ALUOp
0000	COM	Complement	00101
0001	NEG	Complement a 2	00110
0010	SWAP	Intercanvi de *nibbles*	01110
0011	INC	Suma 1	00111
0101	ASR	Shift Aritmètic a la Dreta	01011
0110	LSR	Shift Lògic a la Dreta	01001
0111	ROR	Rotació a la Dreta	01010
1010	DEC	Resta 1	01000

3.4.5 Format 3: Instruccions aritmètiques entre un registre i una constant

El Format d'Instruccions 3, que acumula les instruccions aritmètiques entre un registre i un valor constant, també es pot controlar amb un sol camí d'estats, molt similar al que s'ha explicat prèviament.

L'estat S12, de la mateixa forma que els estats S6 i S11, executa l'operació marcada pel *flag ALUControl* i 'n', la diferencia entre aquest estat i els anteriorment mencionats sent que aquest selecciona la constant K de la sortida del descodificador com a operand B. Després, mitjançant l'estat S7, es guarda el resultat al registre de destí i passa a S0, per començar la següent instrucció.

La instrucció del Format 10, CPI, que compara el valor d'un registre amb una constant, segueix exactament el mateix procediment que la instrucció SUBI.

El resum de les operacions a executar segons la instrucció es troba a la Taula 3.10.

Taula 3.10 Resum dels Opcodes del Format 3 i el valor de ALUOp necessari per executar-les

nn	Instrucció	Operació	ALUOp
00	SBCI	Resta amb Carry	10001
01	SUBI	Resta	00001
10	ORI	OR	00010
11	ANDI	AND	00011

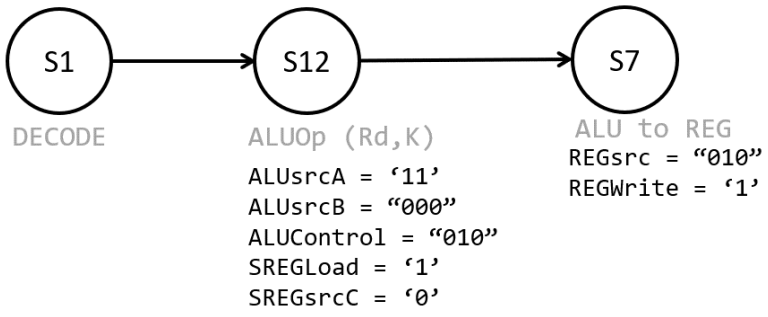


Figura 3.29 Estats de la Unitat de Control per l'execució de les instruccions del Format 3

3.4.6 Format 5: Instruccions de bit i de salt respecte un registre

El Format 4 ja s'ha explicat, sent les instruccions que li pertanyen les de ST i LD, de càrrega i escriptura de la RAM. El Format 5 d'instruccions és potser el més divers, ja que conté 4 instruccions, i cada una s'executa amb un camí d'estats diferent. Les instruccions que s'inclouen en aquest format són de transferència de dades i de salt, però el que tenen en comú és que actuen fent referència a un bit específic d'un registre del banc de registres. En aquest format d'instruccions es troben les instruccions de guardar o carregar el valor d'un bit d'un registre al *flag* T del registre *SREG*, i les instruccions de salt condicional que usen el valor d'un bit del registre com a condició de salt.

Per començar l'anàlisi del control de les instruccions d'aquest format, s'estudiarà la instrucció BLD, que escriu el contingut del *flag* T a un registre del banc de registres.

Aquesta instrucció es bifurca depenent del valor de T. Si després de l'estat S1 el flag T val '0', l'estat S15 executa una operació OR entre el vector B i el contingut del registre Rd. Això ho fa seleccionant RD1 com a operand A, amb *ALUsrcA*, i B com a operand B, aplicant el valor "100" al *flag ALUsrcB. ALUControl* valdrà "101", indicant que s'ha d'executar una operació OR. Si, en canvi, el valor de T és '1', s'aplicaran els valors dels flags de l'estat S16, que és idèntic en funcionament a S15, excepte que el seu valor de *ALUControl* és "100", que indica que s'ha d'executar l'operació AND entre Rd i B.

Una vegada executada l'operació, sigui AND o OR, el resultat es guarda al banc de registres amb S7 i es reinicia el cicle tornant a S0.

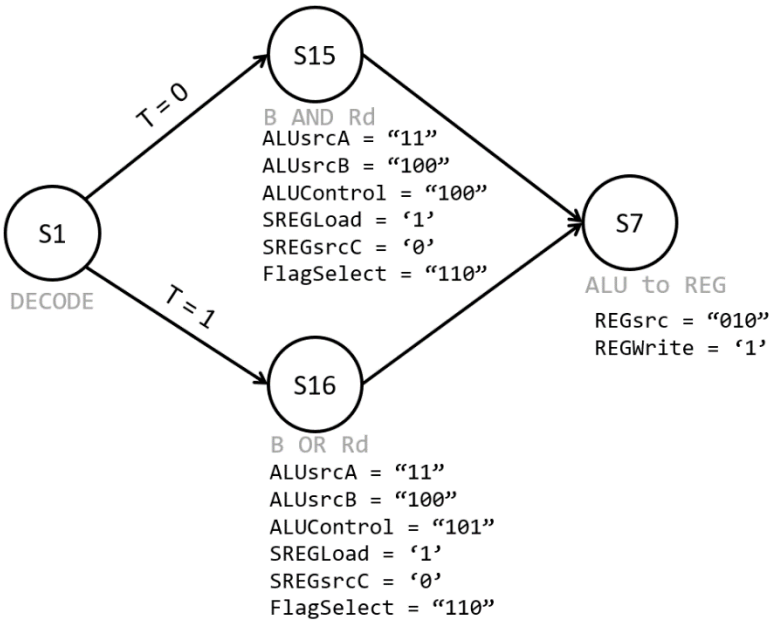


Figura 3.30 Estats de la Unitat de Control per l'execució de la instrucció BLD

La següent instrucció del Format 5 és BST, que guarda un valor d'un bit del banc de registres al *flag* T. Amb l'estat S17, que no activa cap element, es dona temps a que s'extregui el valor del bit a guardar del registre seleccionat. Quan s'ha assolit, l'estat S18 selecciona '6' com a entrada 's' del codificador de SREG amb el *flag SREGsrcA*, porta el valor del bit a l'entrada *BitSet* amb el *flag SREGsrcB* i porta el SREG' resultant a l'entrada de *SREG* amb *SREGsrcC*. Amb els camins de la informació preparats, activa l'escriptura de *SREG* i carrega el vector d'estat amb el nou valor de T, mitjançant *SREGLoad*. Després, com en tots els casos, torna a l'estat S0.

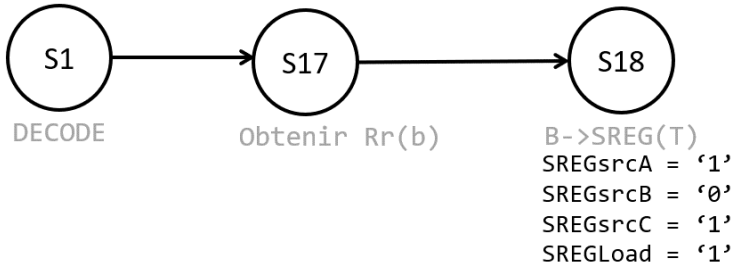


Figura 3.31 Estats de la Unitat de Control per l'execució de la instrucció BST

Els següents dos casos, SBRS i SBRC, són casi idèntics. SBRS fa un salt d'omissió condicional, ja que només el fa si el bit, especificat per 'b', del registre, especificat per 'd', val '1'. SBRC té un funcionament complementari: fa el salt quan el bit 'b' del registre 'd' val '0'. Ambdues instruccions fan servir l'estat S17 per obtenir el valor del bit i, en cas de que es compleixi la condició de salt, usen els estats S9 i S10 per realitzar el *skip*. Finalment, tornen a S0.

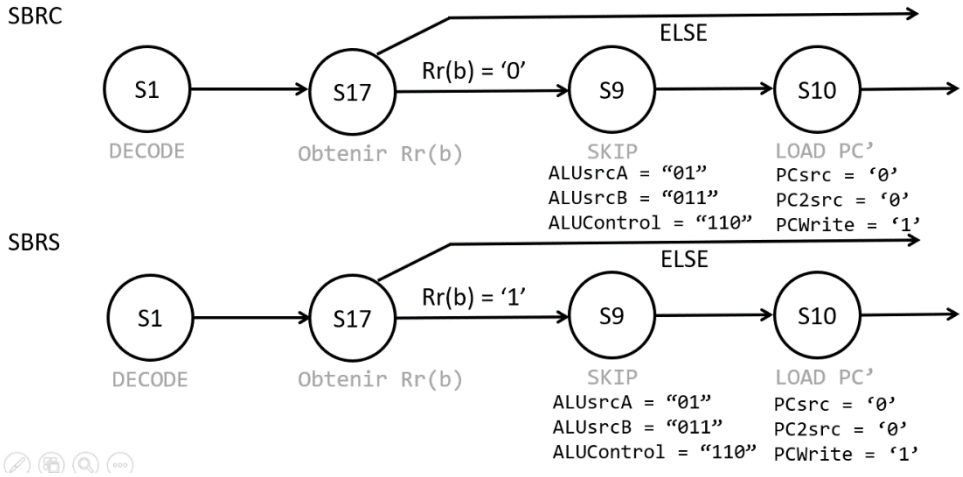


Figura 3.32 Estats de la Unitat de Control per l'execució de les instruccions SBRC (superior) i SBRS (inferior)

3.4.7 Format 6: Operacions de transferència: IN i OUT

Les operacions del format d'instrucció 6 són operacions que importen i exporten dades mitjançant els ports *IREG* i *OREG*, respectivament. L'operació OUT, que treu dades del banc de registres a l'exterior pel port *OREG*, és la més senzilla de les dues. Només cal, amb l'estat S27, activar el flag *OREGEN*, que escriurà el valor a treure a *OREG*. Ara, a la sortida, estarà el mateix valor fins que el valor de *OREG* es modifiqui, per *reset* o per la càrrega d'un nou valor.

La instrucció IN és lleugerament més llarga: Primer cal carregar el valor que entra al port d'entrada a *IREG*. Per això, a l'estat S28, s'activen els *flags IREGEN* i *IREGWR*. Amb el valor carregat al registre d'entrada, caldrà mourà les dades al banc de registres. Per fer-ho, S19 marca

els valors de *REGsrc* i *REGWrite* com "100" i '1', respectivament. També manté habilitat IREGEN, per mantenir el valor de sortida del Registre mentre es traspassa al banc de registres. Finalment, en ambdós casos, es passa a l'estat S0 i comença l'execució de la següent instrucció.

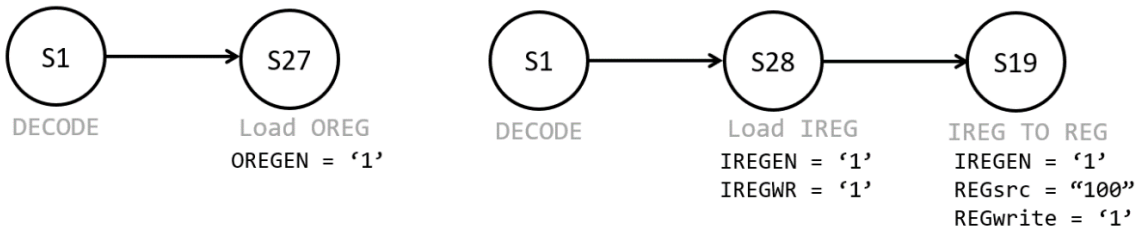


Figura 3.33 Estats de la Unitat de Control per l'execució de les instruccions OUT (esquerra) i IN (dreta)

3.4.8 Format 8: Instruccions de bit respecte el registre SREG

El format 8 d'instruccions inclou totes les operacions que posen a '1' o a '0' bits específics del registre d'estat, *SREG*. Tot i incloure 16 instruccions, la seva execució es controla de forma igual. L'estat S20 carrega el *flag*, determinat per la sortida 'sss' del descodificador, que amb *SREGsrcA* porta al port 's' del codificador de *SREG*. Amb *SREGsrcB* selecciona el *flag BitSet* com a entrada del codificador de *SREG* que marca si cal posar el bit a '1' o a '0'. Finalment, amb *SREGsrcC* i *SREGLoad*, carrega el nou valor SREG' al registre d'estat.

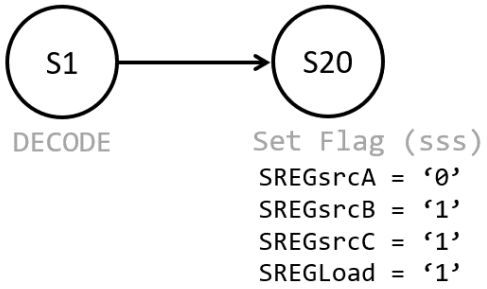


Figura 3.34 Estats de la Unitat de Control per l'execució de les instruccions del Format 8

3.4.9 Format 9: Instrucció de transferència. LDI

El Format 9 només inclou la instrucció LDI, que carrega una constant (K) a un registre de la meitat superior (de R16 a R31) del banc de registres. Per fer-ho, amb l'estat S21 connecta la sortida K del descodificador amb l'entrada WD3 del banc de registres posant *REGsrc* a "001" i activant l'escriptura del banc de registres amb *REGWrite*. Després, com en totes les instruccions, torna a l'estat S0.

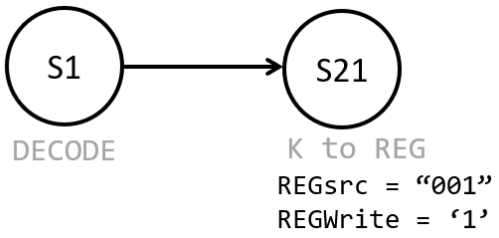


Figura 3.35 Estats de la Unitat de Control per l'execució de les instruccions del Format 9

3.4.10 Format 12: Instruccions de salt incondicional. RJMP i RCALL

El Format 12 és el que inclou els salts incondicionals: RJMP i RCALL. RJMP s'executa amb els estats S22 i S10. S22 s'encarrega de la suma de PC i K seleccionant adequadament els valors de *ALUsrcA*, *ALUsrcB* i *ALUControl*, com s'ha explicat prèviament en aquest apartat. Amb S10 es carrega el nou valor de PC al comptador del programa.

L'estat S23, en l'execució de RCALL, permet l'execució de la suma de PC i K de S22, però a més, activant *StackEN* i *StackWR*, activa l'escriptura del *Stack* i guarda el valor de PC abans del salt. S26 és un estat d'espera abans d'iniciar la següent instrucció.

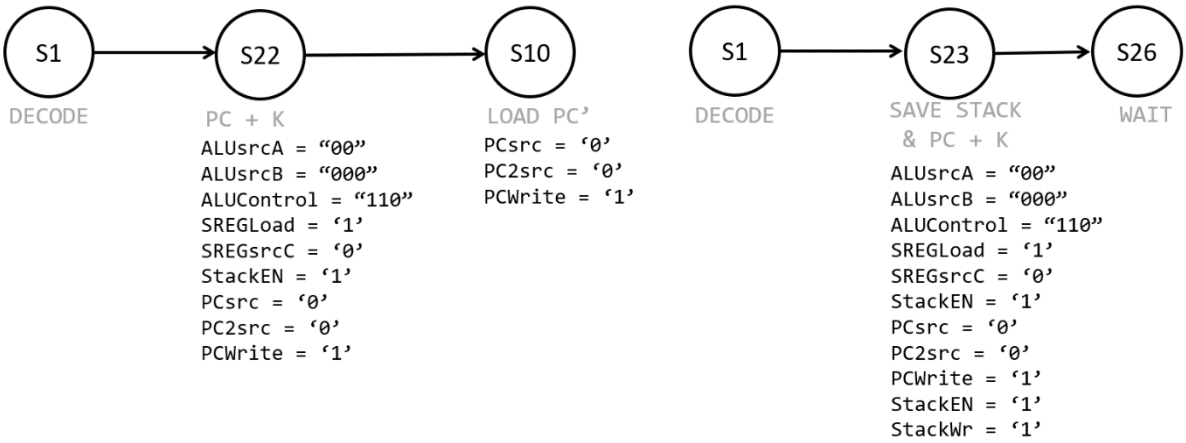


Figura 3.36 Estats de la Unitat de Control per l'execució de les instruccions del Format 12 RJMP (esquerra) i RCALL (dreta)

3.4.11 Format 13: Instrucció de retorn del salt. RET

RETI, la instrucció del Format 13 que retorna el PC al valor que tenia abans del salt RCALL, s'executa amb dos estats: S24 i S13. S24, amb *StackEN* activat i *StackWR* desactivat, permet la lectura de l'adreça superior de la memòria *stack*. S13, d'altre banda, carrega aquest valor a PC d'una forma molt similar a la que ho fa S10, només que en aquest cas *PC2src* és '1', així que a l'entrada de PC trobem la sortida *StackOut*.

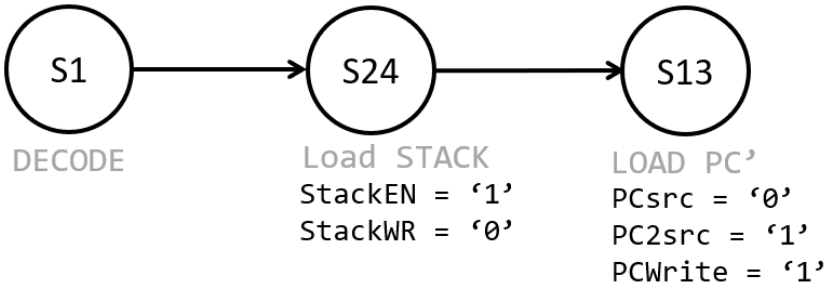


Figura 3.37 Estats de la Unitat de Control per l'execució de la instrucció RET

3.4.12 Format 14: Instruccions de salt condicional

Les instruccions del Format 14, de forma similar a les del Format 8, són moltes però s'executen totes de forma igual, a nivell de la Unitat de Control. Si 'n' és '0' significa que la condició del salt és positiva, és a dir, que el bit de la condició ha d'estar posat a '1'. En canvi, si 'n' val '1' només es produirà el salt quan el bit de la condició estigui a nivell baix.

En qualsevol cas, si les condicions es compleixen, la màquina d'estats saltarà a l'estat S14. Aquest estat fa la mateixa funció que l'estat S22, però en aquest cas agafa com a operand A el valor de PC anterior a l'actualització, donant el valor "01" al *flag ALUsrcA*. La raó d'això és que durant la comprovació de les condicions de salt ja s'haurà actualitzat el valor de PC i això fa que el salt no es produeixi correctament. Després del salt, amb S10, es guarda el resultat del salt a PC, i es torna a S0 per iniciar la següent instrucció.

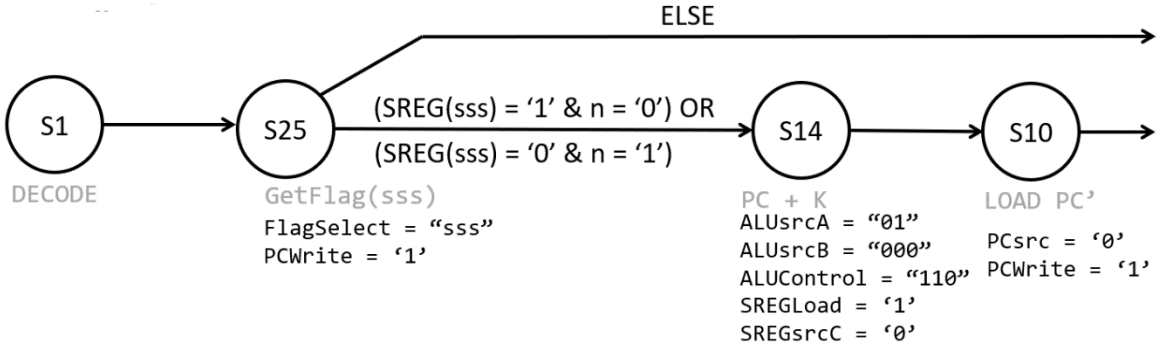


Figura 3.38 Estats de la Unitat de Control per l'execució de les instruccions del Format 14

3.4.13 Formats 15 i 16: SLEEP i NOP

Finalment, els Formats 15 i 16 només tenen una instrucció a cada un, SLEEP i NOP. La instrucció SLEEP, segons la documentació d'aquesta arquitectura, redueix les funcions del microprocessador per estalviar energia. Quan s'aplica en un microcontrolador, només queden habilitats alguns perifèrics i el rellotge intern. Per sortir d'aquest estat, cal un *reset* manual, un *reset* provinent del comptador *Watchdog* o una interrupció externa. En aquest cas, com que no hi ha interrupcions i tampoc perifèrics, el processador romandrà *dormint* fins que rebi una senyal de *reset*. Tots els senyals d'habilitació de chip i d'escriptura es posaran a 0, així evitant els canvis de valor interns i reduint el consum energètic que generen.

La instrucció NOP, que no executa cap instrucció, té un funcionament similar. Posa tots els *flags* a nivell baix, per no executar cap operació. Les dues operacions tenen un comportament similar, la diferencia es troba en la forma en que es reactiva el processador, en la instrucció SLEEP és gràcies a un *reset*, i a la instrucció NOP passa de forma natural passat un cicle de rellotge. Per ambdues instruccions, doncs es fa servir el mateix estat, S26, que posa tots els *flags* a nivell baix.

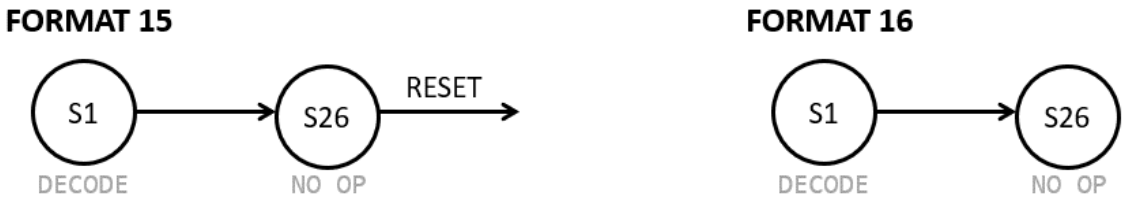


Figura 3.39 Estats de la Unitat de Control per l'execució de les instruccions SLEEP, del Format 15 (esquerra) i NOP, del Format 16 (dreta)

3.5 Programació en VHDL

Després del disseny del *Datapath* i de la Unitat de Control, s'ha de plasmar aquest disseny en VHDL. Per fer-ho, donat que el sistema és composat per un gran nombre de components, s'ha dissenyat de forma estructurada. S'ha començat escrivint el codi VHDL de cada component per separat, després s'han ajuntat els components en el *Datapath*. A part, s'ha plasmat el funcionament de la Unitat de Control en un altre arxiu VHDL, i després s'ha escrit també el codi del Comptador *Watchdog*. Finalment, s'han unit aquests 3 elements en un general, anomenat TOP, que fa totes les funcions del microprocessador.

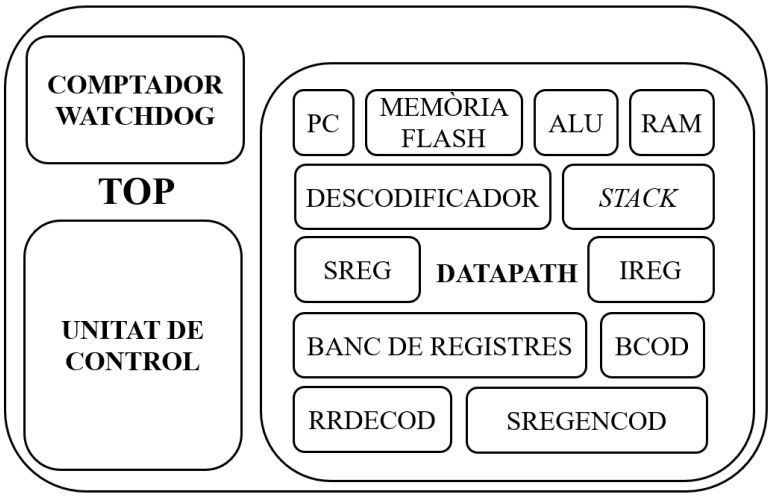


Figura 3.40 Esquema de l'organització del programa de VHDL del processador

El recull de codis VHDL del sistema es pot trobar a l'Annex D.

Capítol 4. Simulació del microprocessador

En aquest apartat es demostrarà el funcionament del microprocessador mitjançant simulacions fetes amb el programa Vivado. Les simulacions es faran concatenant instruccions i veient com reaccionen els diferents components i com varien els valors dels registres.

4.1. *Fetch, Decode* i Carregar un valor a un registre (LDI)

Per començar, es carregarà un valor a un registre, mitjançant la instrucció LDI. La instrucció LDI, del Format 9, permet carregar una constant K a un registre de la meitat superior del banc de registres. Es decideix carregar el valor decimal 10 al registre R16. Sent el format de la instrucció "1110 KKKK dddd KKKK", el codi resultant serà, en codi assemblador, en binari i en hexadecimal:

```
LDI R16, 10 ;   "1110 0000 0000 1010";   E00A
```

Com totes les instruccions, partirà de l'estat S0, que s'encarregarà de realitzar el *fetch*. Com es pot veure a la Figura 4.1, la Unitat de Control, o controlador, el component encarregat d'entregar les senyals adequades segons l'estat en el que es troba el processador. Aquestes senyals venen donades pel vector de 33 bits *VEC_CONTROL*. Per l'estat S0 aquest vector pren el valor "0 0010 1110 0000 0010 1100 0000 0000 0000", posant els valors als *Flags* que s'indiquen a la Figura 3.24 per l'estat S0.

Amb aquests flags es duen a terme totes les operacions necessàries a S0. Primer, *PCN*, la sortida del PC, imposa '0' com adreça a la memòria *Flash*, que per la seva sortida (*SPO*) treu el codi de la instrucció a executar: "E00A".

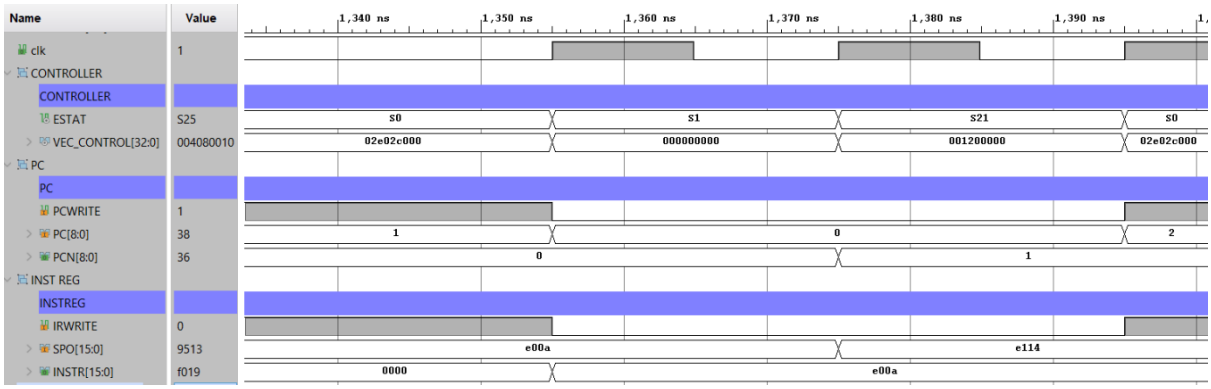


Figura 4.1 Simulació de les operacions de Fetch

Amb la instrucció preparada per la descodificació, s'executa el càlcul del següent valor de PC, afegint '1' al valor actual. Com es veu a la Figura 4.2, *ALUA*, l'operand A de la ALU, val '0' i *ALUB*, l'operand B, val 1. El senyal *ALUControl* val "110", indicant que és una suma sense carry, i per tant *ALUOp*, a través del descodificador intern de la ALU, val "00000".

El resultat, que surt del port *ALUOUT* de la ALU, i va al multiplexor *MUXPC* per l'entrada *PCALU* i és seleccionat pel senyal *PCSRC*, que val '1', com es pot veure a la Figura 4.2. A través de la sortida, *PCMUXOUT*, arribant a *MUXPC2* per l'entrada *PC1M* i sortint per *PC2MUXOUT* gràcies al senyal *PC2SRC*. Finalment, arriba a l'entrada de del comptador de programa on entra la nova adreça, que s'anomena *PC*.

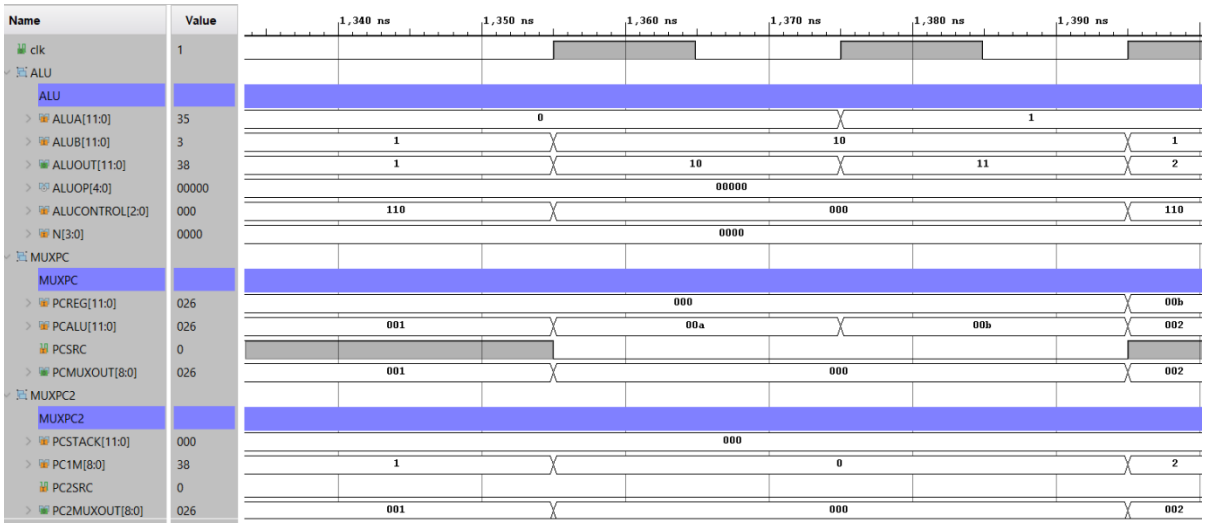


Figura 4.2 Simulació de les operacions de càrrega del nou valor de PC

A l'estat S1, passen dues coses: El nou valor de PC es grava al registre, canviant el valor de *PCN*, com es pot veure a la Figura 4.1, i es descodifica la instrucció. Aquesta descodificació es fa gràcies al descodificador d'instruccions, *DECODINST*. Com es veu a la Figura 4.3, durant l'estat S1, el descodificador rep el codi d'instrucció, E00A, i n'extreu la informació que porta dins. Primer, identifica la instrucció com una instrucció de Format 9 (a la figura mostra un 8, perquè en comptes de numerar-les 1-16 les numera 0-15, així que sempre caldrà incrementar el valor de la sortida de *FORMAT* en '1' per interpretar-ho correctament), i per tant troba els valors de 'd' i 'K', que, com s'ha dit abans, son 16 i 10. Les altres sortides, com que no estan incloses en aquest format, es mantenen a '0'.

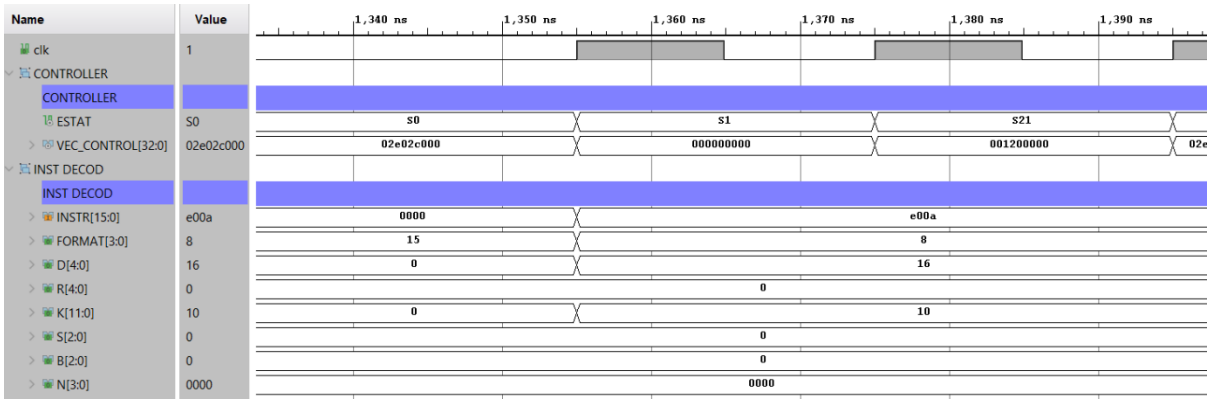


Figura 4.3 Simulació de les operacions de descodificació de la instrucció LDI

Una vegada s'ha descodificat la instrucció, es procedirà a la seva execució. En aquest cas, com que LDI és l'única instrucció del Format 9, no caldrà observar el valor de 'n' per decidir a quin estat anar, sinó que el controlador passarà automàticament a l'estat S21, que carrega K al banc de registres.

Es pot veure, a la Figura 4.4, que el valor del vector de sortida de la Unitat de Control, *VEC_CONTROL*, ha canviat de valor, escollint els valor adequat de *REGSRCIN*, "001", per que la sortida *K* de *DECODINST* arribi a l'entrada *WD3* del banc de registres. Com es pot veure a l'esquema del *Datapath* a l'Annex B, les entrades A1 i A3 estan connectades a la sortida 'd'

del descodificador, i això es reflexa en la simulació en que sempre tenen el mateix valor, 16 en aquest cas, perquè és el registre que es vol escriure. Per poder gravar el valor de K al registre R16, caldrà activar l'entrada *WRITEEN* del banc de registres, que es fa amb el *flag REGWrite* de la Unitat de Control. Amb tot llest, es veu a la figura com, al finalitzar l'estat S21, el valor del registre R16 (*REG16*) canvia a tenir el valor decimal 10. El següent estat és S0, que repara l'execució de la instrucció guardada a l'adreça 1 de la memòria *Flash*.

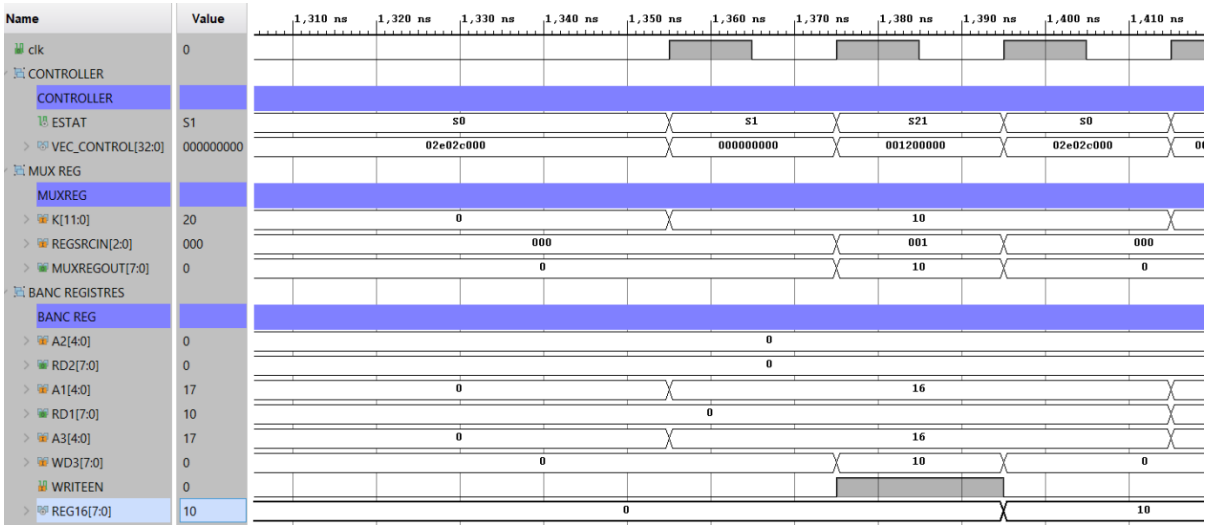


Figura 4.4 Simulació de les operacions de càrrega del resultat al banc de registres

4.2. Operacions Aritmètiques i Lògiques

4.2.1 ADD

Per seguir provant el funcionament del microprocessador, ara s'executaran unes operacions aritmètiques i lògiques entre 2 registres, R16 i R17. S'ha carregat el valor decimal 20 al registre R17, usant la instrucció LDI, i ara es procedirà a sumar els dos valors, sense carry, i guardar el resultat a R16. Aquesta instrucció, ADD, pertany al Format 1 d'instruccions, les instruccions del qual segueixen l'estructura "00nn nnrd dddd rrrr", on 'd' i 'r' són les adreces dels registres operands i 'n' indica la operació a realitzar. Sabent tots els components de la instrucció, es pot afirmar que la instrucció es formularà de la següent forma:

```
ADD r16, r17 ;  "0000 1111 0000 0001";   0x0F01
```

El primer pas, com sempre, serà la descodificació de la instrucció. A la Figura 4.5 es veu com a l'estat S1 el descodificador ha interpretat l'operació correctament com una operació de Format 1, que el registre de destí, 'd', és el registre R16, i que el registre de l'altre operand, és el registre R17. Mirant VEC_*CONTROL*, s'observa que *ALUCONTROL* marca "000", que indica que s'ha de mirar l'entrada 'n' quan toqui fer l'operació amb la ALU. El descodificador també extreu el valor de 'n' del codi de instrucció, i en aquest cas, com s'indica a la figura, val "0011", que indicarà que l'operació desitjada és la suma sense carry.

A l'estat S6 es produeix la suma dels dos valors, que surten del banc de registres per les sortides RD1 i RD2, i entren a la ALU com a operand A i B, respectivament, com es pot veure a la Figura 4.6. També es pot veure com, amb *ALUCONTROL* sent "000" i *N* sent "0011" (3 en decimal) el codi d'operació de la ALU, *ALUOP*, és "00000", que indica que s'ha de fer una suma sense *carry*. Finalment, el resultat surt pel port *ALUOUT*, i es carrega el valor actualitzat

del registre d'estat, *SREG*. Desgraciadament, aquesta operació no genera cap canvi a *SREG*, així que no es mostrarà aquest pas encara.

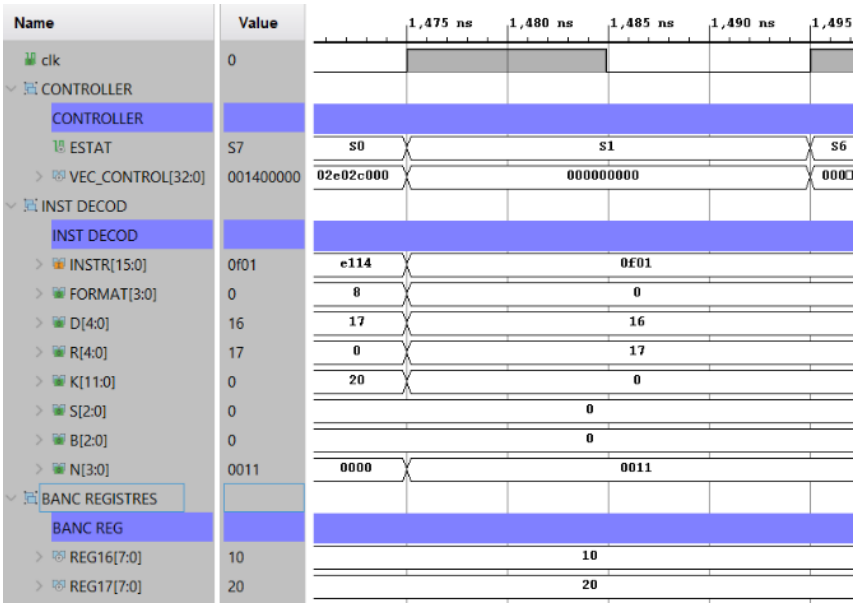


Figura 4.5 Simulació de les operacions de descodificació de la instrucció ADD

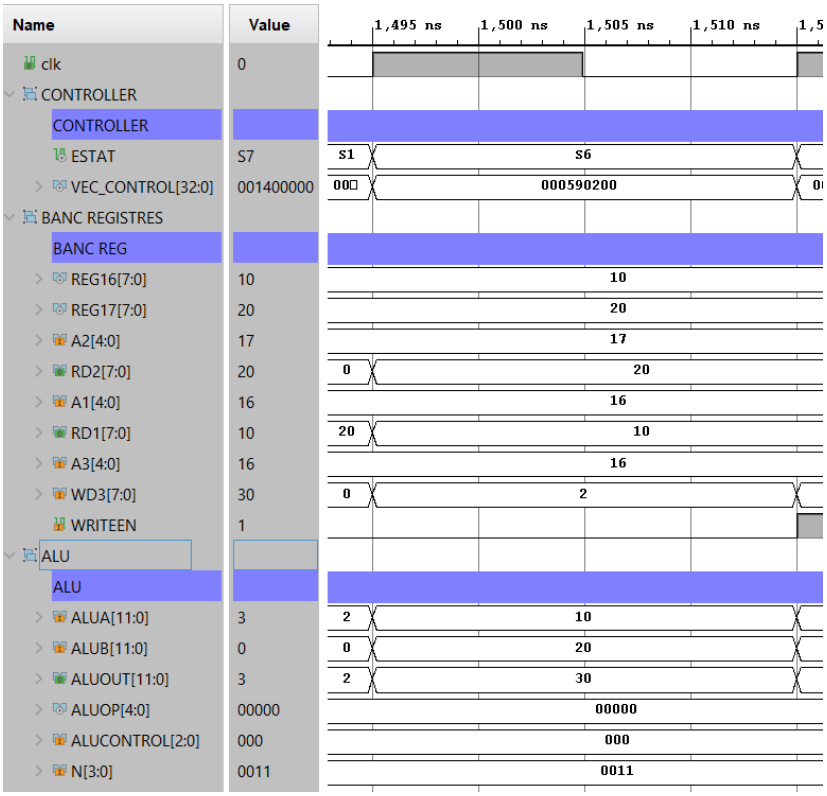


Figura 4.6 Simulació de les operacions de suma dels registres R16 i R17

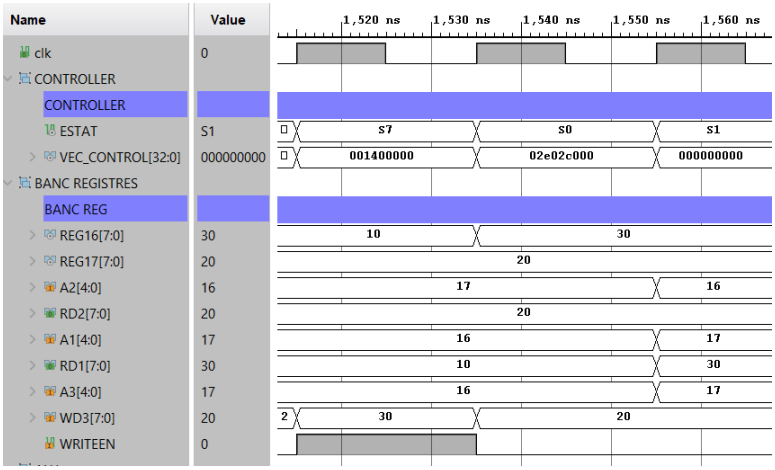


Figura 4.7 Simulació de les operacions d'escriptura del resultat al banc de registres

El pas final és escriure el resultat de la suma al banc de registres. Per això, com es veu a la Figura 4.7, a l'estat S7 s'activa el *flag WRITEEN*, de la mateixa forma que a l'estat S21, del subapartat anterior. A l'entrada *WD3* es troba el resultat de la suma, el valor decimal 30. L'adreça a la qual es desitja escriure aquest valor es R16, que es el valor de tant l'entrada d'adreces A1 com d'A3, ja que estan connectades a la mateixa sortida del descodificador. Així, quan la màquina d'estats torna a l'estat S0 per iniciar l'execució de la següent instrucció, el valor 30 queda guardat al registre *REG16*.

4.2.2 SUB

La següent operació és SUB, una resta sense carry entre dos registres, on el resultat es guarda al registre minuend. En aquest cas, el registre minuend serà R17 i el subtrahend serà el registre R16, quedant la instrucció de la següent forma en llenguatge assemblador i en codi màquina, expressat tant en binari com en hexadecimal :

```
SUB r17, r16   ; "0001 1011 0001 0000" ;    0x1B10
```

Primer de tot, com en les instruccions prèvies, s'observarà el resultat de la descodificació del codi de la instrucció, realitzada durant l'estat S1. A la Figura 4.8 es veu que, després de la descodificació, queda guardat R17 com a 'd', o registre de destí, i R16 com a registre d'origen, o 'r'. Aquesta distribució es la inversa a la de l'operació anterior, perquè si abans es volia executar la suma R16 + R17, i guardar el resultat a R16, ara es vol executar la resta R17 – R16, i guardar el resultat a R17. Al banc de registres de la figura es pot veure que el valor dels registres R16 i R17 és 30 i 20, respectivament. Seguint amb el descodificador d'instruccions, s'observa que, de nou, la sortida *FORMAT* val 0, que indica que el format d'instrucció és 1, i que el *OpCode N* de la instrucció és "0110".

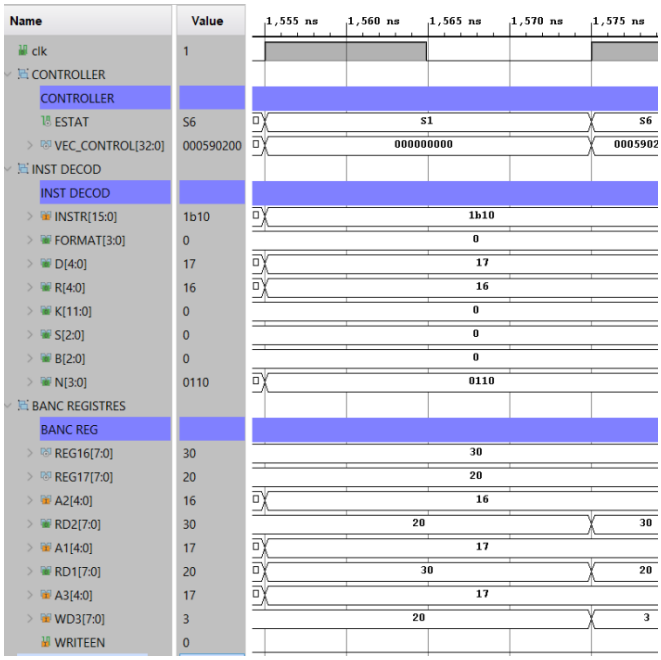


Figura 4.8 Simulació de les operacions de descodificació de la instrucció SUB

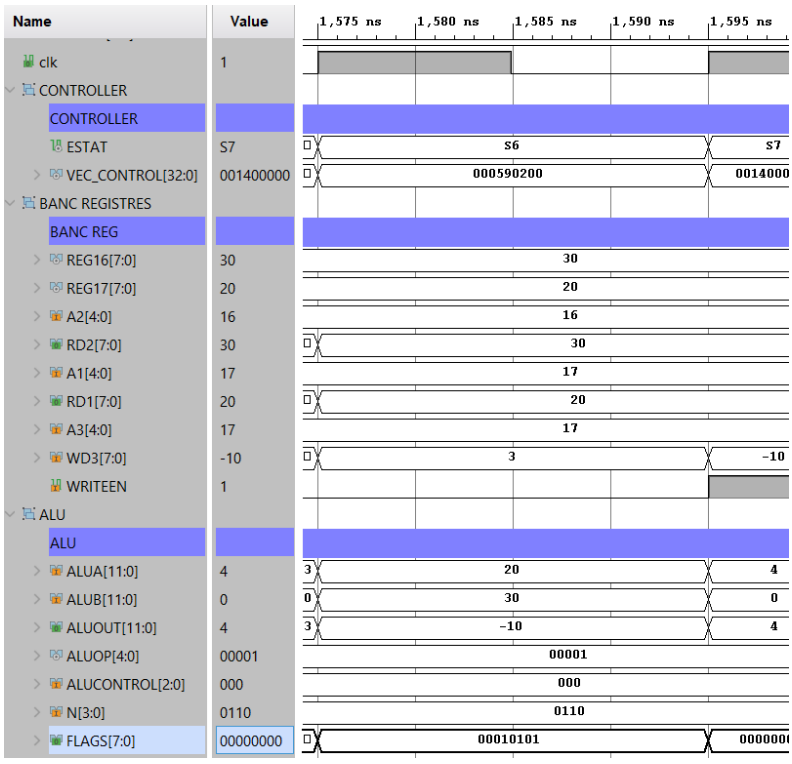


Figura 4.9 Simulació de les operacions de resta dels registres R16 i R17

El següent estat, S6, és on es produeix la resta. Igual que en l'estat S6 de l'operació de suma, es comença extraient els valors dels registres d i r, per les sortides *RD1* i *RD2* del Banc de Registres, que corresponen a les adreces de les entrades *A1* i *A2*, respectivament. Aquests valors, a través dels multiplexors de selecció d'operands de la ALU, passen a les entrades A i

B de la ALU, com es veu a l'apartat de la ALU de la Figura 4.9. *ALUCONTROL*, de nou, val "000"i per tant el descodificador de la ALU usa el valor de N per decidir el codi d'operació, *ALUOP*. Com que N val "0110", l'operació a realitzar és una resta sense carry, com s'indica a la Taula 2.7. Aquesta operació es correspon al codi *ALUOP* "00001".

El resultat, *ALUOUT*, mostra el valor decimal negatiu -10, i el vector de *flags* resultant de l'operació val "00010101", que significa que s'han activat els *flags* S,N i C, que significa que el número resultant és negatiu, que està expressat correctament en complement a 2 i que s'ha produït un "préstec" en la resta.

A la captura de la simulació de la Figura 4.10 es veu com a l'estat S6, el *flag SREGLOAD* s'activa, carregant el contingut de l'entrada, *SREGIN*, al registre SREG. El valor actualitzat de SREG es mostra al següent estat, S7. En aquest estat, a més, es carrega el valor del resultat, -10, al registre R17, *REG17*, activant el flag *REGWRITE* del vector de la Unitat de Control.

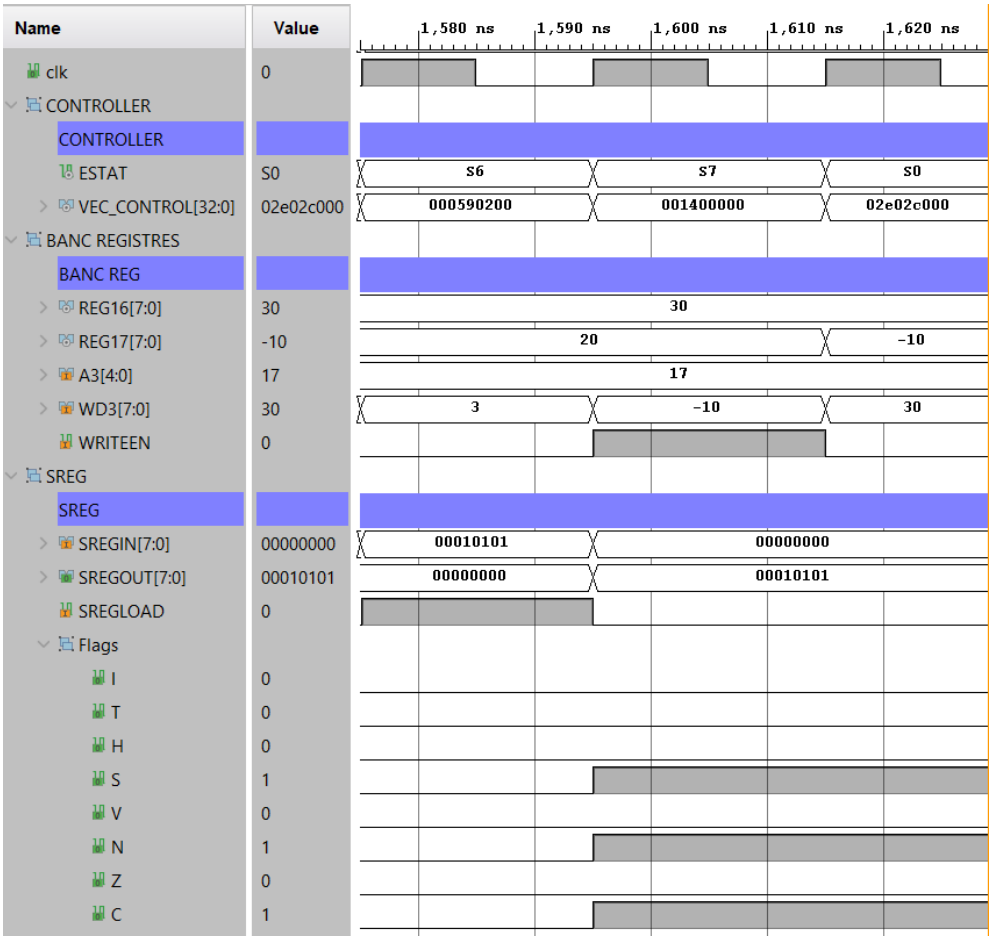


Figura 4.10 Simulació de les operacions de l'escriptura del resultat de la resta al banc de registres

4.2.3 SBCI

La següent instrucció a provar és SBCI, una resta amb *carry* entre un registre i una constant. En aquest cas, el registre minuend és R16, i la constant subtrahend és 15. La instrucció SBCI és del format 3, així que el seu codi seguirà l'estructura "01nn KKKK dddd KKKK", on "dddd"

és l'adreça del registre, que de nou haurà de ser de la meitat superior del banc de registres, i "KKKK" és el valor a restar. Així, la operació quedarà expressada com:

```
SBCI r16, 15 ; "0100 0000 0000 1111" ;   0x400F
```

Com sempre, el primer pas serà la descodificació de la instrucció. En aquest cas, la descodificació de l'estat S1, mostrat a la Figura 4.11, indica que la instrucció s'ha interpretat de forma correcta: la sortida *K* indica que el valor a restar és el valor decimal 15, la sortida 'd' val 16, en referència al registre R16, la sortida *FORMAT* mostra el valor 2, que indica que la instrucció es del Format 3 i la sortida N val "0000", que és el codi 'n' atribuït a l'operació SBCI.

EL següent pas és l'execució de l'operació. En les altres operacions, aquesta acció es duia a terme en l'estat S6. En aquest cas, però, la resta s'executarà en l'estat S12, que s'encarrega de les operacions aritmètiques i lògiques entre un registre i una constant.

Primer, a l'estat S6 s'extreu el valor del registre 'd', R16 en aquest cas, que conté el valor decimal 30, i es porta com a operand A de la ALU. Després, mitjançant el multiplexor *MUXALUSRCB,* es porta la constant 15 de la sortida del descodificador d'instruccions a l'entrada de l'operand B de la ALU. Com que *ALUCONTROL* val "011", per executar l'operació correcta, la ALU té en compte *N*, com s'indicava a al Taula 4.12. Ja que *N* val "0000", *ALUOP* tindrà el valor "10001", que indica la operació de resta amb *carry*. El valor del *carry* ve donat pel component de menys pes del vector *SREG*, que com es veu desglossat a la Figura 4.12, val '1'. No s'ha de confondre amb el vector *FLAGS,* que indica el valor del vector d'estat després de l'execució de l'operació.

Finalment, s'executa l'operació R16 – K – C que, com que C val '1', dona com a resultat el valor decimal 14, que es transporta al Banc de registres mitjançant l'estat S7, com s'ha explicat en les operacions anteriors i com es pot veure en la Figura 4.13.

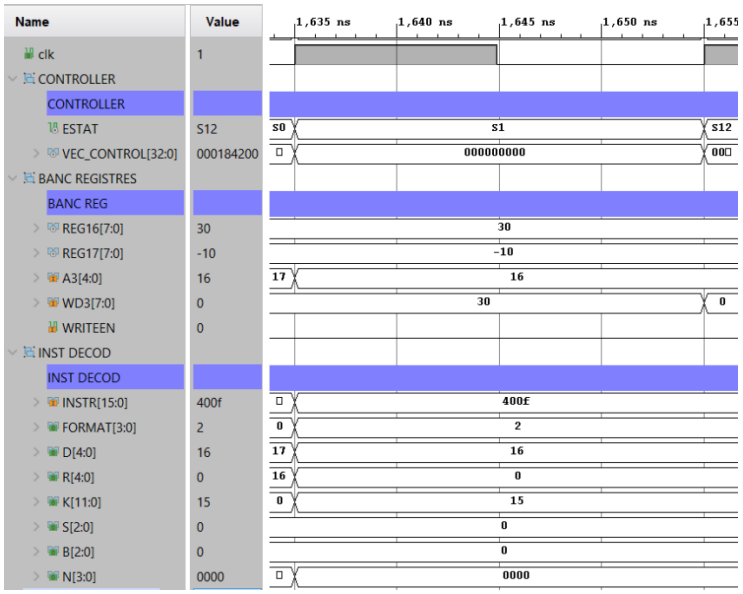


Figura 4.11 Simulació de les operacions de descodificació de la instrucció SBCI

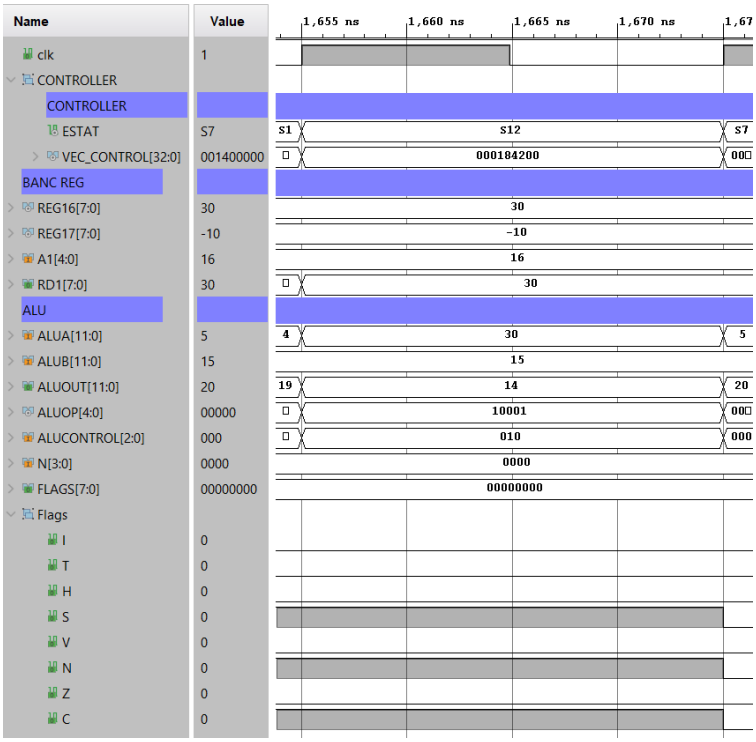


Figura 4.12 Simulació de les operacions de resta del registre R16, la constant K i el flag C

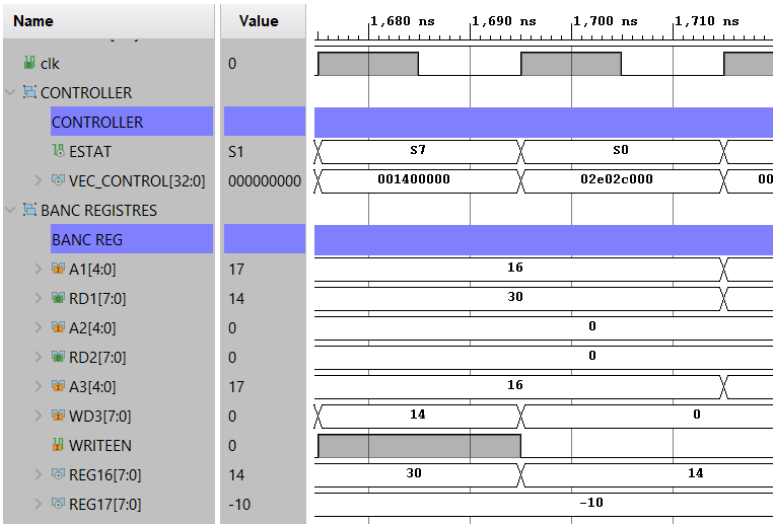


Figura 4.13 Simulació de les operacions de càrrega dels resultats de la resta al banc de registres

4.2.4 NEG

Finalment, s'executarà una instrucció aritmètica-lògica del Format 2, l'únic format pendent de simular. La instrucció que s'executarà serà la instrucció NEG, que nega el valor de un registre, convertint-lo en el seu complement a 2. El resultat, haurà de ser el mateix número interpretat en decimal amb signe, però negat.

Les instruccions del Format 2 tenen la estructura "1001 010d dddd nnnn". Com que el registre que es vol negar és el registre R17 i la operació es la negació en complement a 2, que té el *OpCode* "0001", com s'indica a la Taula 2.8, es pot assegurar que el codi d'instrucció serà:

```
NEG R17;      "1001 0101 0001 0001";      0x9511
```

El primer pas serà, com d'habitual, la descodificació. En aquest cas, a la Figura 4.14 es pot veure que el descodificador ha interpretat bé la instrucció a executar com una del Format 2, i que té a les seves sortides *D* i *N* els valors 17 i "0001", que son els valors esperats per aquesta instrucció.

De l'estat S1 passa a l'estat S11, l'estat d'execució de les operacions del Format 2. Aquest estat, molt similar als estats S6 i S12, mencionats prèviament, realitza diverses operacions lògiques en funció del valor de *N*. En aquest cas, amb *ALUCONTROL* tenint el valor "001" i N sent "0001", realitza l'operació de negació en complementa a 2, amb *OpCode* "00110". El número resultant, en números decimals i interpretat amb signe, serà -10.

Finalment, com s'ha explicat en les operacions anteriors, es guarda el valor del resultat de la sortida de la ALU al banc de registres amb S7, seleccionant el registre R17 amb l'entrada *A3*, i habilitant l'escriptura amb *WRITEN*. El resultat de la càrrega es veu a l'estat S0, com es veu a la Figura 4.15.

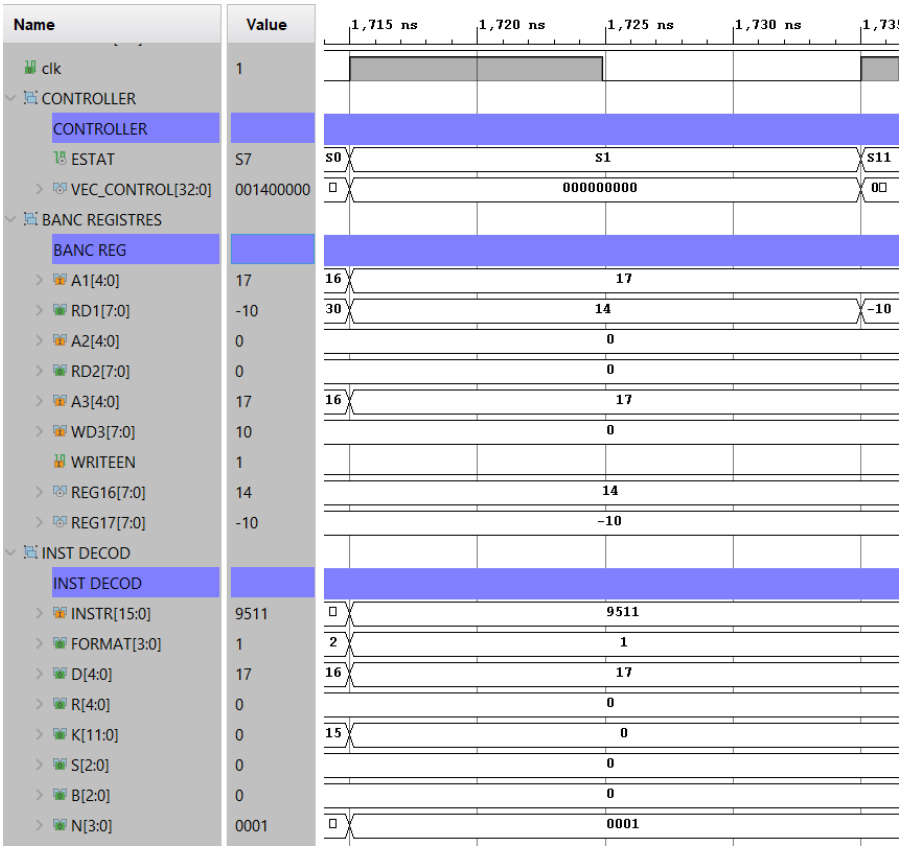


Figura 4.14 Simulació de les operacions de descodificació de la instrucció NEG

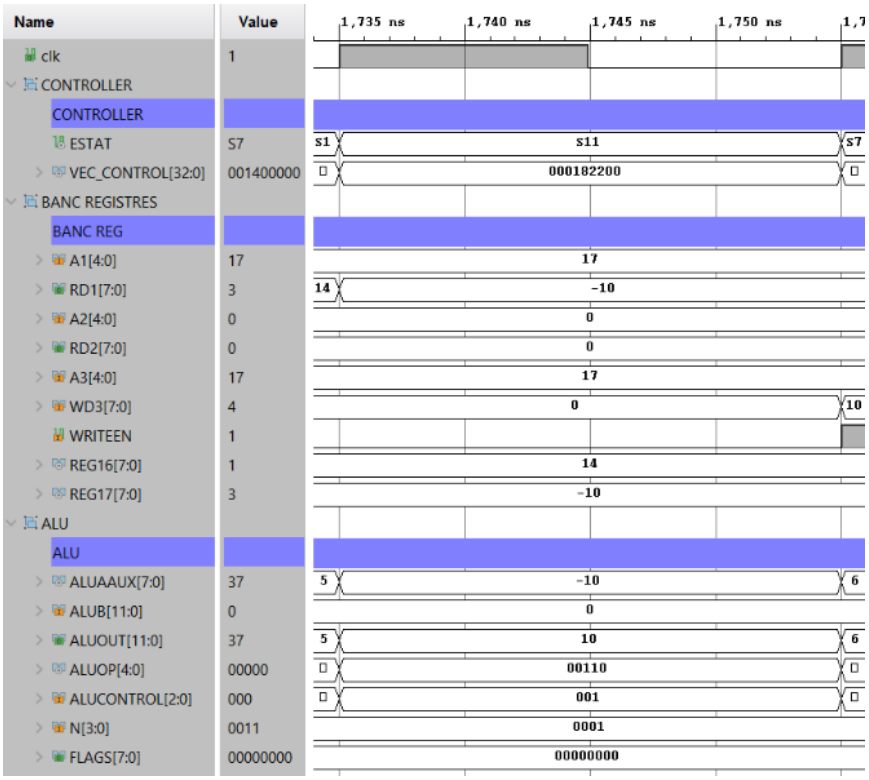


Figura 4.15 Simulació de les operacions de negació del contingut del registre R17

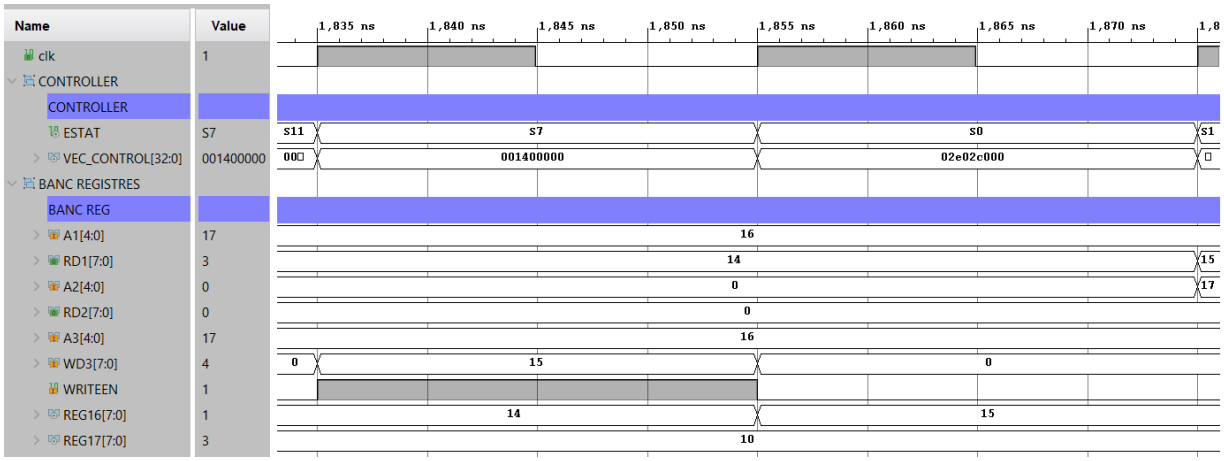


Figura 4.16 Simulació de les operacions de càrrega del resultat al banc de registres

4.3. Instruccions de Bit

Una vegada acabada la comprovació del funcionament de les instruccions aritmètiques i lògiques, ara es simularà la resposta del microprocessador a les instruccions de bit. Per fer-ho, es provaran les instruccions d'activar un bit del registre SREG, de fer rotar el contingut d'un registre i de guardar un bit d'un registre al *flag T* i després carregar-lo a un registre.

4.3.1 SEI

La primera instrucció en comprovar serà una de les encarregades de posar un '1' o un '0' a un *flag* de *SREG* de forma arbitrària. Hi ha una instrucció a nivell alt o baix qualsevol dels bit de

SREG, fent un total de 16 instruccions d'aquest tipus. S'ha escollit la instrucció SEI, que posa a '1' el *flag* I de *SREG*, perquè el flag de interrupcions no s'activa per cap altre mitjà ja que aquest microprocessador no inclou la opció de gestió d'interrupcions.

La instrucció SEI i totes les operacions que afecten els bits de *SREG* pertanyen al Format 8. Aquest format d'instruccions segueix l'estructura "1001 0100 nsss 1000", on 'n' indica el nivell al que cal posar el *flag* i "sss" marca el bit del vector (0-7) que cal canviar. Com que el *flag* I correspon a la posició 7 (el bit més significatiu) de *SREG*, i es desitja posar-lo a nivell alt, que significa que 'n' val '1', el codi de la instrucció quedarà:

```
SEI;        "1001 0100 0111 1000";          0x9478
```

Com amb totes les instruccions, el primer pas serà la descodificació de la instrucció. En la Figura 4.17 es veu com el descodificador ha identificat la instrucció correctament com una instrucció del Format 8, amb les sortides *S* i *N* mostrant els valors "111" i "0000" que corresponen. Com que el registre *SREG* no forma part del banc de registres, les sortides *R* i *D* no tenen valor en aquest tipus d'instrucció.

El següent pas serà l'escriptura de '1' al *flag* I de *SREG*. Per fer-ho, es requerirà l'ús del codificador de *SREG, SREGDECOD* i dels multiplexors *SREGSRCA, SREGSRCB* i *SREGSRCC*. Això es durà a terme en l'estat S20.

A la Figura 4.18 es pot veure com els multiplexors *SREGSRCA* i *SREGSRCB,* els multiplexors encarregats de transportar les variables correctes a les entrades *S* i *BitSet* del codificador de *SREG,* fan la seva feina de forma correcta. El multiplexor *MUXSRCA,* tot i que no es mostri al gràfic, té dues entrades, una connectada a la sortida *S* del descodificador, i una altre que val '6', l'adreça del *flag* T. Per tant, gràcies al vector de la Unitat de Control *VEC_CONTROL*, i més específicament el bit 12 d'aquest, que representa el *flag SREGSRCA*, l'adreça del bit correcte surt per la sortida de multiplexor, el port *SSSMUXOUT*.

Al multiplexor B succeeix una situació similar. Aquest multiplexor està connectat a dues entrades, una és el *flag de VEC_CONTROL* anomenat *BitSet*, que depèn realment de la sortida *N* del descodificador, i l'altre ve connectada de la sortida d'un altre descodificador, que extreu el valor d'un bit d'un registre del banc de registres. Amb el *flag SREGSRCB,* a la sortida del multiplexor, anomenada *BITSETMUXOUT*, es pot trobar el valor desitjat, '0'.

Amb totes les entrades preparades, caldrà executar el canvi de valor del bit dins del codificador de *SREG,* com es veu a la Figura 4.19. En aquesta figura, on es segueix estant a l'estat S20 de la Unitat de Control, es pot veure com funciona el codificador de SREG. Pel port d'entrada *SREGIN* entra el valor actual del registre SREG, que surt del registre d'estat per la sortida *SREGOUT*. Amb *SREGIN*, *BITSETIN* i *SIN* activa el *flag* I i extreu el nou valor de SREG per la sortida *SREGOUT* del codificador *SREGDECOD*, que val "1000 0000", és a dir, amb el *flag* I activat.

Finalment, aquest nou valor del vector de SREG cal guardar-lo al registre d'estat. Per fer-ho, com s'ha vist en altres subapartats, cal activar el *flag SREGLOAD*, però en aquest cas no serà suficient. A més, caldrà que el multiplexor *SREGSRCC* porti l'entrada correcta al port de càrrega de *SREG*. Amb un '1' al flag *SREGSRCC* del vector *VEC_CONTROL*, el multiplexor portarà el valor de la sortida de codificador a l'entrada del vector *SREG* a carregar.

Al retornar a l'estat S0, es pot veure que el contingut de *SREG* està actualitzat, i que a la seva sortida sèrie ja es troba el nou valor, que a més ja es troba a l'entrada del codificador de *SREG*, per si s'ha d'efectuar una nova operació de canvi de valor als *flags*.

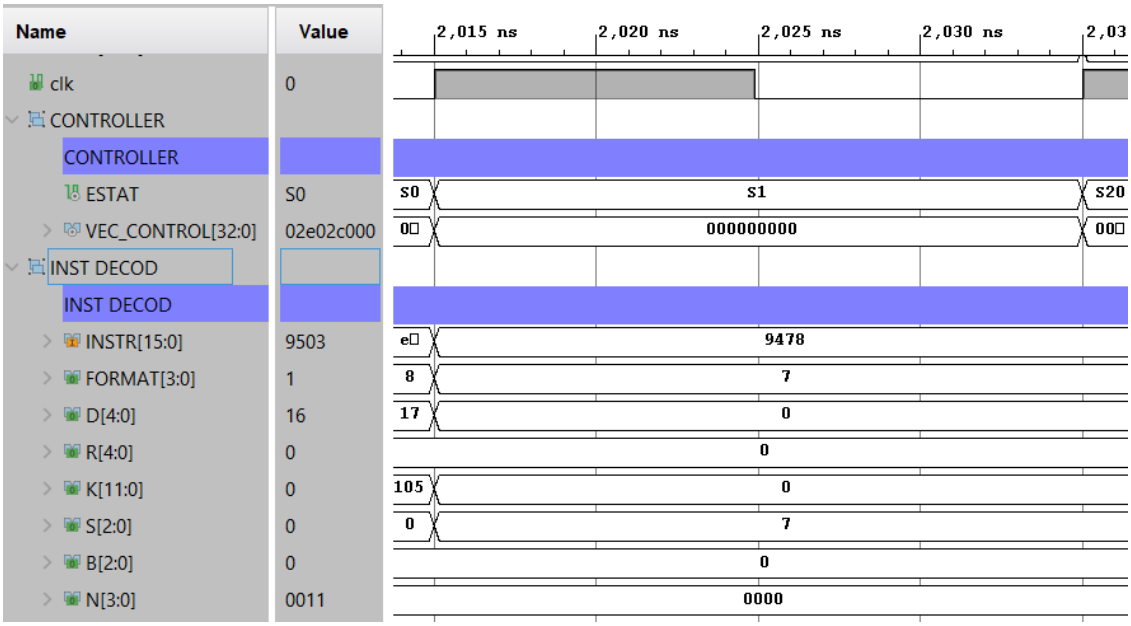


Figura 4.17 Simulació de les operacions de descodificació de la instrucció SEI

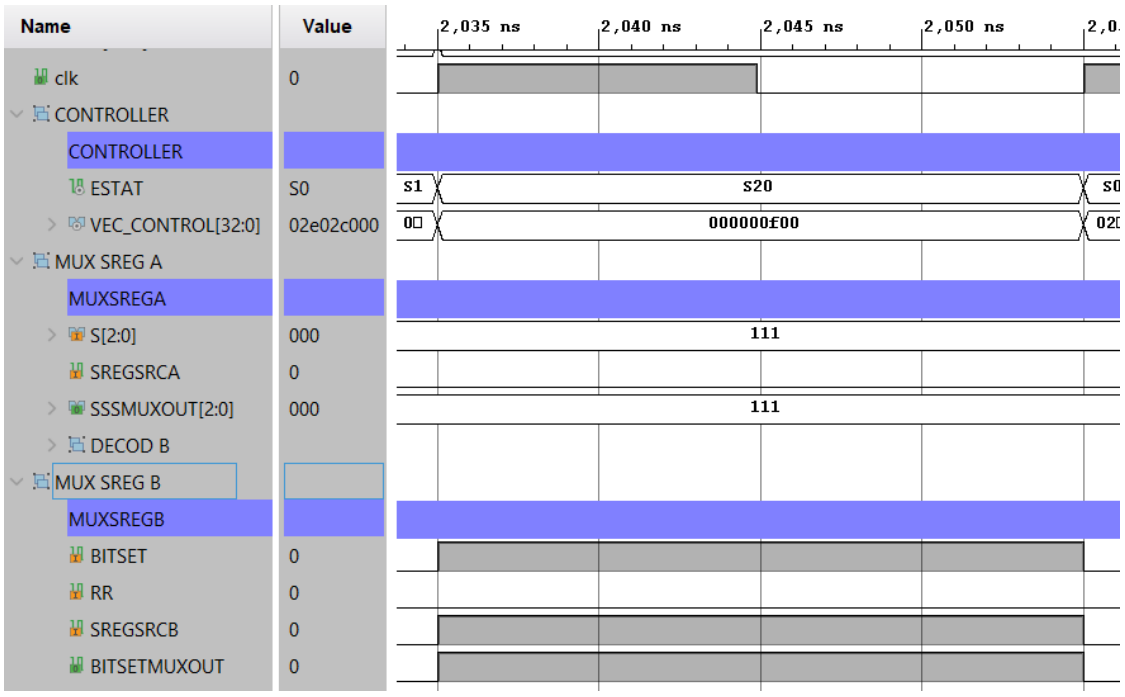


Figura 4.18 Simulació de com es seleccionen els senyals de les entrades del descodificador de SREG

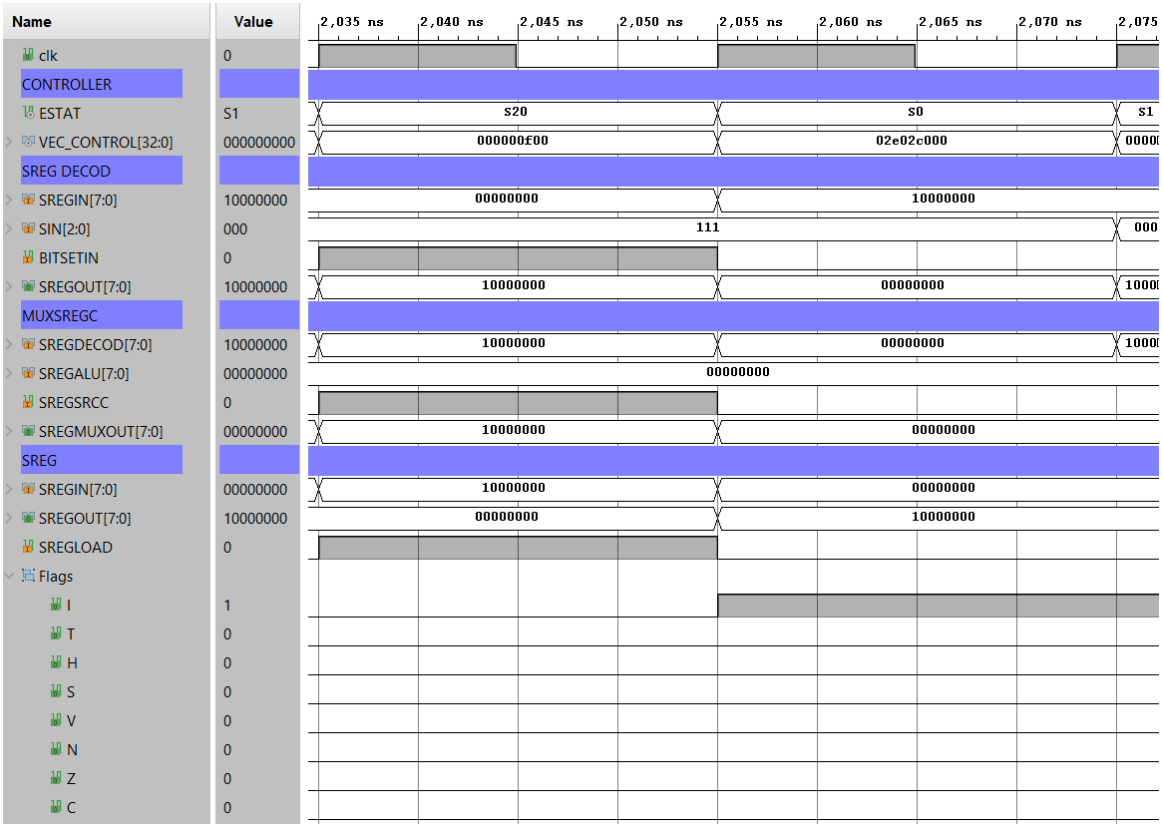


Figura 4.19 Simulació de les operacions de càrrega del nou vector d'estat al registre SREG

4.3.2 ROR

Aquesta instrucció, anomenada ROR, executa en el contingut d'un registre del banc de registres el que s'anomena una rotació a la dreta per *carry*. Aquesta operació mou tots els bit del vector una posició cap a la dreta. El bit de la posició 7 passarà a la posició 6, el bit 5 a la posició 4, etc. El bit 0, que en altres casos es perdria, en aquest tipus de rotació es guarda al bit C de *SREG*, i el contingut previ d'aquest flag es transmet al bit 7 del vector del banc de registres.

Aquesta instrucció pertany al format 2 que, com s'ha explicat prèviament, segueix el format "1001 010d dddd nnnn", sent 'd' el registre a operar i 'n' el codi de la operació. En aquest cas, 'd' serà 17 i 'n', "0111". Per tant, el codi de la instrucció serà:

```
ROR R17;          "1001 0101 0001 0111";          0x9517
```

Per comprovar que funcioni aquesta instrucció, s'ha executat 4 cops seguides, per poder veure millor els canvis en el registre i en SREG. Donat que el valor actual del registre R17 és 10, que en binari es representa en "1010", i això no dona gaire peu a veure els canvis generats per repetides rotacions, es carrega prèviament el valor "0110 1001" al registre R17 mitjançant la instrucció LDI.

A la figura 4.20 es pot veure com, per començar, R17 val "01101001", el valor assignat. Mirant el descodificador, s'observa que la instrucció s'ha descodificat correctament, interpretant-la com una del Format 2, amb *N* marcant "0111" i *D* tenint el valor 17. Per acabar, també cal fixar-se en que el *flag* C de *SREG* també val '0', d'entrada.

El següent pas serà l'execució de l'operació, que es farà en l'estat S11, explicat prèviament. En aquest pas, com es veu a la Figura 4.21, amb els valors adequats de *ALUCONTROL* i *N*, el

processador escull el codi *ALUOP* correcte per executar la instrucció. Mou tots els bits una posició a la dreta, carrega el valor del *flag Carry* (0) a la posició de més pes, i transfereix el valor del bit de menys pes al *flag* C de *SREG*. El nou valor de C i del registre R17 es poden veure en l'estat S7, que el carrega al banc de registres, com s'ha explicat en apartats anteriors.

Per finalitzar amb la prova d'aquesta instrucció, s'ha executat la rotació 4 cops, de forma consecutiva. Els resultats d'aquestes rotacions es poden veure a la Figura 4.22, on es pot apreciar com el bit de menys pes sempre va a parar al bit C de *SREG*, i el bit 7 del vector pren el valor anterior del *flag*.

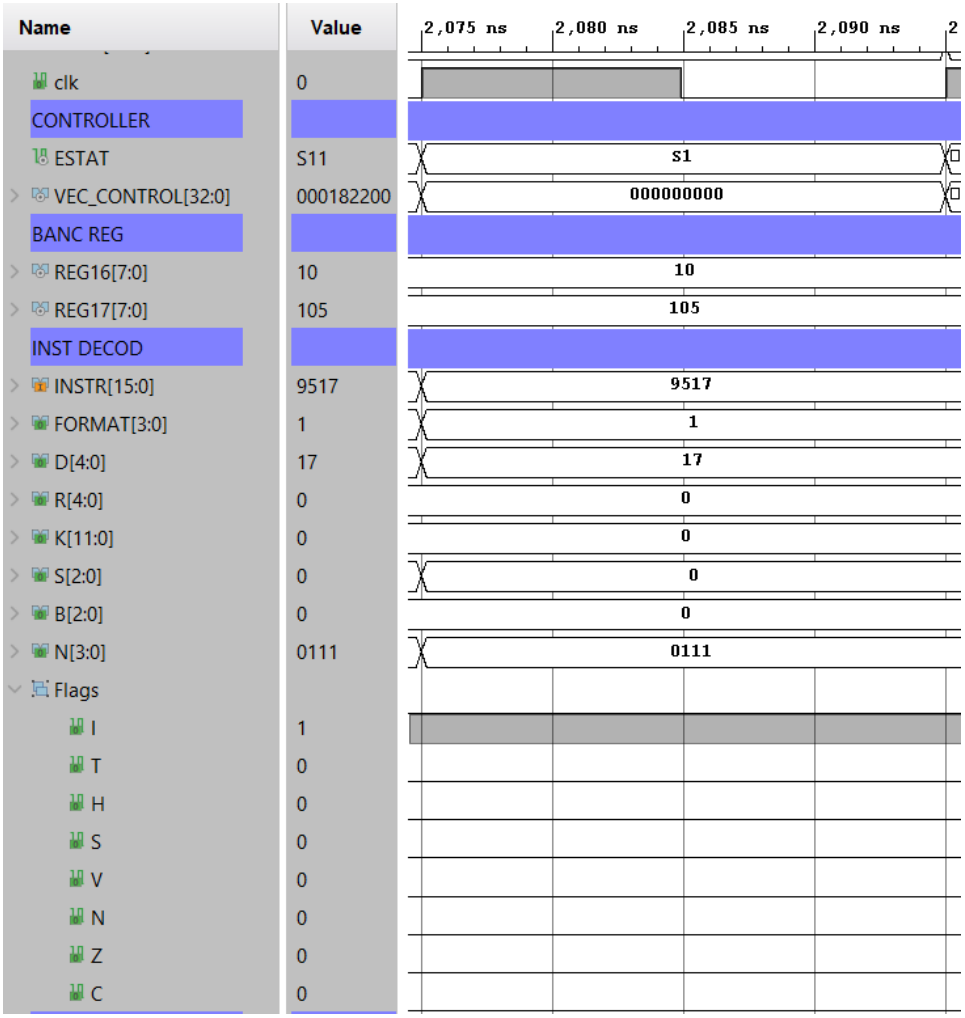


Figura 4.20 Simulació de les operacions de descodificació de la instrucció ROR

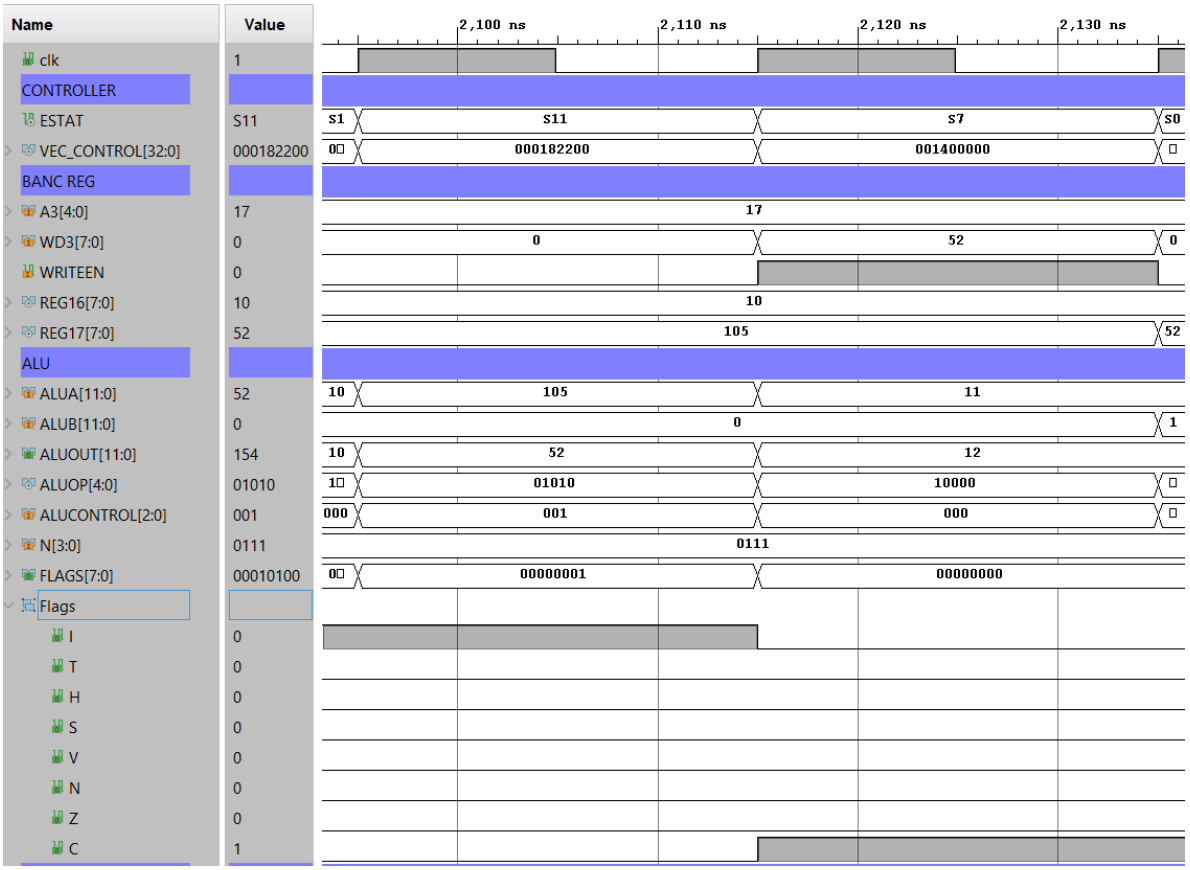


Figura 4.21 Simulació de les operacions de rotació del contingut del registre R17

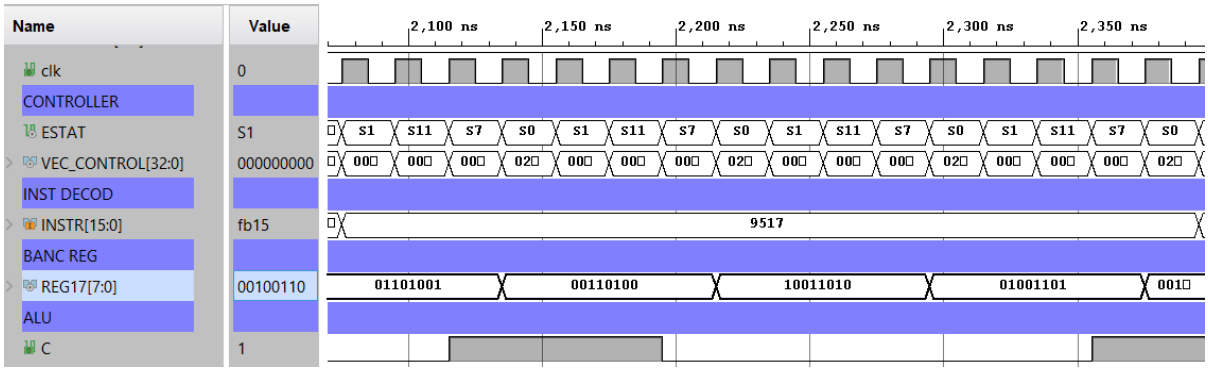


Figura 4.22 Simulació de les 4 operacions de rotació ROR executades de forma consecutiva

4.3.3 BST

La instrucció BST, o Bit Store, serveix per guardar el valor d'un sol bit d'un vector del banc de registres a *SREG*. Aquesta instrucció, del Format 5, segueix l'estructura "1111 1nnd dddd 0bbb", on 'n' i 'd' indiquen el *OpCode* i el registre d'origen, i 's' indica l'adreça del bit que cal guardar. En aquest cas es vol guardar un bit del registre R16, que després de les rotacions anteriors té un valor de "0010 0110". Per poder guardar un '1', i així poder observar el canvi en el bit T de *SREG*, 's' haurà de valer 1, 2 o 5. Per cap raó en particular, es prendrà 5 com adreça del bit a guardar. Com que hi ha quatre instruccions al Format 5, s'han de distingir

mitjançant 'n', que per BST és "01", com indica la Taula de A.5. Per tant, el codi de la instrucció quedarà com:

```
BST r17, b5;     "1111 1011 0001 0101";          0xFB15
```

El primer pas, com sempre, serà la descodificació de la instrucció. A la Figura 4.23 es pot veure que aquesta descodificació s'ha efectuat satisfactòriament, identificant la instrucció com una del format 5, amb les sortides R, *N* i *B* valent 17, "0001" i "101", respectivament. La raó per la que es mira la sortida *R* i no la *D*, que té el mateix valor, és per que en aquesta instrucció, el registre no actua com un operand de destí, sinó com a operand d'origen. Per tant, com en l'esquema de la Figura 3.21 es pot veure que el circuit lògic està preparat per implementar aquesta instrucció si l'adreça surt pel port *R* del descodificador, mentre que quan es vulguin guardar dades al registre, el processador usarà la sortida *D* del descodificador.

A l'estat S17 és on es produeix l'extracció de bit del registre R17. Per fer-ho, com es veu a la Figura 4.24, el processador utilitza el codificador Rr, *RRDECOD*. Aquest component té com a entrada el vector del banc de registres i el valor de la sortida *B* del descodificador d'instruccions. Per la seva sortida, *RRDECODOUT*, extreu el valor del bit B del vector R. El valor d'aquest bit, mitjançant el multiplexor *SREGSRCB*, es transporta a l'entrada *BitSet* del codificador *SREGDECOD*, on marca si cal posar el bit T a nivell al t o baix.

A la Figura 4.25 es pot veure el funcionament de l'estat S18, on es produeix l'escriptura del valor del bit B al *flag* T. El multiplexor *SREGSRCB* fa arribar les dades a l'entrada *BitSet*, i el multiplexor *SREGSRCA*, d'altra banda, porta a l'entrada *S* de *SREGDECOD* el valor "110", que apunta al bit T de *SREG*, que te la posició 6 al vector d'estat. Així, i amb el valor actual de *SREG*, el codificador canvia el valor de T en funció de *BitSet*, guardant el valor del bit B del vector R a *SREG*. Finalment, treu per la sortida *SREGOUT* el nou valor del registre d'estat, que mitjançant el multiplexor *SREGSRCC* arriba a l'entrada sèrie del registre *SREG* i amb l'activació de *SREGLOAD* es carrega al registre. A l'estat S0 es pot veure que el valor del *flag* T ha canviat a '1'.

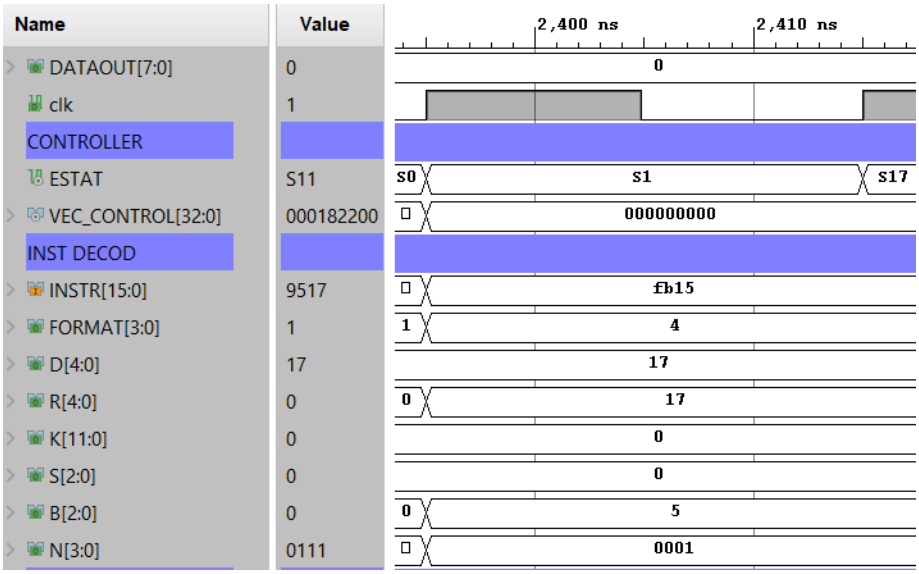


Figura 4.23 Simulació de les operacions de descodificació de la instrucció BST

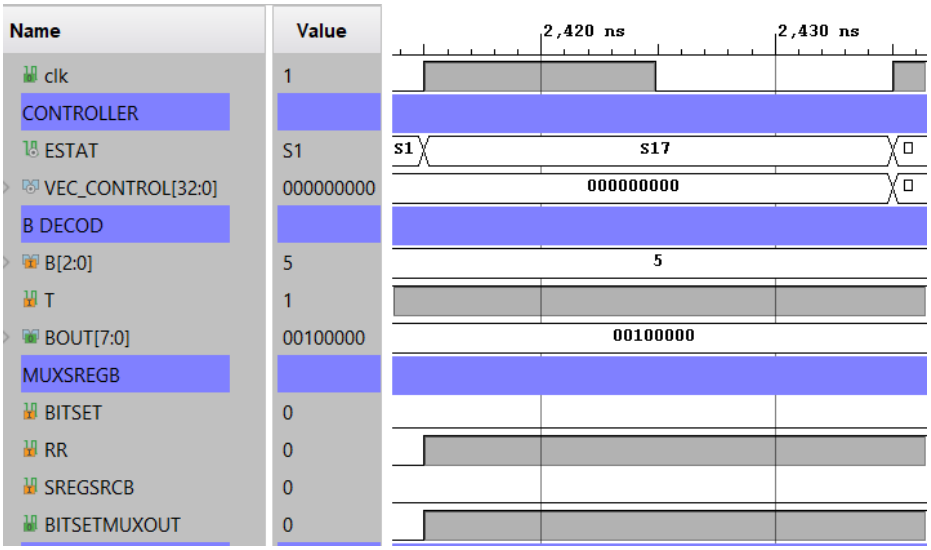


Figura 4.24 Simulació de les operacions del descodificador Rr i l'entrada del bit resultant al port BitSet del descodificador de SREG

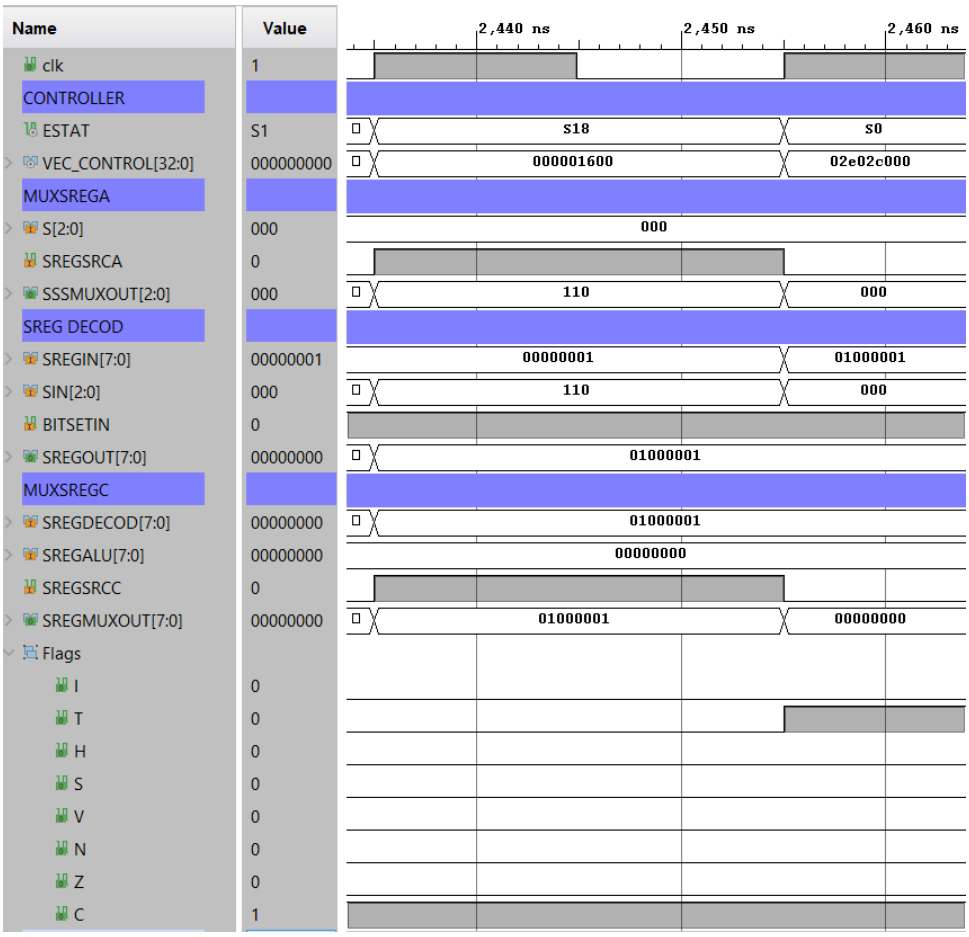


Figura 4.25 Simulació de les operacions de la càrrega del bit Rr al bit T de SREG i la càrrega del nou vector SREG al registre d'estat

4.3.4 BLD

Per acabar amb les instruccions de bit, es simularà la instrucció BLD, o Bit Load, que fa la tasca de copiar el valor del bit guardat al *flag* T de *SREG* a un bit qualsevol del contingut d'un registre del banc de registres. El camí dels estats de control d'aquesta instrucció es pot trobar a la Figura 3.30 d'aquest document, i es pot veure que es bifurca, depenent de si es vol escriure al registre un '1' o un '0'. En aquest cas, el valor guardat a T és '1', provinent del bit 5 del registre R17, i es vol escriure al bit 6 del registre R16.

La instrucció BLD, igual que la instrucció BST, simulada prèviament, pertany al Format 5. Per tant, segueix l'estructura "1111 1nnd dddd 0bbb", on 'n' i 'd' indiquen el *OpCode* i el registre de destí, i 's' indica l'adreça del bit on cal guardar el valor del *flag* T. La instrucció, per tant, tindrà la forma:

```
BLD r16, b6;    "1111 1001 0000 0110";     0xF906
```

La instrucció, com sempre, començarà amb la descodificació. A la Figura 4.26 es pot veure que s'ha produït de forma satisfactòria, identificant BLD com una instrucció de format 5, i posant els valors correctes a les sortides del descodificador. Durant al descodificació però, un altre component, *DECODB*, genera un vector per poder copiar el bit desitjat al registre. El que fa aquest component és, amb el valor de T, generar un vector que, aplicant després una funció lògica, només imposi el valor de T al bit desitjat. Per fer-ho, si el valor de T és '1', com en aquest cas, genera a la seva sortida un vector de tot zeros, excepte el bit B, que valdrà '1'. En aquest cas el vector de la sortida de B és "01000000". Així, amb una operació OR a la ALU en l'estat S16, com es pot veure a la figura 4.27, es copiarà el valor correcte al bit B del registre *D*.

Per fer l'operació, a l'estat S16 es posa el vector *ALUControl* a "101", que indica que s'ha d'executar l'operació OR entre l'operand A, el registre R16, i l'operand B, el vector de la sortida del codificador B. En aquest cas, *ALUOP* és fixe, i o depèn del valor de *N*. A més, en aquestes operacions de la *ALU*, no s'actualitza el valor de *SREG*, per que l'operació no és l'acció principal de la instrucció, i només generaria confusió en l'usuari, que no té per que saber com s'està executant la instrucció BLD.

A la sortida ALURESULT es pot veure que el '1' del bit T de *SREG* s'ha copiat exitosament al registre R16 i finalment, mitjançant l'estat S7, es guarda el nou valor de R16 al banc de registres.

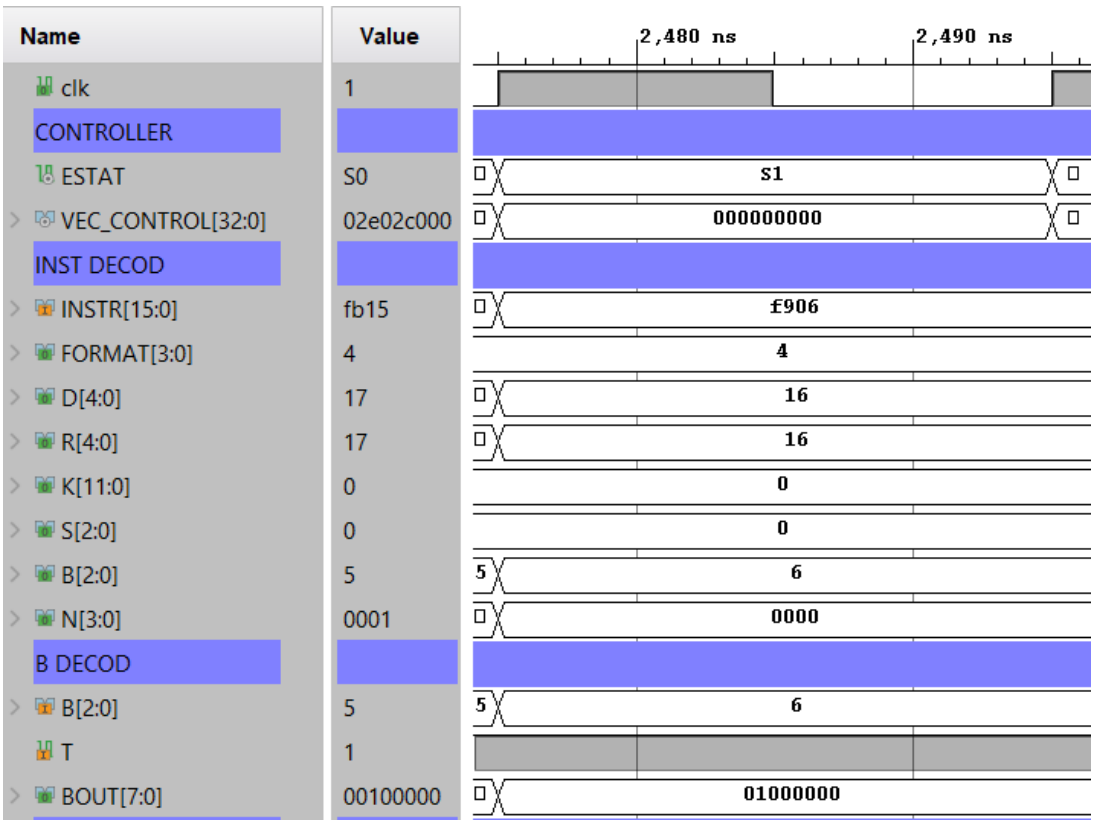


Figura 4.26 Simulació de les operacions de descodificació de la instrucció BLD i la generació del vector de càrrega de T amb el descodificador B

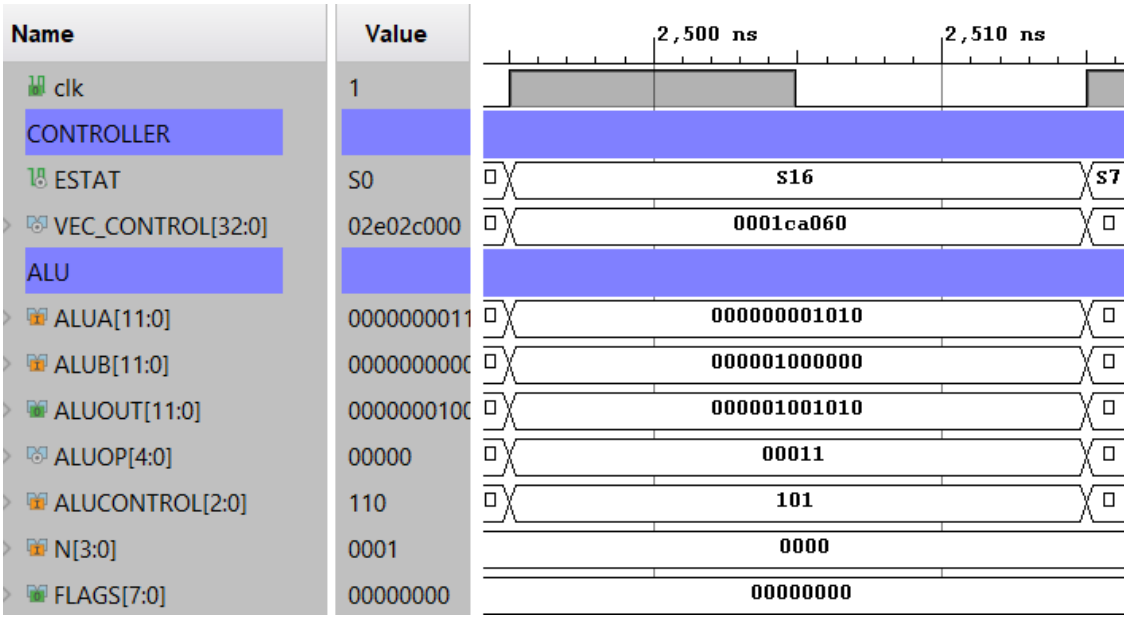


Figura 4.27 Simulació de l'operació OR entre el contingut del registre R16 i el vector de sortida del descodificador B

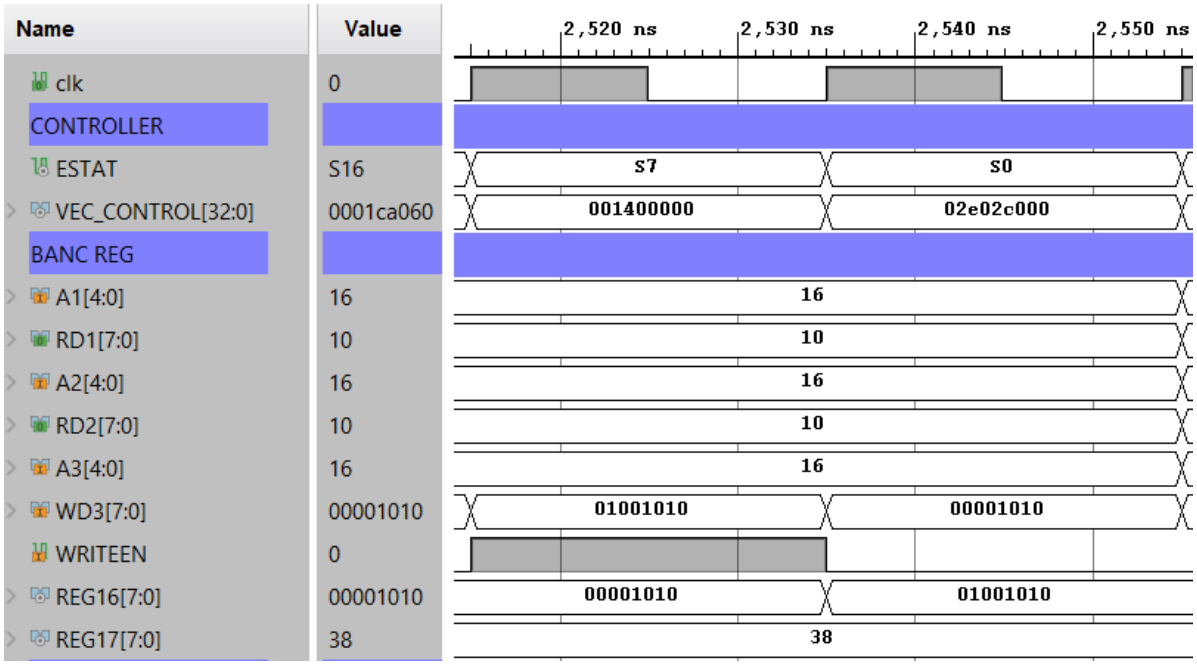


Figura 4.28 Simulació de les operacions de càrrega del resultat al banc de registres

4.4. Instruccions de Transferència

El següent conjunt d'instruccions a analitzar seran les instruccions de transferència de dades. Les instruccions d'aquest tipus, una de les quals ja s'ha vist en aquest capítol, la instrucció anomenada LDI, són les encarregades de moure dades, sense tractar-les, d'un espai de memòria a un altre, sigui la memòria RAM, el banc de registres o l'exterior.

En aquest apartat es simularan les instruccions de carrega i lectura de dades de la RAM, usant les instruccions ST i LD, i la transferència de dades amb l'exterior, amb les operacions IN i OUT.

4.4.1 ST

Aquesta instrucció comunica el banc de registres i la memòria RAM incorporada al processador. S'encarrega de guardar el contingut d'un registre del banc de registres a una adreça de la memòria RAM. L'adreçament de la RAM es fa mitjançant el registre ZLo del banc de registres, que haurà de ser carregat amb un valor vàlid d'adreça abans d'iniciar la instrucció ST. Com que el registre ZLo ocupa la posició de R30 al banc de registres, és accessible per la instrucció LDI, i se li escriurà el valor 3, que serà l'adreça de la memòria RAM que s'escriurà.

La instrucció ST, juntament amb LD, formen el conjunt de les instruccions del Format 4. Aquestes instruccions segueixen l'estructura "10q0 qqnd dddd nqqq", on 'n' i 'd', com d'habitual, representen el codi de l'operació a dur a terme i l'adreça del registre operand. La incorporació nova 'q', serveix per indicar el desplaçament en la càrrega. Aquest desplaçament, en altres processadors, serveix per carregar valors a la RAM mentre es va augmentant o disminuint el valor de l'adreça. Alguns processadors tenen fins a 9 formes diferents d'adreçar-se a la memòria RAM, però el que s'està provant ara només en té una, per lo qual 'q' romandrà

sempre a '0'. Això no suposa una disminució de les possibilitats d'adreçament, sinó que significa que el processador trigarà més temps i requerirà més instruccions per dur a terme les mateixes accions. La instrucció quedarà, per tant, de la següent manera:

```
ST Z, r16;        "1000 0011 0000 0000";          0x8300
```

Com en qualsevol instrucció, s'iniciarà la prova descodificant la instrucció. A la Figura 4.29 es pot veure que s'ha produït de forma satisfactòria, tot i que 'q' no té sortida, perquè sempre manté el seu valor a '0', com s'ha dit abans. La sortida *D*, el registre de destí, val 16, que és el registre d'on s'extraurà el valor a guardar a la RAM. De la sortida *R* surt el valor 30, l'adreça del banc de registres de ZLo, que conté l'adreça de la RAM on guardar les dades. Això s'ha fet per facilitar el disseny del *Datapath*, i per això és un canvi excepcional dels registres d'origen i destí.

A la Figura 4.30 es pot veure el funcionament de l'estat S2, on s'extreu el valor de R16, que surt del banc de memòria per la sortida RD1, i l'adreça 3 de la RAM, que surt del port RD2 del banc de registres. El vector de la sortida RD1 passa per ALU mitjançant l'operació PASS, indicada per *ALUCONTROL* amb el valor "100".

Després, a l'estat S5, visible a la Figura 4.31, amb els *flags RAMEN* i *MEMWRITE*, que habiliten la memòria RAM i hi permeten l'escriptura d'aquesta, es carrega el valor de R16 a l'adreça marcada per ZLo. Es pot veure el nou valor de RAM a la Figura 4.31, a l'estat S0.

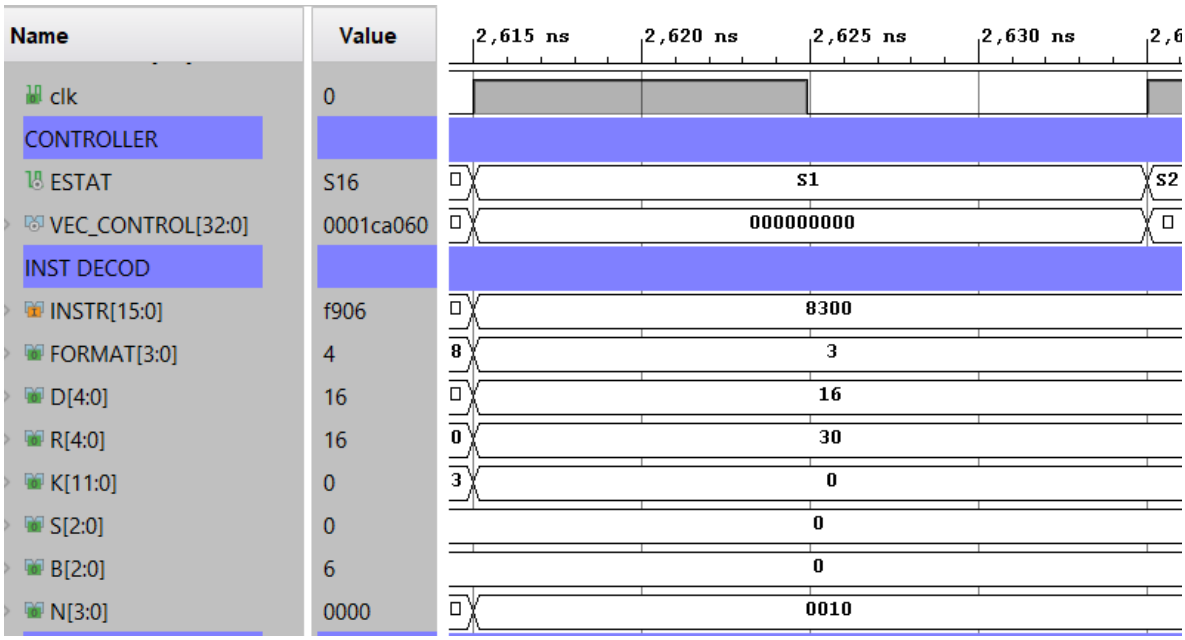


Figura 4.29 Simulació de les operacions de descodificació de la instrucció ST

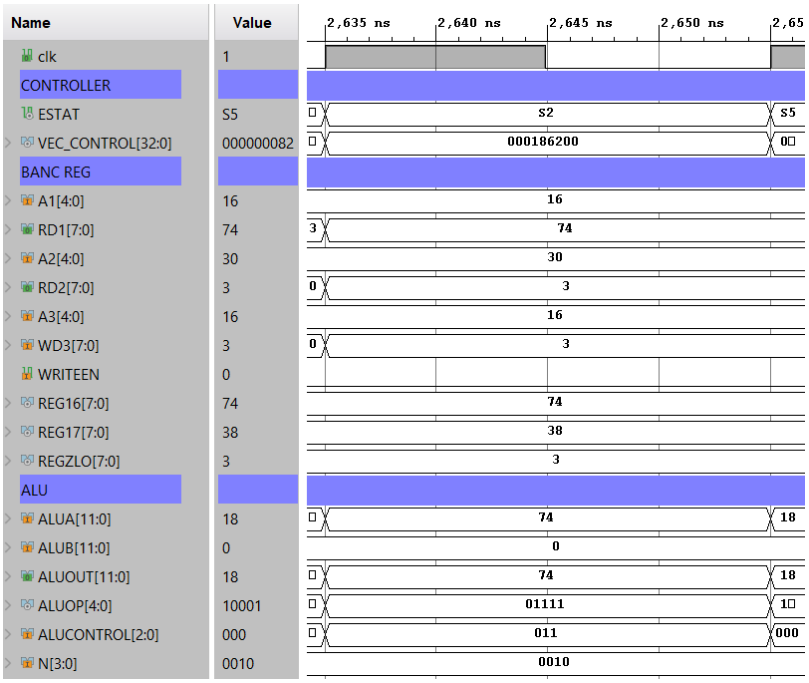


Figura 4.30 Simulació de l'obtenció de l'adreça de la memòria RAM mitjançant el registre ZLo

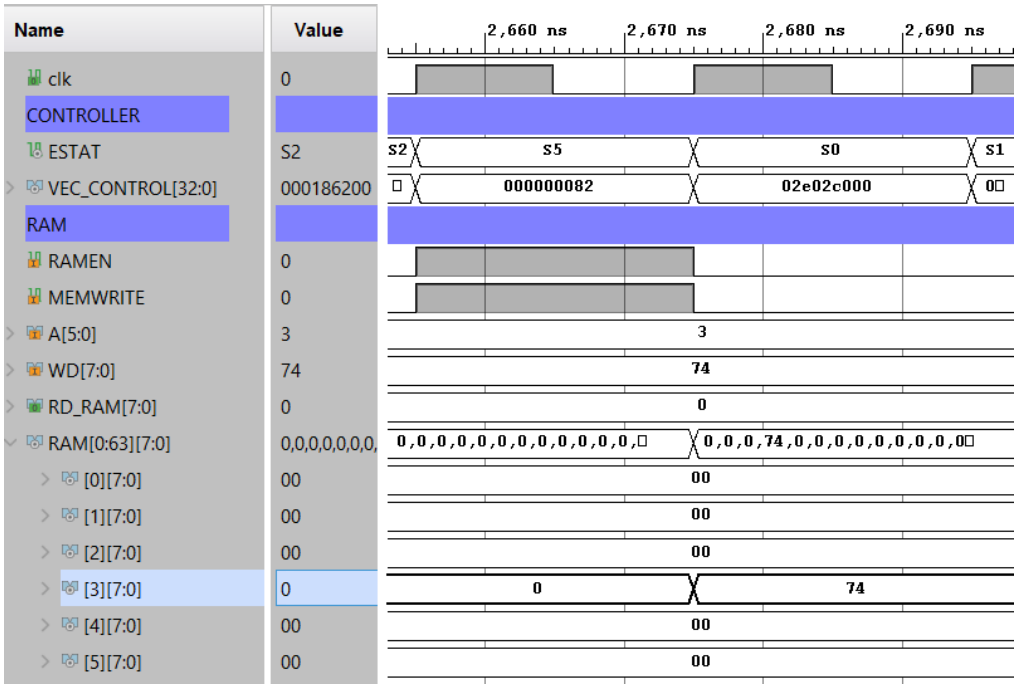


Figura 4.31 Simulació de les operacions d'escriptura de les dades a la memòria RAM

4.4.2 LD

Amb un valor carregat a la Memòria RAM, el següent pas per comprovar el seu funcionament serà carregar el valor guardat a l'adreça 3, a un registre del banc de registres. Per fer això, es farà servir la instrucció LD, del Format 4.

Aquesta instrucció necessita, per executar-se de forma correcta, una adreça on escriure el vector guardat a la RAM, que en aquest cas serà el registre R17; un *OpCode* 'n', que per la instrucció LD és "00", i un desplaçament 'q' que en aquest processador ha de ser '0', com s'ha explicat a l'apartat anterior. Aquests components del vector d'instrucció, seguint l'estructura de les instruccions del Format 4, quedaran posats de la següent forma:

```
LD R17, Z;        "1000 0001 0001 0000";          0x8110
```

A la Figura 4.32, es veu com, a l'estat de descodificació, l'estat S1, els valors han estat interpretats de forma correcta pel processador, i com a l'estat S2 es preparen les entrades de la RAM per obtenir el valor guardat a la seva adreça 3. Després, a la instrucció S3, es llegeix la RAM. En aquesta instrucció, a diferencia de l'estat S5 de la instrucció ST, no cal activar el *flag MEMWRITE*, perquè no es vol escriure a l'adreça de memòria 3, sinó que es vol llegir. Per fer-ho, de RD2 surt el valor de l'adreça de la RAM que es vol guardar al registre R17, i es llegeix la memòria RAM.

Finalment, a l'estat S4, el valor extret de la memòria RAM, que es troba a la sortida *RD_RAM*, es guarda al banc de registres a l'adreça marcada per la sortida *D* del descodificador d'instruccions, R17. El canvi en el valor del registre es pot veure a l'estat S0, a la Figura 4.33.

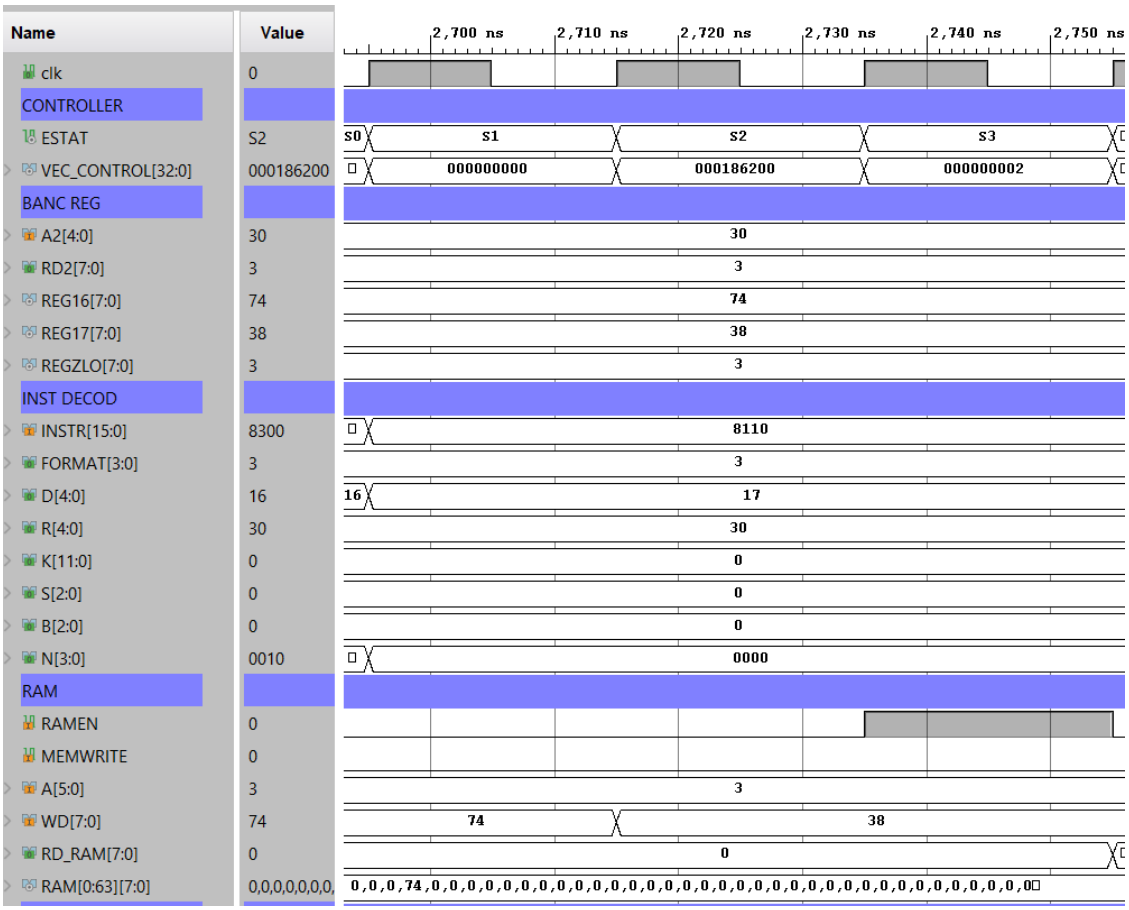


Figura 4.32 Simulació de les operacions de descodificació de la instrucció LD i l'obtenció de les dades guardades a la RAM

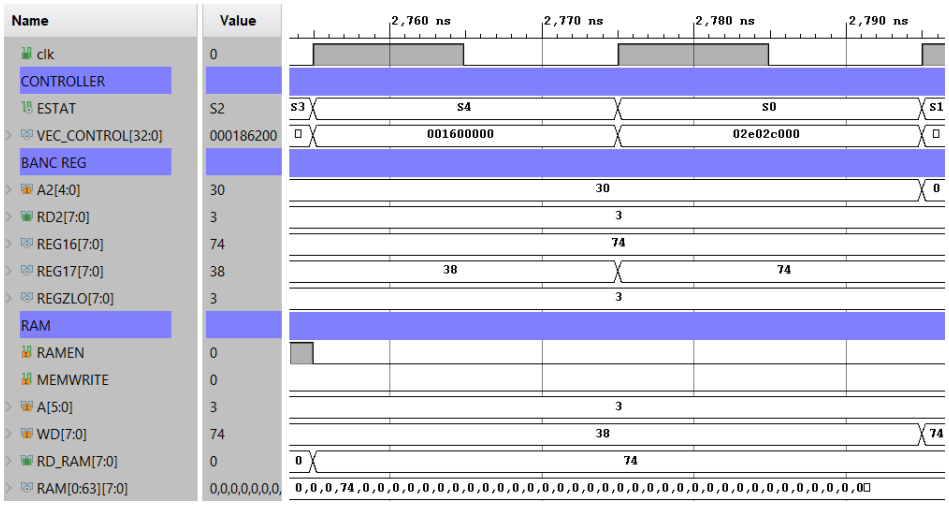


Figura 4.33 Simulació de les operacions d'escriptura de les dades de la RAM al banc de registres

4.4.3 IN

Una vegada provat el funcionament de la memòria RAM i la seva comunicació amb el banc de registres, es provarà el la connexió entre el microprocessador i l'exterior. La transferència de dades es farà mitjançant dos ports, un d'entrada de dades en paral·lel i un altre de sortida. Primer es provarà l'entrada de dades, mitjançant la instrucció IN.

Per realitzar aquesta simulació, al *testbench* que s'està utilitzant en la simulació, s'afegirà un valor fixe decimal 92 a l'entrada *DATAIN*. Aquest valor, que entrarà pel port i es guardarà al registre REGIN, es guardarà posteriorment al registre R16 del banc de registres.

La instrucció IN pertany al Format 6, que segueix l'estructura "1011 nAAd dddd AAAA", on 'n' i 'd' són el *OpCode* i el registre operand, i A és l'adreça del port d'on rebre la informació. En altres processadors, poden haver-hi múltiples ports de sortida, i per això caldria adreçar-los amb el component A, però com en aquest processador només hi ha un port d'entrada i un de sortida, A sempre tindrà el valor 0. Sabent els valors de 'n', 'd' i 'A', podem assegurar que el codi de la instrucció quedarà així:

```
IN r16, 0;      "1011 0001 0000 0000";          0xB100
```

A la Figura 4.34 es pot veure com el descodificador, a l'estat S1, identifica correctament la instrucció com una del Format 6, i troba que 'd' val 16, i 'n' i 'A' valen 0. A la següent figura, la Figura 4.35, es pot veure com, a l'estat S28, es carrega el valor del port al registre IREG, mitjançant l'activació dels *flags IREGEN* i *IREGWR*. Després, a l'estat S19, aquest valor de IREG es carrega al banc de registres, mantenint activat el bit *IREGEN*, i activant també el senyal *REGWrite*. El valor importat pel port d'entrada es veu carregat al registre R16 en l'apartat S0.

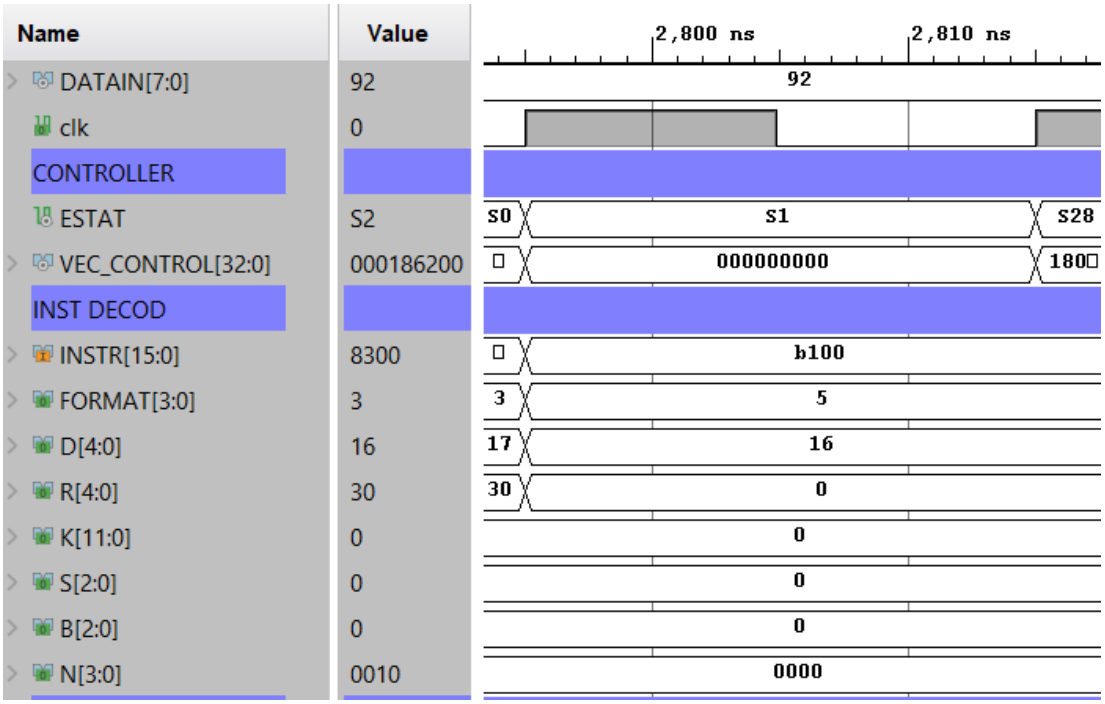


Figura 4.34 Simulació de les operacions de descodificació de la instrucció IN

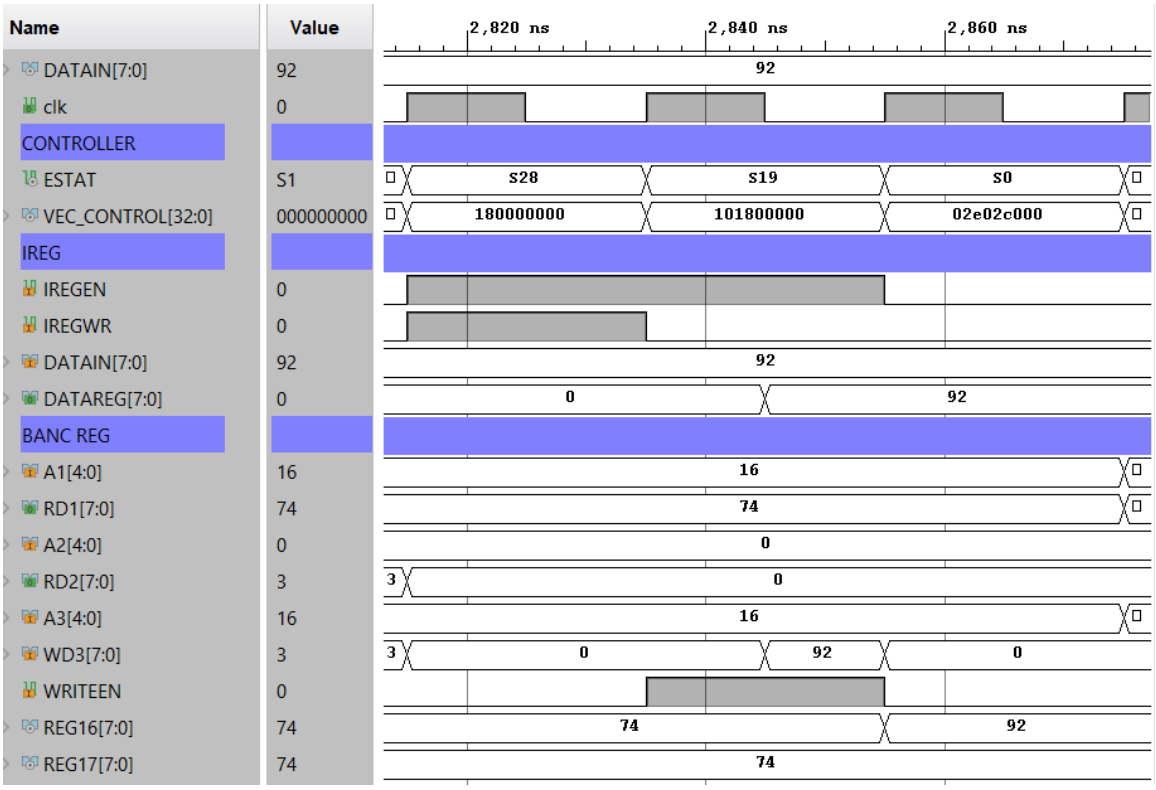


Figura 4.35 Simulació de les operacions d'entrada de dades pel registre REGIN i la seva escriptura al banc de registres

4.4.4 OUT

Per acabar amb la simulació de les operacions de transferència, es carregarà el valor del registre R17 al port de sortida de dades mitjançant la instrucció OUT. Aquesta instrucció, com IN, forma part del Format 6, però a diferencia de la instrucció anterior, aquest cop 'n' tindrà el valor '1', per diferenciar-les. 'A' seguirà sent 0, ja que només hi a un port de sortida, i 'd' serà 17 aquest cop. Per tant, el codi de la instrucció prendrà la següent forma:

```
OUT 0, r17;      "1011 1001 0001 0000";          0xB910
```

El pas de la descodificació, mostrat en la Figura 4.36, es veu que és exitosa: la sortida *D* val 17, *N* val "0001" i identifica correctament la instrucció com una del Format 6. A la següent figura, la Figura 4.37, es pot veure com es carrega el valor de R17 al registre de sortida, *OREG*, i com aquest el transfereix al port de sortida, *DATAOUT*.

A l'estat S27, les dades del registre R17 surten pel port *RD2* del banc de registres, i van a parar al port d'entrada *REGDATA* del registre OREG. Amb l'habilitació d'escriptura del registre OREG, el contingut de R17 es guarda al registre, i es transmet a la sortida del microprocessador, *DATAOUT*.

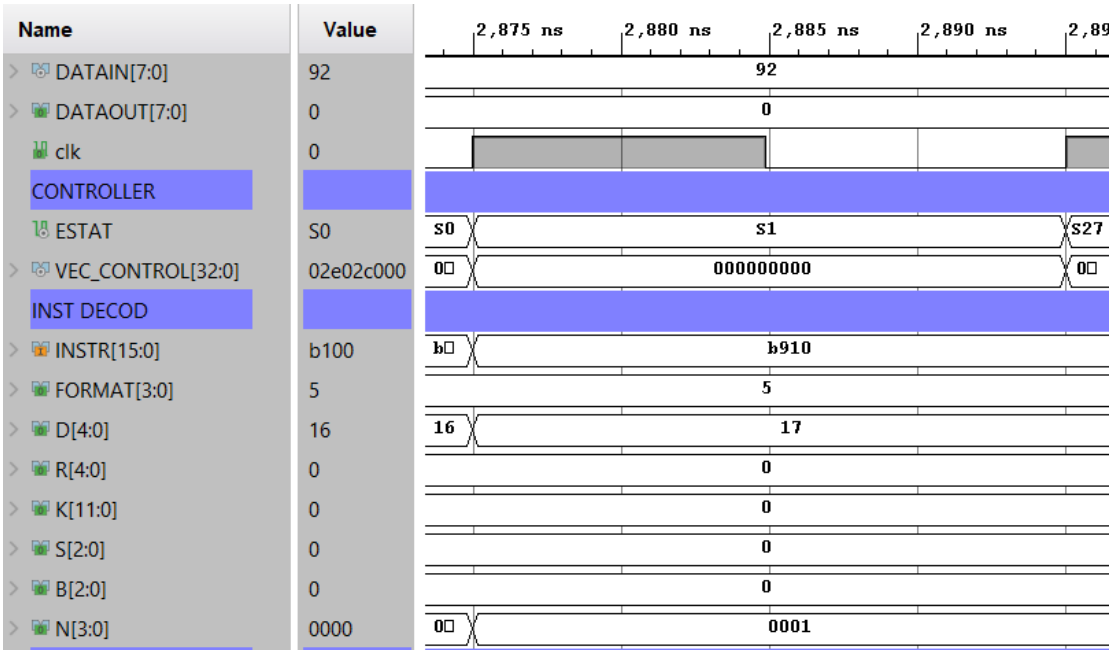


Figura 4.36 Simulació de les operacions de descodificació de la instrucció OUT

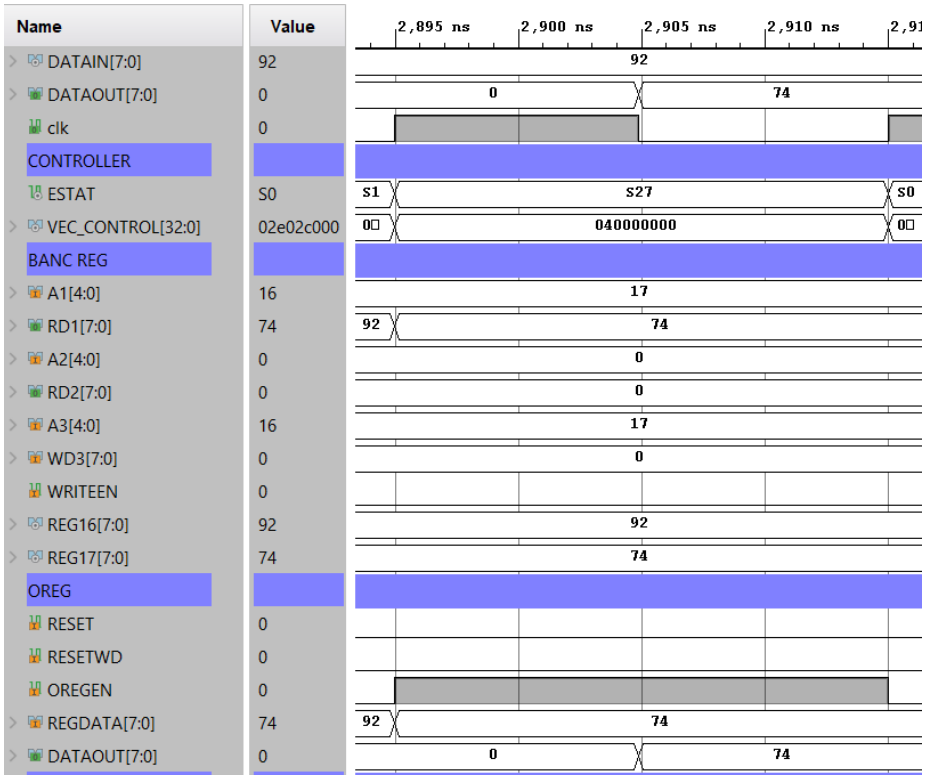


Figura 4.37 Simulació de les operacions d'escriptura de les dades del banc de registres al registre OREG i la seva sortida pel port de sortida de dades DATAOUT

4.5. Instruccions de Branca

Per acabar amb les simulacions, es simularan les instruccions de salt o de branca. Aquestes instruccions fan cavis al comptador de programa, PC, permetent la creació de bucles i diverses opcions d'execució condicionals. Hi ha 2 tipus de salt, el salt condicional i l'incondicional, i després també es poden distingir segons la forma d'indicar la mida del salt. Per fer una mostra completa, es simularan dos tipus de salt, el salt incondicional amb retorn, que fa ús de la memòria *Stack,* i el salt condicional depenent del valor d'un bit del registre *SREG*.

4.5.1 RCALL i RET

Per realitzar la simulació de les operacions RCALL i RET, es farà servir el programa que s'ha usat per la simulació de les altres instruccions, però s'afegiran instruccions al final, a les quals saltarà el programa. Una vegada executades aquestes operacions, amb la instrucció RET, el programa retornarà on era i seguirà executant el programa amb normalitat.

Així doncs, es començarà presentant els codis de instrucció i el programa resultant. La instrucció RCALL és del Format 12, i per tant segueix el format "110n kkkk kkkk kkkk". El component 'n', que en aquest cas valdrà '1', serveix per diferenciar RCALL de l'altre instrucció del Format 12, RJMP. El vector 'k' té 12 bits, i es el vector més llarg amb el que treballa el processador. Té aquesta llargada perquè ha de permetre fer salts de des de qualsevol punt del programa fins a qualsevol punt del programa, és a dir, ha de poder prendre qualsevol valor en el rang [-512, 512]. Això es podria aconseguir amb només 10 bits de K, però si en un

futur es desitja fer més gran la memòria Flash, es podria arribar a expandir sense problema fins a les 1024 adreces.

En el cas actual, el salt desitjat serà des de la posició 2 de PC fins al final del programa actual, que suposa un salt de 19 posicions de PC. Per tant, el codi de la instrucció serà:

```
RCALL 19;  "1101 0000 0001 0011";     0xD013
```

La instrucció de retorn, RET, és una instrucció del Format 13, que torna el valor de PC al que era abans del salt amb RCALL. Les instruccions d'aquest format segueixen l'estructura "1001 010n 000n 100n", on 'n' marca la diferència entre les diferents instruccions que hi pertanyen. En aquest cas, com que 'n' val "000", el codi de la instrucció serà:

```
RET;       "1001 0101 0000 1000";     0x9508
```

Per veure l'efecte del salt i del retorn, s'executarà el programa següent:

```
0.    "1110000000001010", --LDI R16 <- 10      E00A
1.    "1110000100010100", --LDI R17 <- 20      E114
2.    "1101000000010011", --RCALL (19)         C013


3.    "0000111100000001", --ADD R16, R17       0F01
4.    "0001101100010000", --SUB R17, R16       1B10
5.    "0100000000001111", --SBCI R16, 15       400F
6.    "1001010100010001", --NEG R17            9511
7.    "1001010100000011", --INC R16            9503
8.    "0010001100000001", --AND R16, R17       2301
9.    "1110011000011001", --LDI R17 <--75      E114
10.   "1001010001111000", --SEI                 9478
11.   "1001010100010111", --ROR R17            9517
12.   "1001010100010111", --ROR R17            9517
13.   "1001010100010111", --ROR R17            9517
14.   "1001010100010111", --ROR R17            9517
15.   "1111101100010101", --BST R17, B5        FA55
16.   "1111100100000110", --BLD R16, B6        F863
17.   "1110000011100011", --LDI RZLo <- 3      E0E3
18.   "1000001100000000", --ST R16             8300
19.   "1000000100010000", --LD R17             8110
20.   "1011000100000000", --IN R16             B100
21.   "1011100100010000", --OUT R17            B910


22.   "1110000100101110", --LDI R18 <- 30      E12E
23.   "1110001000111000", --LDI R19 <- 40      E238
24.   "1110001101000010", --LDI R20 <- 50      E342
25.   "1110001101011100", --LDI R21 <- 60      E35C

26.   "1001010100001000", --RET                9508
```

Per començar, es tractarà la simulació del salt, RCALL. Primer, a la Figura 4.38, es pot veure que, quan la sortida de PC, *PCN*, val 2, s'inicia l'execució de la instrucció RCALL, amb l'entrada *INSTR* de PC tenint el valor en hexadecimal "D013". En la figura també s'observa l'estat S1, on es fa la descodificació de la instrucció. Es poden identificar tots els valors esperats, amb la sortida *N* valent '1' i *K* tenint el valor decimal 19.

Després, com es veu reflectit a la Figura 4.39, es guarda el valor actual de PC a la memòria *Stack* a l'estat S23. Per fer-ho, s'activen els senyals *STACKEN* i *STACKWR*, que serveixen per activar la memòria *stack* i permetre l'entrada de dades. Segons el funcionament de la memòria *stack*, explicada en apartats anteriors, l'adreça actual de PC, és a dir, l'adreça de la instrucció que hauria d'executar el programa si no fos pel salt, es guarda al nivell superior de la *stack*, com es pot veure al senyal *STACKAUX*.

A continuació, encara en l'estat S23, però ara observant la Figura 4.40, es genera el canvi de valor de PC. Primer, amb la ALU, es suma el valor de PC a executar a continuació, 3, amb el salt que es vol fer, 19. Com sempre en els casos on l'operació aritmètica no és la finalitat de la instrucció, no es genera un canvi a SREG, encara que el senyal intern de *FLAGS* canviï.

El nou valor de PC, 22, passa a l'entrada *PC* del comptador de programa, i s'activa el senyal de PCWRITE. El següent estat, S26 manté inactiu el processador, però li dona temps a carregar correctament el valor de PC sense començar l'execució de la següent instrucció, que ha de ser la del salt. Finalment, a S0, es pot veure que el valor de *PCN* és el correcte, i que la següent instrucció a executar és E12E, la instrucció afegida al final del programa inicial.

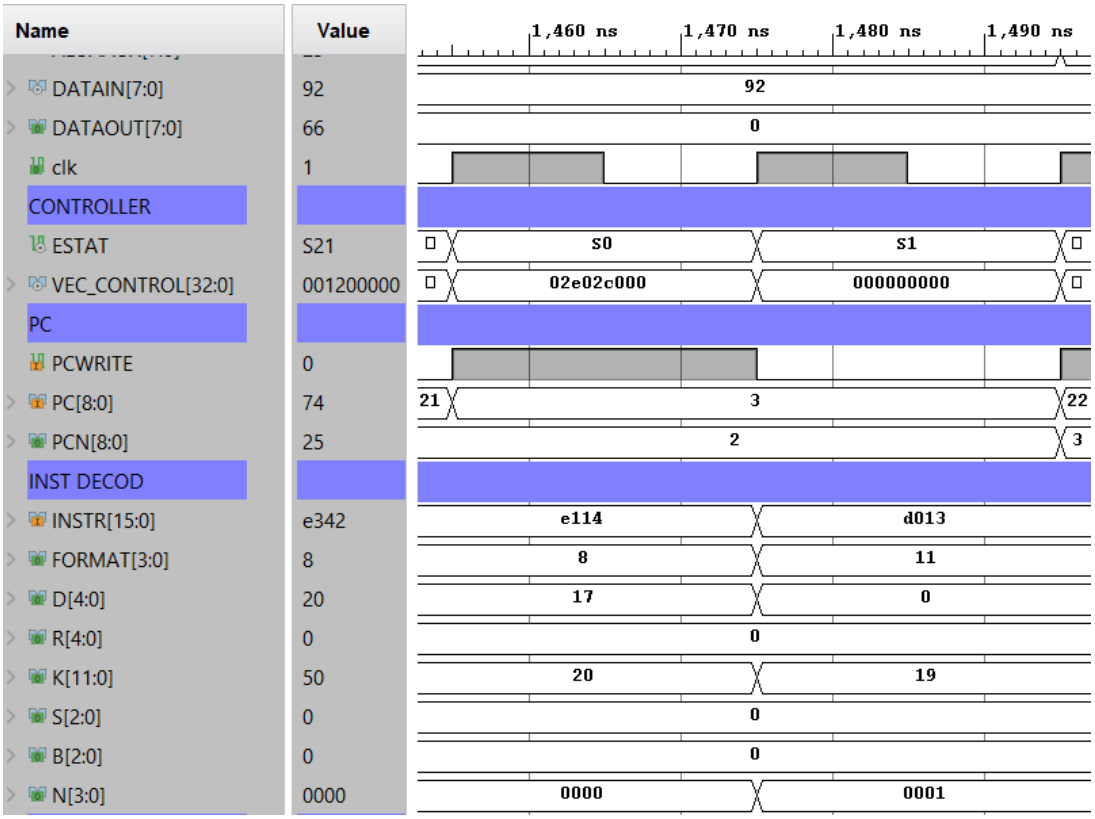


Figura 4.38 Simulació de les operacions de descodificació de la instrucció RCALL

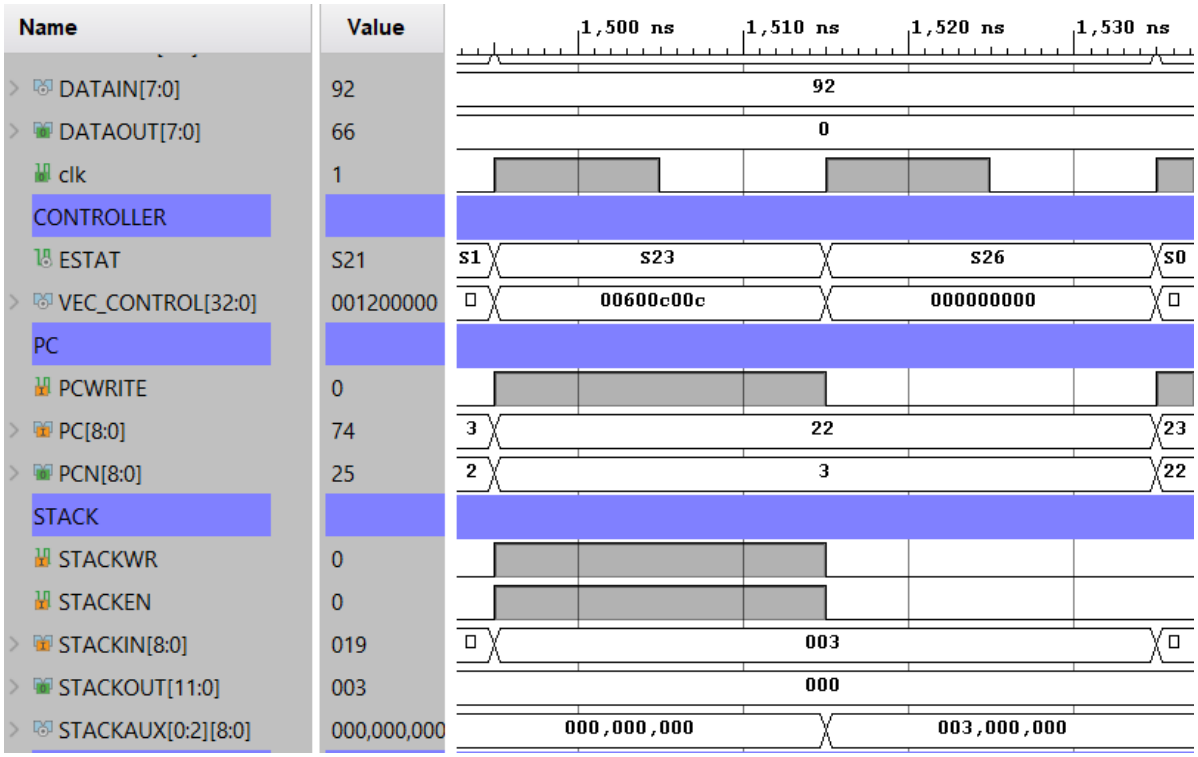


Figura 4.39 Simulació de les operacions de guardar el valor de PC a la memòria Stack

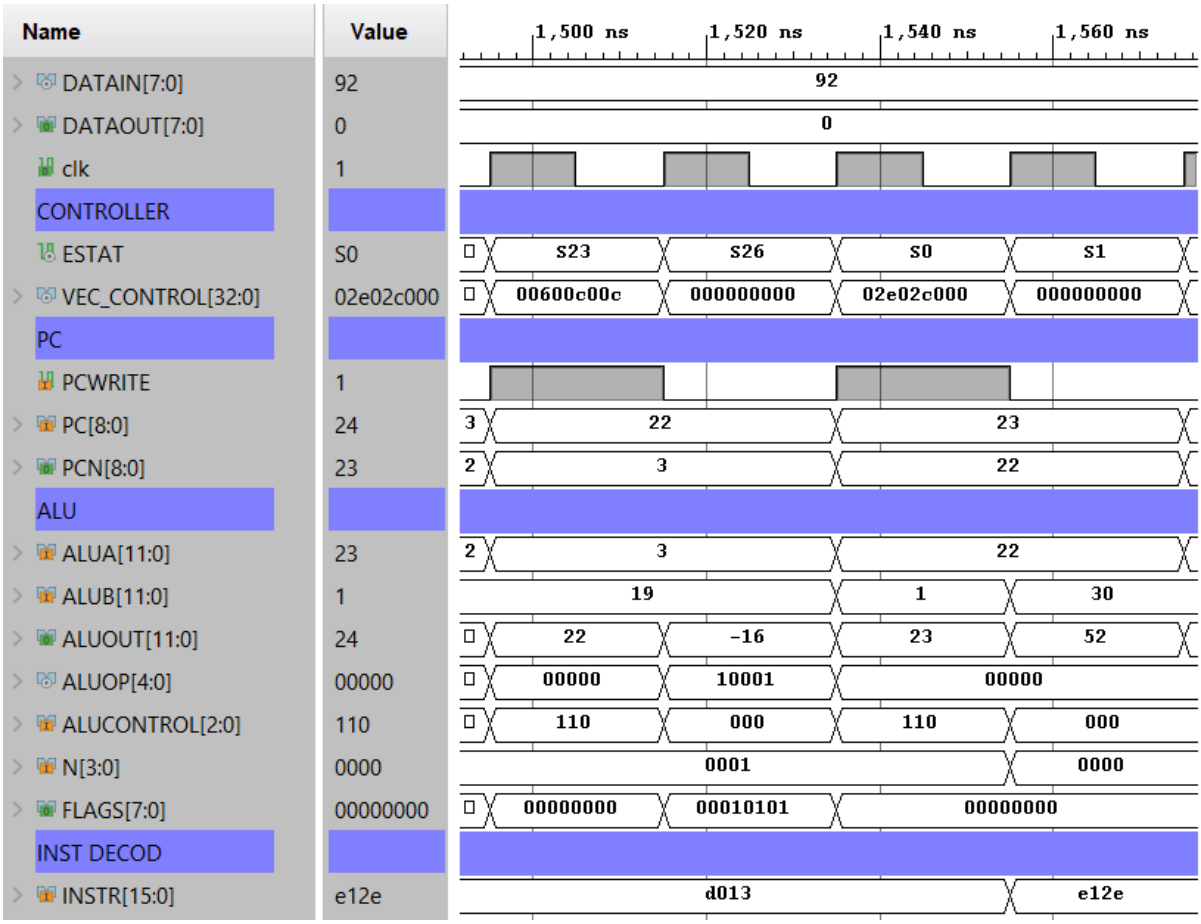


Figura 4.40 Simulació de les operacions de realització del salt i d'actualització de PC

A continuació, a la Figura 4.41, es pot observar que s'executen correctament les instruccions `LDI r18,30; LDI r19, 40; LDI R20, 50; LDI R21, 60,` pels canvis que es generen en els valors dels registres R18, R19, R20 i R21.

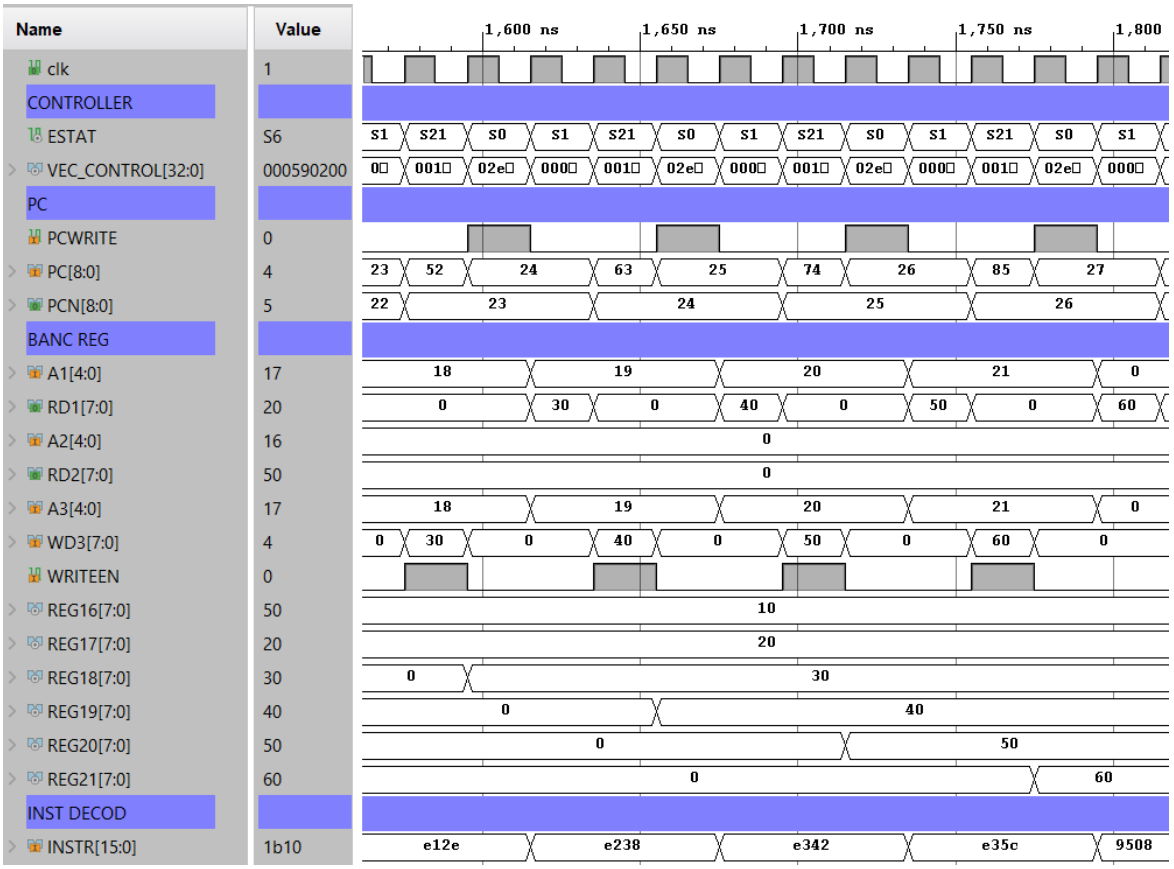


Figura 4.41 Simulació de les instruccions LDI abans del salt de retorn

Una vegada ha finalitzat l'execució de les instruccions, arriba el moment del retorn a l'adreça de PC anterior. Per fer-ho, s'executa la instrucció RCALL. No es mostrarà l'execució de l'estat S1, ja que, com que la instrucció RET no té mes components que la 'n', no és de gaire interès. En la Figura 4.42, en canvi, es pot veure com a l'estat S24 es carrega el valor guardat a la memòria *stack* al PC. Aquesta acció es duu a terme activant el senyal *STACKEN* amb *STACKWR* desactivat. Això fa que el valor superior de la memòria *stack*, 3, es posi a la sortida, *STACKOUT*.

Mitjançant el senyal *PC2SRC* del multiplexor *MUXPC2*, s'escull l'entrada connectada a *STACKOUT*, i es porta a l'entrada del comptador de programa, habilitant la senyal *PCWrite*.

Finalment, s'imposa el nou valor del PC mitjançant l'estat S13. Com es pot veure a la Figura 4.43, el valor desitjat de PC triga un cicle de rellotge en aplicar-se a PC. Això és per la carrega del següent valor de PC que es produeix en cada estat S0. Per això, el processador està preparat per executar la següent instrucció, que en aquest cas és NOP, perquè la resta de la memòria *Flash* és buida. Aquesta situació és remeiable, canviant l'origen de la instrucció que rep el descodificador, de la sortida del registre d'instruccions a agafar la instrucció directament de la sortida de la memòria *Flash*. Això, que es fa mitjançant el multiplexor *MUXINST*, permet el

descodificador rebre la instrucció de l'adreça de PC correcte, de tal forma que *PCN* pren el valor adequat i s'executa la operació desitjada, que es troba a l'adreça 3 de la memòria *Flash*.

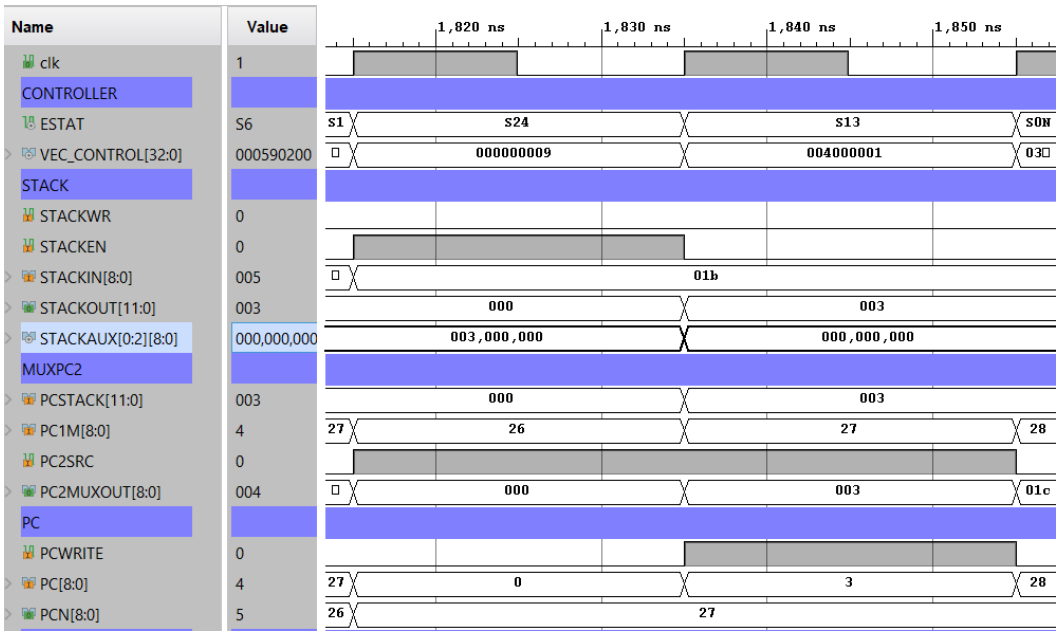


Figura 4.42 Simulació de les operacions de descodificació de la instrucció RET

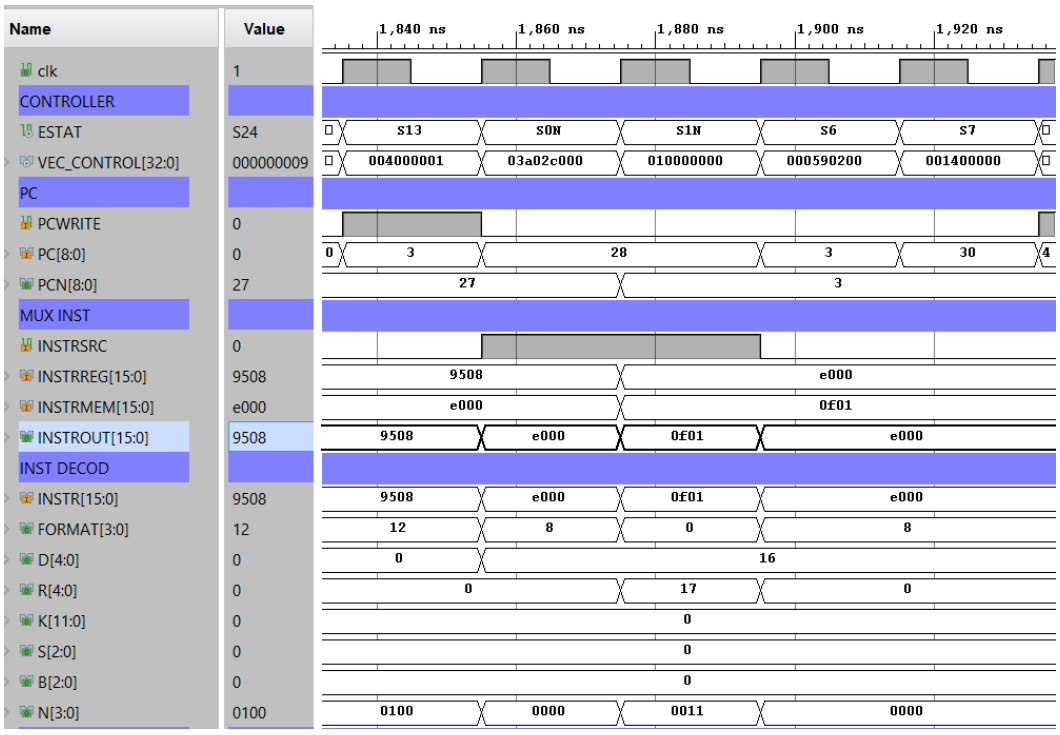


Figura 4.43 Simulació de les operacions del salt de retorn, actualització de PC i execució de la següent instrucció

4.5.2 BRIE

Finalment, s'analitzarà l'execució de la instrucció BRIE, una instrucció de salt condicional, que salta un número fix de instruccions si la condició de salt es confirma. En aquest cas, BRIE significa *Branch if Interrupt Enabled*, és a dir, efectua el salt si el bit I del registre SREG està activat. Aquesta instrucció, de forma similar a les instruccions del Format 8, és part d'un conjunt d'instruccions, el Format 14, que inclou una operació de salt condicional per cada eventualitat de cada bit de *SREG*. La raó per la qual s'ha escollit aquesta instrucció es que prèviament en aquest document s'ha explicat l'execució de la instrucció SEI, que posa a '1' el *flag* I. Per tant, com que aquesta instrucció usa com a condició de salt l'alt nivell de I, per provar el seu funcionament, s'incorporarà aquesta instrucció al programa, fent que salti les següents 5 instruccions.

La instrucció BRIE, com s'ha dit abans, és del Format 14. Això significa que segueix la estructura "1111 0nkk kkkk ksss", on 'n' indica si la condició de salt és positiva (*flag* en nivell alt) o negativa (*flag* en nivell baix), 'k' és la mida del salt i 's' és l'adreça del bit de la condició. Com que es sap que la condició de salt és positiva, que és sobre el *flag* I, que té la posició 7 a *SREG*, i que es vol fer un salt de 5 instruccions, el codi de la instrucció quedarà de la següent forma:

```
BRIE 15;   ""1111 0000 0010 1111"";   0xF02F
```

Per començar a examinar l'execució de la instrucció, s'analitzaran els estats S1 i S25. A S1, com a cada instrucció, es duu a terme la descodificació. Es pot veure a la Figura 4.44 que aquesta es realitza de forma satisfactòria, ja que els valors de 'k', 's' i 'n' es veuen correctament reflectits a les sortides *K, S* i *N*. Una vegada finalitzada la descodificació, el processador passa a l'estat S25, que examina la condició de salt i passa, si es compleix, a l'estat S14, i ni no a l'estat S0. Com es veu a la figura, la condició es compleix, i el processador entra a l'estat S14, on s'efectua el salt.

En aquest salt, no es guarda el valor actual a la memòria *stack* abans de saltar, i per tant la funció RET no tindrà cap ús. En canvi, a l'estat S14, com es pot veure a la Figura 4.45, es suma el valor K del salt al valor actual de PC, mitjançant el senyal *ALUCONTROL*, i a l'estat S10 es carrega al comptador de programa, que pren el valor actualitzat i executa la instrucció pertinent, com es veu a la figura.

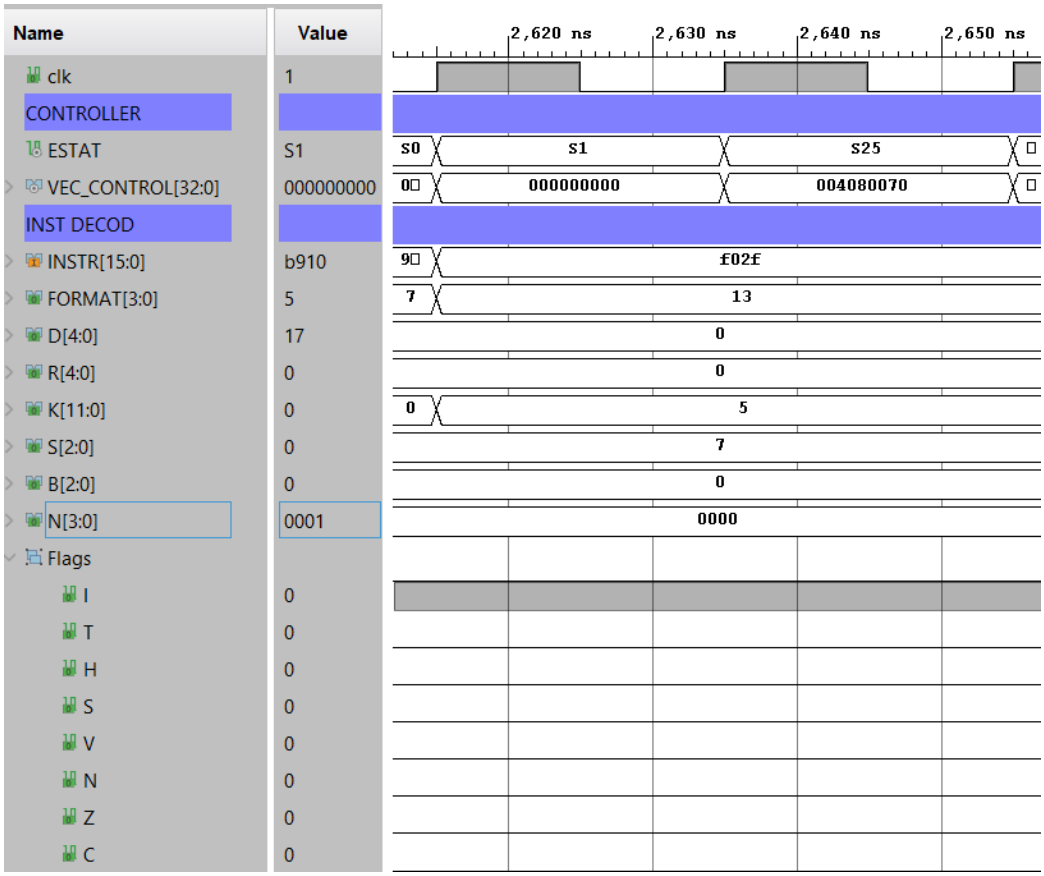


Figura 4.44 Simulació de les operacions de descodificació de la instrucció BRIE i comprovació de la condició de salt, el nivell alt del flag T

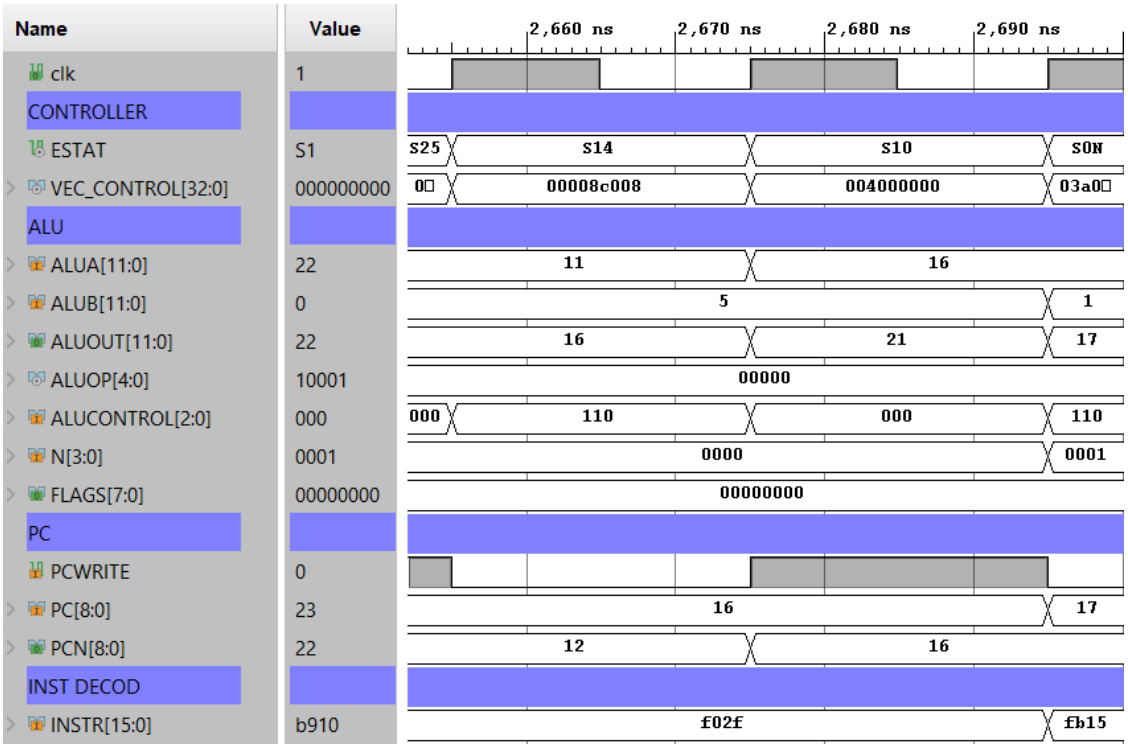


Figura 4.45 Simulació de les operacions de salt i actualització del valor de PC

Capítol 5. Implementació

Havent comprovat que les instruccions s'executen correctament, caldrà fer la implementació del disseny sobre la FPGA. La placa escollida per fer la implementació és la Nexys 4 DDR, una placa de desenvolupament de Digilent, que porta incorporada una FPGA Artix-7 de la marca Xilinx. Aquesta placa també porta incorporada una gran quantitat de perifèrics, com interruptors, botons, llums LED, i ports d'entrada i sortida de dades.

Per comprovar el funcionament implementat del disseny, s'ha escrit un programa, guardat en la memòria *Flash*, que fa la funció d'un comptador. Aquest programa, com es pot observar a continuació, executa instruccions de salt condicional dependent de bit, operacions aritmètiques i transferència de dades. Per això, amb el funcionament d'aquest programa es pot confirmar el funcionament del sistema.

```
---Carrega valor inicial del comptador----
"1110000000000000",     --LDI R16, 0      0
---Posa valor a la sortida, prepara el següent----
"1011100100000000",     --OUT R16           1
"1001010100000011",     --INC R16           2
--Carrega els valors dels registres del comptador-
"1110000001010000",     --LDI R21, 0        3
"1110001001101010",     --LDI R22, 42       4
"1110000000110000",     --LDI R19, 0        5
"1110111101001111",     --LDI R20, 255      6
"1110000000010000",     --LDI R17, 0        7
"1110111100101111",     --LDI R18, 255      8
"0001101100100001",     --SUB R18, R17      9
"1111000000011001",     --BREQ(3)           10
---Si R18 no val ZERO----
"1001010100010011",     --INC R17           11
"1100111111111100",     --RJMP(-4)          12
-----Si R18 val ZERO------
"0000000000000000",             --          13
"1001010100110011",     --INC R19           14
"0001101101000011",     --SUB R20, R19      15
"1111000000010001",     --BREQ(2)           16
---Si R20 no val ZERO----
"1100111111110100",     --RJMP(-12)         17
-----Si R20 val ZERO------
"0000000000000000",             --          18
"1001010101010011",     --INC R21           19
"0001101101100101",     --SUB R22, R21      20
"1111000000010001",     --BREQ(2)           21
---Si R22 no val ZERO----
"1100111111101101",     --RJMP(-19)         22
-----Si R22 val ZERO------
"1100111111101010",     --RJMP(-22)         23
```

El programa del comptador es basa en l'execució d'instruccions senzilles de forma reiterada. Primer, es carrega el valor inicial del cronòmetre a R16, i es mostra en els LEDs mitjançant la instrucció OUT. Acte seguit, s'actualitza el valor del cronòmetre, incrementant el seu valor en un.

Després, es carreguen els valors dels registre que actuaran com a comptadors. S'utilitzen els registres R17, R19 i R21 per anar incrementant el valor i els registres R18, R20 i R22 com a

limitadors, és a dir, per marcar el valor a partir del qual es reiniciïn els valors dels registres comptadors.

La raó per la qual s'utilitzen tres comptadors és perquè el valor màxim que poden tenir aquests registres és 255, perquè és el valor màxim que es pot representar amb 8 bits. Mitjançant la simulació de la Figura 5.1, es comprova que una volta completa (anar del valor 0 al 255) del registre R17 triga aproximadament 93 us en produir-se. Cada vegada que el registre R17 arribi al valor 255, el registre R19 incrementarà el seu valor en 1. Si aquest procediment es repeteix 255 vegades, una volta del registre R19 trigarà 23,75 ms en produir-se. Finalment, quan el registre R19 arribi al valor 255, incrementarà el valor del registre R21. Si el límit del registre R21 també fos 255, faria una volta cada 6 segons, aproximadament. Per això, es limita el registre R21 a 42, modificant el valor del registre R22, i així la volta del registre R21 es farà aproximadament cada 996 ms.

Cada vegada que es finalitzi una volta del registre R21, es mostra el valor del registre R16, incrementat prèviament, pels LEDs de la FPGA. Així, el valor del cronòmetre incrementa cada segon.

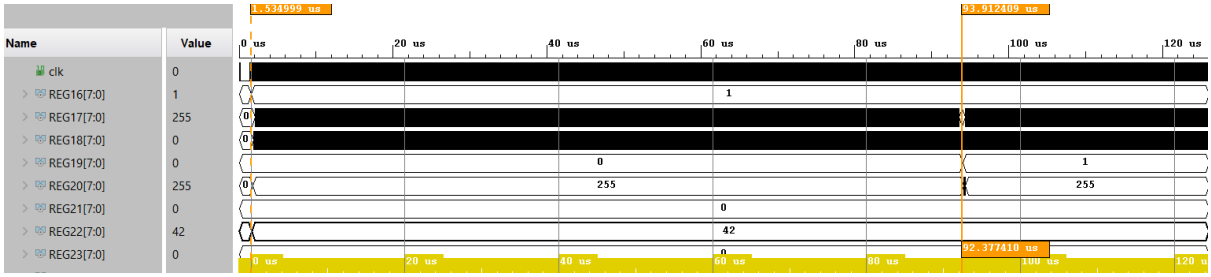


Figura 5.1 Simulació del programa a implementar, que mostra que una volta del comptador R17 triga 92,3 µs, que s'ha arrodonit a 93 µs

El primer pas per poder executar la implementació del disseny és afegir-hi un arxiu de restriccions, que es pot trobar a l'Annex E. Aquest arxiu indica a la placa quins ports dels perifèrics cal connectar als ports del disseny implementat a la FPGA, com el rellotge, el port de dades d'entrada i el de sortida i, en aquest cas, els díodes LED.

Una vegada incorporat l'arxiu de restriccions, s'executa la síntesi del disseny, on es pren el codi VHDL i es representa en forma de portes lògiques. Es creen tots els components (registres, *latch*, biestables, portes lògiques) necessaris pel funcionament del disseny. Si després de la síntesi el programa no mostra cap error, es procedeix a la implementació del disseny. En aquest procés, s'incorpora el disseny sintetitzat a la placa, restringit per les limitacions lògiques, físiques i de temporització d'aquesta.

Una vegada finalitzada la implementació, es pot accedir a diversos informes sobre el rendiment del disseny a la placa. El primer cop que es va implementar el disseny, es va trobar que el disseny no es podia implementar correctament a la placa perquè no es complien els requisits de temporització. Això passa quan la suma del retard d'un camí lògic és superior al període del rellotge que el controla. Per corregir-ho, es pot dividir aquest camí lògic mitjançant l'ús de registres síncrons, o es pot reduir la freqüència del rellotge perquè la informació es pugui transmetre respectant els requisits de temporització.

Per augmentar el temps del període del rellotge, s'ha usat una IP de Vivado, un bloc predissenyat on només cal modificar certs atributs per ajustar-lo a les necessitats del projecte. Aquest bloc IP, anomenat *CLK_Wizard*, permet canviar la freqüència del rellotge multiplicant-lo per un número racional. En aquest cas, es multiplica la freqüència per 0.8, que significa que el rellotge passa de tenir una freqüència de 100MHz a 50MHz, i per tant el període del rellotge incrementa a 20 ns. El canvi de freqüència del rellotge no te cap efecte en l'execució simulada de les instruccions, com es pot veure a la Figura 5.2, on *clk_in* és el senyal del rellotge de 100 MHz que produeix la FPGA, i *clk* és el rellotge que dirigeix el funcionament del processador, que té una freqüència de 50 MHz.

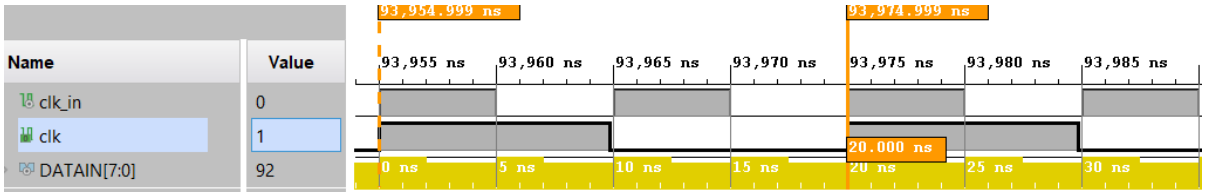


Figura 5.2 Simulació dels dos senyals de rellotge que te el sistema

Un efecte que té la incorporació d'aquest bloc és que triga un cert temps en començar a funcionar, com es pot apreciar a la Figura 5.3. Com que el rellotge de la sortida del *CLK_Wizard* dirigeix el funcionament del processador, el retard del bloc genera un retard en l'activació de tot el sistema de 1350 ns.

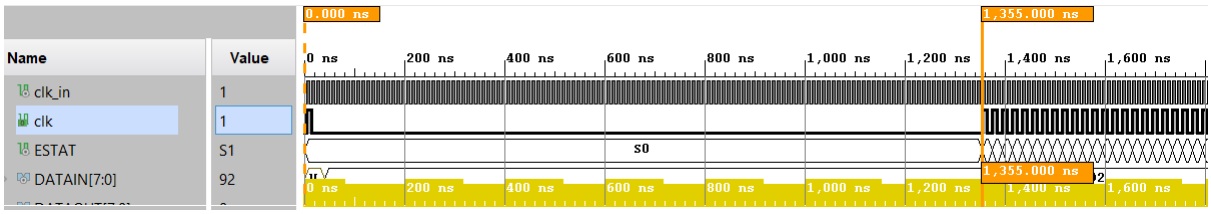


Figura 5.3 Simulació del retard que genera el bloc CLK_Wizard en iniciar el seu funcionament

Amb el bloc incorporat, però, ja no hi ha problemes amb la implementació. L'informe de temporització de la implementació indica que per cap dels 3 origen de retard, el temps de *setup*, el temps de *hold*, i l'ample de pols, es genera un *slack* negatiu.

El temps de *setup* és l'interval de temps necessari en que el valor d'un senyal ha de ser estable abans de que canviï el valor del rellotge. El temps de *hold* és l'interval de temps després del canvi del rellotge en el que s'ha de mantenir fix el valor d'un senyal. El retard d'amplada de pols fa referencia a les limitacions que suposen els retards mencionats anteriorment respecte l'amplada de pols a nivell alt, a nivell baix i total.

El que es vol evitar és tenir un *slack*, o retard, negatiu. Si el *slack* al final d'un camí de transmissió d'informació, anomenat *endpoint* en la taula de la Figura 5.4, és negatiu, aquest *endpoint* falla, és a dir, pot generar errors en forma de *glitches*. Com es pot veure a la figura, però, no és el cas, donat que o hi ha cap *endpoint* que falli.

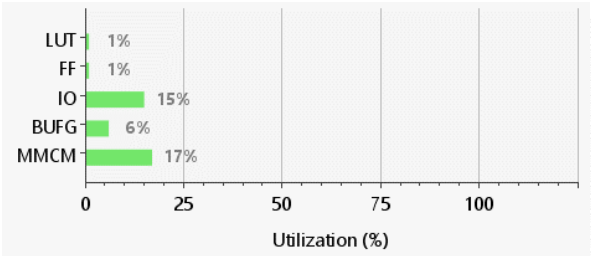
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5,061 ns	Worst Hold Slack (WHS):	0,047 ns	Worst Pulse Width Slack (WPWS):	3,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1994	Total Number of Endpoints:	1994	Total Number of Endpoints:	968

All user specified timing constraints are met.

Figura 5.4 Taula de resum de l'informe de temporització de la implementació del disseny

A part dels requisits de temporització, també cal observar la quantitat de recursos que consumirà la implementació d'aquest sistema en la FPGA. L'informe post-implementació ofereix informació en forma de gràfics i taules sobre l'ús dels recursos disponibles en la implementació del disseny. Observant les figures de la Figura 5.5, es pot dir que el disseny utilitza un percentatge molt baix dels recursos disponibles. Cal tenir en compte que, incloent el MMCM, que significa *Mixed-Mode Clock Manager* i és el mòdul afegit per poder complir els requisits de temporització, s'han fet servir un percentatge de recursos menor al 20%, així que és un disseny que exigeix pocs recursos a la FPGA. La Figura 5.6 mostra el mapa de la FPGA on es troben il·luminades les cel·les lògiques usades en la implementació del disseny.



Resource	Utilization	Available	Utilization %
LUT	911	63400	1.44
FF	962	126800	0.76
IO	32	210	15.24
BUFG	2	32	6.25
MMCM	1	6	16.67

Figura 5.5 Gràfica d'ús de recursos de al FPGA en la implementació del disseny (dreta) i Taula on es desglossa la quantitat de recursos usats respecte el total de recursos disponible

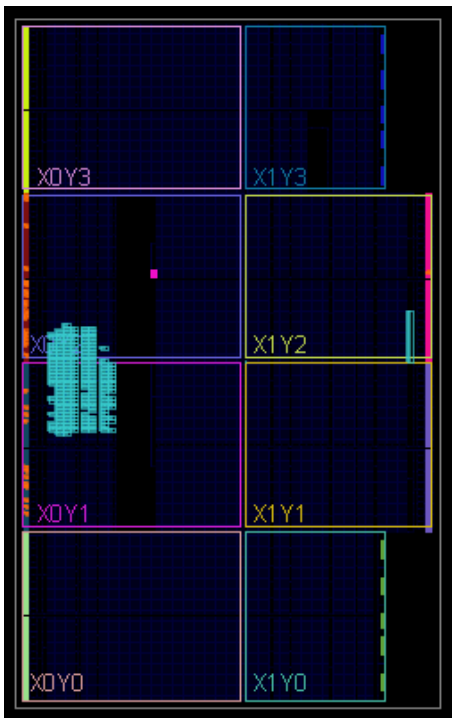


Figura 5.6 Mapa de la les cel·les lògiques usades en la implementació del disseny en la FPGA

Finalment, cal observar la potencia consumida pel disseny a la placa. De nou, com passa amb el consum de recursos, el consum de potencia ve donat en gran mesura per l'ús del MMCM, el bloc predissenyat per Vivado que modifica la freqüència del rellotge. De totes formes, la temperatura d'unió és molt baixa, i la potencia dissipada pel chip si no s'hagués incorporat *CLK_Wizard* seria de 92 mW, que és un valor de consum molt baix.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.186 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**25,8°C**
Thermal Margin:	59,2°C (12,8 W)
Effective ϑJA:	4,6°C/W
Power supplied to off-chip devices:	0 W

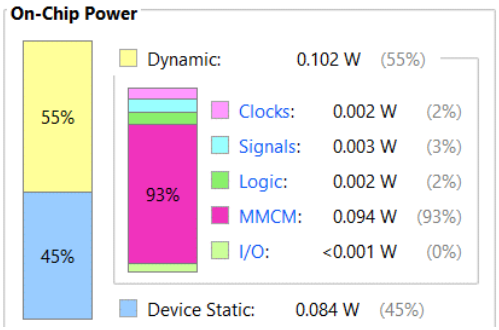


Figura 5.7 Resum de l'informe de consum de potència del disseny implementat

Una vegada consultats els informes de implementació, toca generar el *BitStream* i programar el disseny a la FPGA. Els resultats, implementats a la placa, son els observables en les Figures 5.8, 5.9, 5.10 i 5.11. Com es pot veure en les Figures 5.8 i 5.9, que són captures de pantalla d'una gravació en vídeo del resultat de la implementació, alhora que avança el cronòmetre del mòbil, el valor mostrat en els LEDs de la FPGA va mostrant el valor del temps transcorregut.

Figura 5.8 Resultat de la implementació en el segon 1,35

Figura 5.9 Resultat de la implementació en el segon 2,33

Per demostrar que el cronòmetre funciona de forma correcte, s'han pres les imatges mostrades a les Figures 5.10 i 5.11, que mostren com el canvi de valor en el LED es produeix alhora que canvia el valor del cronòmetre.

Figura 5.10 Resultat de la implementació en el segon 2,99

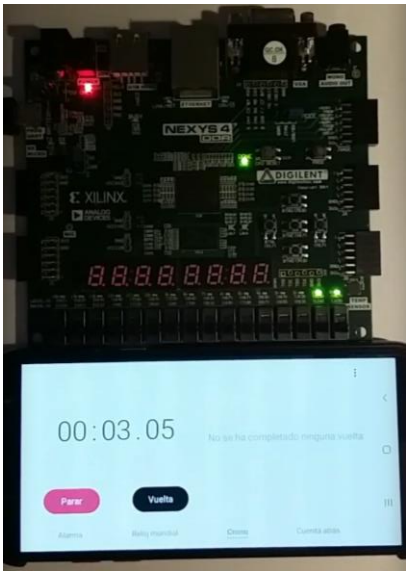


Figura 5.11 Resultat de la implementació en el segon 3,05

Anàlisi d'impacte ambiental

En el desenvolupament d'aquest projecte, l'impacte ambiental ha estat molt reduït, ja que només s'ha consumit electricitat en els aparells electrònics utilitzats. Per tant, el desenvolupament d'aquest treball no ha generat cap residu sòlid ni contaminant, ja que tots els instruments utilitzats en el seu desenvolupament han estat reutilitzats i podran ser reutilitzats de nou.

De fet, aquest treball pot arribar a evitar un futur consum de recursos naturals, ja que estalviarà temps de desenvolupament en projectes futurs que puguin necessitar la tecnologia desenvolupada en aquest treball. Per tot això, es pot assegurar que el balanç ambiental d'aquest treball és positiu.

Pressupost

El cost de desenvolupament d'aquest projecte es basa molt en les hores de feina que comporta el disseny del microprocessador. L'opció òptima de distribució de feina seria encarregar les diferents tasques que inclou el desenvolupament del treball a diferents persones que conformen un equip, encara que les desenvolupi després una mateixa persona. Per tant, és necessari el disseny en tasques específiques i marcar una durada per cada una.

Taula 0.1 Desglossament de les hores consumides pel projecte segons les tasques que ha comportat

	Tasques	**Hores**
1	Definició del projecte	10
2	Recaptació de la bibliografia i estudi previ de la matèria	60
3	Disseny del microprocessador	110
4	Programació en VHDL del microprocessador	150
5	Simulació del sistema	100
6	Implementació del sistema sobre FPGA	80
7	Redacció de la Memòria del projecte	90
	TOTAL	**600**

A continuació, es dividiran les tasques entre els responsables d'executar-les, ja que les diferents feines es remuneraran de forma diferent, com es pot veure a la Taula 7.2

Taula 0.2 Divisió de les tasques entre els membres de l'equip del projecte

Càrrec	**Sou**	**Tasques**	**Hores de treball**
Coordinador del projecte	25 €/h	1, 2, 3, 7	270
Programador	20 €/h	4,5,6	330

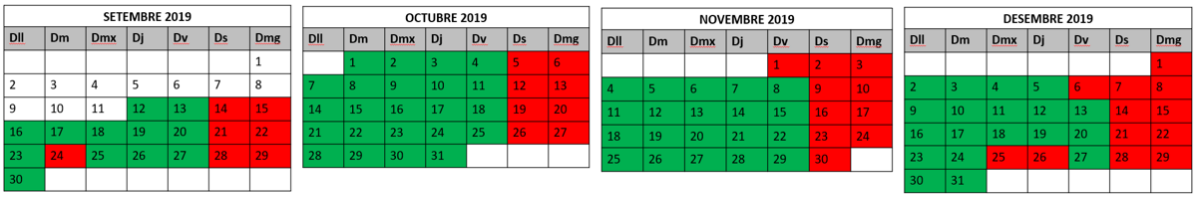
El material fet servir en el desenvolupament del projecte serà:

Taula 0.3 Material requerit pel desenvolupament del projecte

Material	**Quantitat**
Ordinador de Sobretaula	1
Ordinador Portàtil	1
Placa FPGA Nexys 4	1
Microsoft Office	1
Vivado 2019	1

Finalment, el desglossament total dels costos es pot trobar a la taula 7.4. Cal, per interpretar correctament la taula, tenir en compte vàries qüestions. La primera, que el cost atribuïble al projecte del material ve donat pel percentatge de temps de vida del material que es consumirà durant la durada del projecte. Com que la durada total del treball és de 600h, assumint que cada dia laborable es treballa 8 hores, es necessitaran 75 dies laborables per finalitzar-lo.

Si el projecte s'inicia el 12 de setembre de 2019, es trigarà quatre mesos, com es pot comprovar en la Figura 7.1, en finalitzar el treball. Per tant, el projecte consumirà 4 mesos de la vida útil del material utilitzat en el seu desenvolupament.



SETEMBRE 2019						
Dll	Dm	Dmx	Dj	Dv	Ds	Dmg
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30						

OCTUBRE 2019						
Dll	Dm	Dmx	Dj	Dv	Ds	Dmg
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

NOVEMBRE 2019						
Dll	Dm	Dmx	Dj	Dv	Ds	Dmg
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

DESEMBRE 2019						
Dll	Dm	Dmx	Dj	Dv	Ds	Dmg
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Figura 0.1 Planificació dels dies laborables, en verd, que durarà el desenvolupament del projecte

La segona qüestió a considerar en la comprensió de la Taula 7.5 és que el cost dels sous del treball executat ve donat per la divisió de tasques de la Taula 7.2. Les tasques pertanyents al coordinador del projecte es remuneren a 25 €/hora, mentre que al programador se li retribueixen 20 €/hora per les tasques que se li atribueixen. Així, multiplicant el sou vinculat al càrrec per les hores treballades en la realització de les tasques atribuïdes, resulten en els costos reflectits en la Taula 7.5.

Finalment, el cost del consum elèctric vinculat al desenvolupament del projecte és la suma dels consums elèctrics del material usat, i una aproximació del consum de la instal·lació on es duu a terme el projecte, comptant que té 4 llums i un sistema de climatització. El desglossament d'aquests consums es pot veure a la Taula 7.4.

Taula 0.4 Consum elèctric del projecte

Concepte	**Consum**
Ordinador de Sobretaula	200 W
Ordinador Portàtil	150 W
Placa FPGA Nexys 4	10 W
Il·luminació	24 W
Climatització	2000 W
TOTAL	2384 W

Taula 0.5 Desglossament dels costos d'execució del projecte

Concepte	**Quantitat**	**Cost unitat**	**Cost atribuïble al projecte**	**Total**
Ordinador de sobretaula	1	709,00 €	4 mesos / 4 anys de vida útil = 8,3%	59,08 €
Ordinador portàtil	1	879,00 €	4 mesos / 4 anys de vida útil = 8,3%	73,24 €
Nexys 4	1	235,69 €	4 mesos / 4 anys de vida útil = 8,3%	19,64 €
Microsoft Office	1	40,85 €	4 mesos / 2 anys de vida útil = 16,6%	6,81 €
Vivado 2019	1	0 €	-	0 €
Total material				**158,77 €**
Coordinador del projecte	1	6.750,00 €	100%	6.750,00 €
Programadors	1	6.600,00 €	100%	6.600,00 €
Total sous				**13.350,00 €**
Consum elèctric	2,384 kW/h x 600 h = 1430 kW	0,16 €	100%	228,86 €
Total				**13.737,63 €**

Com es pot veure, gran part del cost del projecte vindria donat pels sous dels treballadors, no pel consum material que exigeix el projecte en si, sempre i quan els materials requerits s'amortitzin amb altres projectes.

Conclusions

Al començament d'aquest treball, s'han marcat uns objectius generals clars: el disseny i la implementació en FPGA d'una microarquitectura funcional. Havent acabat el treball, es pot dir que els objectius s'han complert amb èxit. S'ha dissenyat la microarquitectura d'un microprocessador, basant-se en l'arquitectura del *ATtiny11* de la marca AVR, que pot executar correctament més de 90 instruccions diferents.

També s'havia dit a l'inici del projecte que el microprocessador no havia de ser competitiu amb els altres del mercat en velocitat, consum o poder computacional, sinó que es volia dissenyar una arquitectura flexible i fàcilment modificable, per poder adequar-se a les necessitats de projectes futurs.

Per això, el projecte s'ha dissenyat de la forma que s'ha fet, incloent elements que no tenien ús en el sistema actual, però que deixen la porta oberta a la seva implementació futura. Un exemple d'això son les interrupcions, que no s'han pogut incorporar en aquest projecte per manca de temps, però poden ser afegides sense massa problema, ja que, per exemple, el registre d'estat, *SREG*, inclou el senyal d'interrupcions, i varies instruccions el manipulen o l'utilitzen com a condicionant.

La raó per la que es poden afegir elements segons les necessitats futures que es presentin és pel mètode de disseny que s'ha aplicat. Com que el disseny del microprocessador s'ha fet de forma molt modular, dissenyant cada component per separat, ajuntant-los en el *datapath* on només es senyalen les connexions entre components, si es volgués afegir components nous només caldria incorporar l'arxiu amb el disseny del component en qüestió i programar les seves connexions al *datapath*.

Treballs Futurs

En futures línies de treball, potser és necessari expandir la memòria. En el cas de la memòria *Flash*, el comptador de programa es pot modificar, i fins als 1024 bytes de memòria les instruccions de salt RJMP i RCALL funcionarien correctament. En quant a l'expansió de la memòria RAM, el microcontrolador té capacitat per adreçar memòries de fins a 65.5k bytes, usant els registres ZHi i ZLo concatenats per tenir un vector d'adreces de 16 bits.

Si se li afegeixen perifèrics al sistema, per poder disminuir el consum elèctric del chip s'hauria de modificar la funció SLEEP, que fins ara només inhabilita els registres interns, per que inhabilités també aquests perifèrics, afegint els senyals de *enable* d'aquests elements a la Unitat de Control del microprocessador i desactivant-los en l'estat S26.

Una cosa que suposaria una millora al disseny d'aquest microprocessador seria la introducció de ports I/O, capaços de rebre i treure dades pel mateix port. Això permetria tenir més ports de comunicació amb l'exterior i connectar més perifèrics. Amb més ports, però, caldria modificar les instruccions IN i OUT, perquè el seu component A, l'adreça del port pel que s'ha de transmetre la informació, és fixe perquè no hi ha mes d'un port de comunicació.

Finalment, una línia de treball futura sobre el projecte de molta importància seria la resolució de l'incompliment de les restriccions de temporització del microprocessador a 100 MHz. Per fer-ho, caldria analitzar cada camí lògic que es pren en el disseny implementat, i fer els canvis pertinents per reduir el retard que es genera en aquests, afegint registres o *latches.* Això permetria treballar a una freqüència més alta, que seria positiu en si mateix, però a més significaria que no s'hauria de fer servir el bloc IP *CLK_Wizard*, que com s'ha vist a l'apartat d'implementació és el màxim contribuent al consum de potència del microprocessador.

Bibliografia

[1] *Harris, D.M. & Harris, S.L.* ***'Digital Design and Computer Architecture'***. 2nd ed. Amsterdam. Editorial Elseiver, cop. 2013.

[2] *Austin, T & Tanenbaum, A.S.* ***'Structured Computer Organization'***. 6th ed. Upper Saddle River, NJ. Editorial Pearson, cop. 2013.

[3] *Atmel Corporation.* ***'AVR Instruction Set Manual'***. Rev.0856L - 11/2016. San Diego, CA.

[4] *Atmel Corporation.* ***'ATtiny11/12 Datasheet'***. Rev.1006D - 07/2003. San Diego, CA.

[5] *Intel.* ***'Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture'***. 09/2016.

[6] *Intel.* ***'Intel 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference'***. 09/2016.

[7] *ARM.* ***'ARMv7-M Architecture Reference Manual'***. Issue E.b - 12/2014. Cambridge.

[8] *Atmel Corporation.* ***'ATmega328P Datasheet'***. Rev.7810D - 01/15. San Diego, CA.

[9] *Xilinx.* ***'Vivado Design Suite User Guide: Using Constraints'***. UG903 (v2012.2) September 4, 2012.

[10] *Xilinx.* ***'Vivado Design Suite User Guide: Implementation'.*** UG904 (v2019.2) December 18, 2019.

[11] *Atmel Corporation.* ***'AVR Instruction Encoding'***. 02/2019

[12] *Digilent Inc.* ***'Nexys 4 DDR [Reference.Digilentinc]'.*** Consultat Desembre 2019

Annex A : Formats d'instruccions

A.1 Format 1: Adreçament Directe, 2 Registres

00nn	nnrd	dddd	rrrr

nnnn	Instrucció	Instruccions Equivalents
0001	CPC	
0010	SBC	
0011	ADD	LSL = ADD(Rd,Rd)
0100	CPSE	
0101	CP	
0110	SUB	
0111	ADC	ROL = ADC(Rd,Rd)
1000	AND	TST = AND(Rd,Rd)
1001	EOR	CLR = EOR(Rd,Rd)
1010	OR	
1011	MOV	

A.2 Format 2: Adreçament Directe, 1 Registre

1001	010d	dddd	nnnn

nnnn	Instrucció
0000	COM
0001	NEG
0010	SWAP
0011	INC
0101	ASR
0110	LSR
0111	ROR
1010	DEC

A.3 Format 3: Adreçament Directe, Registre i Constant

01nn	KKKK	dddd	KKKK

nn	Instrucció	Instruccions Equivalents
00	SBCI	
01	SUBI	
10	ORI	SBR = ORI(Rd,K)
11	ANDI	CBR = ANDI(Rd, 0xFF

A.4 Format 4: Adreçament Indirecte, SRAM i REG Z

10q0	qqnd	dddd	nqqq

nn	Instrucció
00	LD
01	ST

A.5 Format 5: Adreçament Directe, Registre (Adreça de Bit)

1111	1nnd	dddd	0bbb

nn	Instrucció
00	BLD
01	BST
10	SBRC
11	SBRS

A.6 Format 6: Adreçament Directe, Registre I/O

1011	nAAd	dddd	AAAA

N	Instrucció
0	IN
1	OUT

A.7 Format 7: Adreçament Directe, Registre I/O (Adreça de Bit)

1001	10nn	AAAA	Abbb

nn	Instrucció
00	CBI
01	SBIC
10	SBI
11	SBIS

A.8 Format 8: Adreçament Directe, Registre I/O SREG

1001	0100	nsss	1000

n	Instrucció
0	SET BIT
1	CLR BIT

sss	Flag
000	C
001	Z
010	N
011	V
100	S
101	H
110	T
111	I

A.9 Format 9: Adreçament Directe, Constant (K) a Registre (Rd)

LDI:

1110	KKKK	dddd	KKKK

SER: LDI(Rd, 0Xff)

(Format 10) CPI :

0011	KKKK	dddd	KKKK

A.11 Format 11: Adreçament Indirecte, Memòria del Programa

LPM:

1001	000d	dddd	0100

A.12 Format 12: Control de Transferència Relatiu Incondicional

110n	KKKK	KKKK	KKKK

n	Instrucció
0	RJMP
1	RCALL

A.13 Format 13: Control de Transferència Indirecte

1001	010n	000n	100n

nnn	Instrucció
0	RET
1	RETI

A.14 Format 14: Control de Transferència Relatiu Condicional

1111	0nKK	KKKK	Ksss

n	Instrucció
0	Break if SET
1	Break if Clear

sss	Flag
000	C
001	Z
010	N
011	V
100	S
101	H
110	T
111	I

A.15 Format 15: Control de MCU

1001	0101	10nn	1000

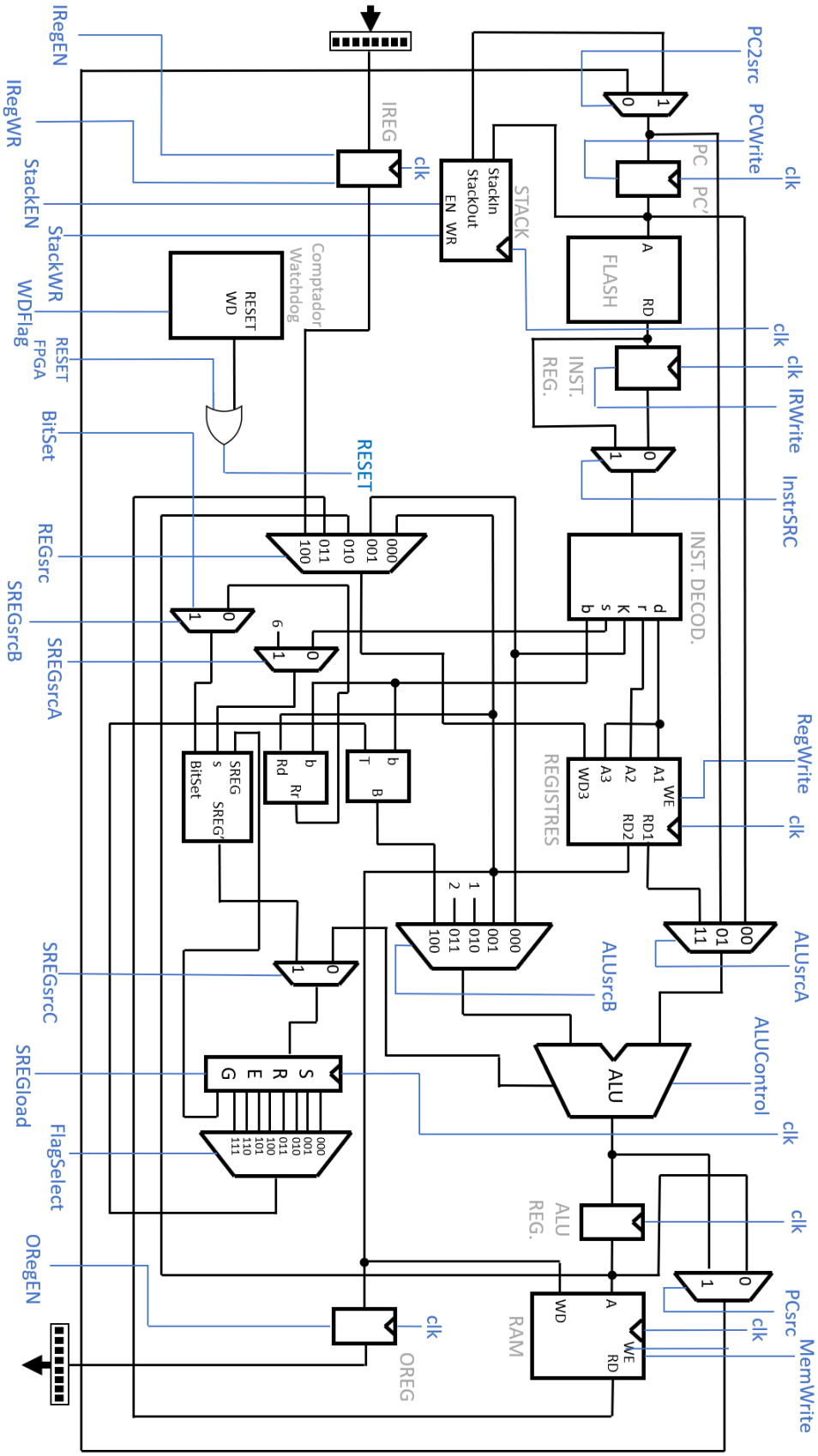
nn	Instrucció
00	SLEEP
10	WDR

A.16 Format 16: Cap operació

NOP:

0000	0000	0000	0000

Annex B: Datapath



Annex C: Taula de Senyals de Control

Estat	IREGEN	IREGWR	OREGEN	FLAGWD	InstrSRC	IRWrite	PCWrite	PCSRC	RegWrite	RegSRC	ALUSRCA	ALUSRCB	ALUOp
S0	0	0	0	1	0	1	1	1	0	000	00	010	110
S1	0	0	0	0	0	0	0	0	0	000	00	000	000
S2	0	0	0	0	0	0	0	0	0	000	11	000	011
S3	0	0	0	0	0	0	0	0	0	000	00	000	000
S4	0	0	0	0	0	0	0	0	1	011	00	000	000
S5	0	0	0	0	0	0	0	0	0	000	00	000	000
S6	0	0	0	0	0	0	0	0	0	010	11	001	000
S7	0	0	0	0	0	0	0	0	1	010	00	000	000
S8	0	0	0	0	0	0	0	0	1	000	00	000	000
S9	0	0	0	0	0	0	0	0	0	000	01	011	110
S10	0	0	0	0	0	0	1	0	0	000	00	000	000
S11	0	0	0	0	0	0	0	0	0	000	11	000	001
S12	0	0	0	0	0	0	0	0	0	000	11	000	010
S13	0	0	0	0	0	0	1	0	0	000	00	000	000
S14	0	0	0	0	0	0	0	0	0	000	01	000	110
S15	0	0	0	0	0	0	0	0	0	000	11	100	100
S16	0	0	0	0	0	0	0	0	0	000	11	100	101
S17	0	0	0	0	0	0	0	0	0	000	00	000	000
S18	0	0	0	0	0	0	0	0	0	000	00	000	000
S19	1	0	0	0	0	0	0	0	1	100	00	000	000
S20	0	0	0	0	0	0	0	0	0	000	00	000	000
S21	0	0	0	0	0	0	0	0	1	001	00	000	000
S22	0	0	0	0	0	0	1	0	0	000	00	000	110
S23	0	0	0	0	0	0	1	1	0	000	00	000	110
S24	0	0	0	0	0	0	0	0	0	000	00	000	000
S25	0	0	0	0	0	0	1	0	0	000	01	000	000
S26	0	0	0	0	0	0	0	0	0	000	00	000	000
S27	0	0	1	0	0	0	0	0	0	000	00	000	000
S28	1	1	0	0	0	0	0	0	0	0	0	0	0
S0n	0	0	0	1	1	1	0	1	0	000	00	010	110
S1n	0	0	0	0	1	0	0	0	0	000	00	000	000
	32	31	30	29	28	27	26	25	24	23:21	20:19	18:16	15:13

SREGSRCA	SREGSRCB	SREGSRCC	SREGLoad	BitSet	MemWrite	FlagSelect	StackEN	StackWR	RAMEN	PC2SRC
0	0	0	0	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	1	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	0	0	1	000	0	0	1	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	0	0	0	110	0	0	0	1
0	0	0	0	0	0	000	1	0	0	0
0	0	0	1	0	0	110	0	0	0	0
0	0	0	1	0	0	110	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
1	0	1	1	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	1	1	1	N	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	0	0	0	000	1	0	0	0
0	0	0	0	0	0	000	1	1	0	0
0	0	0	0	0	0	000	1	0	0	1
0	0	0	0	0	0	SSS	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
0	0	00	0	0	0	0	0	0	0	0
0	0	0	1	0	0	000	0	0	0	0
0	0	0	0	0	0	000	0	0	0	0
12	11	10	9	8	7	6:4	3	2	1	0

Annex D: Codi VHDL

D.1 Comptador de Programa (*PCREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4.
5. ENTITY PCREG IS
6.     PORT (  CLK : IN STD_LOGIC;
7.             RESET : IN STD_LOGIC;
8.             RESETWD: IN STD_LOGIC;
9.             PCWRITE : IN STD_LOGIC;
10.               PC : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
11.               PCN : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
12.  END PCREG;
13.
14.  ARCHITECTURE BEHAVIORAL OF PCREG IS
15.  SIGNAL PCNAUX : STD_LOGIC_VECTOR(8 DOWNTO 0) := "000000000";
16.
17.  BEGIN
18.
19.  PROCESS (CLK)
20.     BEGIN
21.       IF RISING_EDGE(CLK) THEN
22.
23.          IF RESET='1' OR RESETWD = '1' THEN
24.              PCNAUX <= (OTHERS=>'0');
25.          ELSE
26.              IF PCWRITE='1' THEN
27.              PCNAUX <= PC;
28.              END IF;
29.          END IF;
30.          PCN <= PCNAUX;
31.       END IF;
32.   END PROCESS;
33.
34.  END BEHAVIORAL;
```

D.2 Memòria *Flash* (*FLASHMEM*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3.
4. ENTITY FLASHMEM IS
5.     PORT (
6.         A: IN STD_LOGIC_VECTOR(8 DOWNTO 0);
7.         SPO: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
8. END FLASHMEM;
9.
10.   ARCHITECTURE BEHAVIORAL OF FLASHMEM IS
11.   TYPE ROM_512_X_16 IS ARRAY(0 TO 511) OF STD_LOGIC_VECTOR(15 DOWNTO
   0);
12.   SIGNAL MEM : ROM_512_X_16 := (
13.   --PROGRAMA DE PROVES
14.   ------------------------------------------------------------------
15.   --"1110000000001010", --LDI R16 <- 10      E00A        000
16.   --"1110000100010100", --LDI R17 <- 20      E114
17.   --"1101000000010011", --RCALL (19)         C013
18.   --"0000111100000001", --ADD R16, R17       0F01        024
19.   --"0001101100010000", --SUB R17, R16       1B10        026
20.   --"0100000000001111", --SBCI R16, 15       400F
21.   --"1001010100010001", --NEG R17            9511        039
22.   --"1001010100000011", --INC R16            9503        040
23.   --"0010001100000001", --AND R16, R17       2301        028
24.   --"1110011000011001", --LDI R17 <- -75      E114        0
25.   --"1001010001111000", --SEI                 9478
26.   --"1111000000101111", --BRIE(5)            F02F
27.   --"1001010100010111", --ROR R17            9517
28.   --"1001010100010111", --ROR R17            9517
29.   --"1001010100010111", --ROR R17            9517
30.   --"1001010100010111", --ROR R17            9517
31.   --"1111101100010101", --BST R17, B5         FB15
32.   --"1111100100000110", --BLD R16, B6        F863
33.
34.   --"1110000011100011", --LDI RZLo <- 3      E0E3        020
35.   --"1000001100000000", --ST R16             8300        021
36.   --"1000000100010000", --LD R17             8110        023
37.   --"1011000100000000", --IN R16             B100
38.   --"1011100100010000", --OUT R17            B910
39.
40.   --"1110000100101110", --LDI R18 <- 30     E12E        002
41.   --"1110001000111000", --LDI R19 <- 40     E238        003
42.   --"1110001101000010", --LDI R20 <- 50     E342        004
43.   --"1110001101011100", --LDI R21 <- 60     E35C
44.
45.   --"1001010100001000", --RET               9508
46.
47.   OTHERS=>(OTHERS=>('0')));
48.
49.   BEGIN
50.
51.   SPO <= MEM(TO_INTEGER(UNSIGNED(A)));
52.
53.
54.   END BEHAVIORAL;
```

D.3 Registre d'instruccions (*INSTREG)*

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4.
5.
6. ENTITY INSTREG IS
7.     PORT (  CLK : IN STD_LOGIC;
8.             RESET : IN STD_LOGIC;
9.             RESETWD: IN STD_LOGIC;
10.               IRWRITE : IN STD_LOGIC;
11.               SPO : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
12.               INSTR : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
13.   END INSTREG;
14.
15.   ARCHITECTURE BEHAVIORAL OF INSTREG IS
16.   SIGNAL INSTRAUX : STD_LOGIC_VECTOR (15 DOWNTO 0);
17.   BEGIN
18.   PROCESS (CLK)
19.      BEGIN
20.        IF RISING_EDGE(CLK) THEN
21.           IF RESET='1' OR RESETWD = '1' THEN
22.               INSTRAUX <= (OTHERS=>'0');
23.           ELSIF IRWRITE='1' THEN
24.               INSTRAUX <= SPO;
25.           END IF;
26.        END IF;
27.   END PROCESS;
28.   INSTR <= INSTRAUX;
29.   END BEHAVIORAL;
```

D.4 Multiplexor de selecció d'origen d'instruccions (*MUXINSTR*)

```
0. IBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY MUXINSTR IS
4.     PORT ( INSTRSRC : IN STD_LOGIC;
5.            INSTRREG : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
6.            INSTRMEM : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
7.            INSTROUT : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
8. END MUXINSTR;
9.
10.   ARCHITECTURE BEHAVIORAL OF MUXINSTR IS
11.
12.   BEGIN
13.   WITH INSTRSRC SELECT INSTROUT <=
14.       INSTRREG WHEN '0',
15.       INSTRMEM WHEN '1',
16.       (OTHERS => '0') WHEN OTHERS;
17.
18.   END BEHAVIORAL;
```

D.5 Descodificador d'instruccions (*INSTDECOD*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY INSTDECOD IS
4.     PORT ( INSTR : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
5.            D : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
6.            R : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
7.            K : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
8.            S : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
9.            B : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
10.              FORMAT : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
11.              N : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
12.   END INSTDECOD;
13.
14.   ARCHITECTURE BEHAVIORAL OF INSTDECOD IS
15.
16.   SIGNAL FORMATAUX : STD_LOGIC_VECTOR(3 DOWNTO 0);
17.   SIGNAL SIGNAL1  : STD_LOGIC_VECTOR(1 DOWNTO 0);
18.   SIGNAL SIGNAL2  : STD_LOGIC_VECTOR(3 DOWNTO 0);
19.   SIGNAL SIGNAL3  : STD_LOGIC;
20.   SIGNAL SIGNAL4  : STD_LOGIC_VECTOR(9 DOWNTO 0);
21.   SIGNAL SIGNAL5  : STD_LOGIC;
22.   SIGNAL SIGNAL6  : STD_LOGIC_VECTOR(5 DOWNTO 0);
23.   SIGNAL SIGNAL7  : STD_LOGIC;
24.   SIGNAL SIGNAL8  : STD_LOGIC_VECTOR(10 DOWNTO 0);
25.   SIGNAL SIGNAL9  : STD_LOGIC;
26.   SIGNAL SIGNAL10 : STD_LOGIC_VECTOR(9 DOWNTO 0);
27.   SIGNAL SIGNAL11 : STD_LOGIC;
28.   SIGNAL SIGNAL12 : STD_LOGIC_VECTOR(5 DOWNTO 0);
29.   SIGNAL SIGNAL13 : STD_LOGIC;
30.   SIGNAL SIGNAL14 : STD_LOGIC_VECTOR(4 DOWNTO 0);
31.   SIGNAL SIGNAL15 : STD_LOGIC;
32.   SIGNAL SIGNAL16 : STD_LOGIC_VECTOR(2 DOWNTO 0);
33.   SIGNAL SIGNAL17 : STD_LOGIC_VECTOR(1 DOWNTO 0);
34.   SIGNAL SIGNAL18 : STD_LOGIC_VECTOR(3 DOWNTO 0);
35.   SIGNAL SIGNAL19 : STD_LOGIC_VECTOR(1 DOWNTO 0);
36.   SIGNAL SIGNAL20 : STD_LOGIC_VECTOR(7 DOWNTO 0);
37.   SIGNAL SIGNAL21 : STD_LOGIC_VECTOR(1 DOWNTO 0);
38.   SIGNAL SIGNAL22 : STD_LOGIC_VECTOR(7 DOWNTO 0);
39.   SIGNAL SIGNAL23 : STD_LOGIC_VECTOR(1 DOWNTO 0);
40.   SIGNAL SIGNAL24 : STD_LOGIC;
41.   SIGNAL SIGNALFORMAT : STD_LOGIC_VECTOR(17 DOWNTO 0);
42.   BEGIN
43.   ----S'activen senyals segons les característiques del codi de la----
44.   ------------instrucció, associades a un Format específic------------
45.
46.               FORMAT <= FORMATAUX;
47.               WITH INSTR(15 DOWNTO 12) SELECT SIGNAL1 <=
48.                   "00" WHEN "1001",
49.                   "01" WHEN "1111",
50.                   "11" WHEN OTHERS;
51.               SIGNAL2 <= SIGNAL1 & INSTR(11 DOWNTO 10);
52.               WITH SIGNAL2 SELECT SIGNAL3 <=
53.                   '1' WHEN "0010",                --NOMES FORMAT 7
54.                   '0' WHEN OTHERS;
55.               SIGNAL4 <= SIGNAL1 & INSTR(11 DOWNTO 8) & INSTR(3 DOWNTO
   0);
```

```
56.                WITH SIGNAL4 SELECT SIGNAL5 <=
57.                    '1' WHEN "0001001000",          --NOMES FORMAT 8
58.                    '0' WHEN OTHERS;
59.                SIGNAL6 <= SIGNAL1 & INSTR(3 DOWNTO 0);
60.                WITH SIGNAL6 SELECT SIGNAL7 <=
61.                    '1' WHEN "000100",              --NOMES FORMAT 11
62.                    '0' WHEN OTHERS;
63.                SIGNAL8 <= SIGNAL1 & INSTR(11 DOWNTO 9) & INSTR(7 DOWNTO
   5) & INSTR(3 DOWNTO 1);
64.                WITH SIGNAL8 SELECT SIGNAL9 <=
65.                    '1' WHEN "00010000100",             --NOMES FORMAT
   13
66.                    '0' WHEN OTHERS;
67.                SIGNAL10 <= SIGNAL1 & INSTR(11 DOWNTO 8) & INSTR(3 DOWNTO
   0);
68.                WITH SIGNAL10 SELECT SIGNAL11 <=
69.                    '1' WHEN "0001011000",              --NOMES FORMAT 15
70.                    '0' WHEN OTHERS;
71.                SIGNAL12 <= SIGNAL1 & INSTR(3 DOWNTO 0);
72.                WITH SIGNAL12 SELECT SIGNAL13 <=
73.                    '1' WHEN
   "000000"|"000001"|"000010"|"000011"|"000101"|"000110"|"000111"|"001010",
   --NO 0011 (F2  'N' VALID)
74.                    '0' WHEN OTHERS;
75.                SIGNAL14 <= SIGNAL1 & INSTR(11 DOWNTO 9);
76.                WITH SIGNAL14 SELECT SIGNAL15 <=
77.                    '1' WHEN "00010",               --NOMES FORMAT 2
78.                    '0' WHEN OTHERS;
79.                SIGNAL16 <= SIGNAL1 & INSTR(11);
80.                WITH SIGNAL16 SELECT SIGNAL17 <=
81.                    "11" WHEN "010",              --NOMES FORMAT 14
82.                    "10" WHEN "011",              --NOMES FORMAT 5
83.                    "00" WHEN OTHERS;
84.
85.                 SIGNAL18 <= SIGNAL1 & INSTR(15 DOWNTO 14);
86.                 WITH SIGNAL18 SELECT SIGNAL19 <=
87.                    "11" WHEN "1100",              --FORMAT 10 I 1
88.                    "10" WHEN "1101",              --NOMES FORMAT 3
89.                    "00" WHEN OTHERS;
90.
91.                 SIGNAL20 <= SIGNAL1 & SIGNAL19 & INSTR(15 DOWNTO 12);
92.                 WITH SIGNAL20 SELECT SIGNAL21 <=
93.                    "11" WHEN "11110011",            -- NOMES FORMAT 10
94.                    "01" WHEN "11110000" | "11110001" | "11110010", --
   NOMES FORMAT 1
95.                    "00" WHEN OTHERS;
96.
97.                SIGNAL22 <= SIGNAL1 & SIGNAL19 & INSTR(15 DOWNTO 12);
98.                 WITH SIGNAL22 SELECT SIGNAL23 <=
99.                    "11" WHEN "11001000" | "11001010", --NOMES FORMAT 4
100.                   "01" WHEN "11001011",              --NOMES FORMAT 6
101.                   "10" WHEN "11001110",              --NOMES FORMAT 9
102.                   "00" WHEN OTHERS;
103.
104.               WITH INSTR(15 DOWNTO 0) SELECT SIGNAL24 <=
105.                   '1' WHEN "0000000000000000",
106.                   '0' WHEN OTHERS;
107.
108.  -----------Amb les senyals, es dedueix a quin format-----------
109.  ---------------pertany el codi de la instrucció---------------
```

```
110.
111.              SIGNALFORMAT <= SIGNAL1 & SIGNAL3 & SIGNAL5 & SIGNAL7 &
   SIGNAL9 & SIGNAL11 & SIGNAL13 & SIGNAL15 & SIGNAL17 & SIGNAL19  &
   SIGNAL21 & SIGNAL23 & SIGNAL24;
112.
113.              WITH SIGNALFORMAT SELECT FORMATAUX <=
114.                  "0000" WHEN "1100000000011101000",         --FORMAT 1
115.                  "0001" WHEN"0000000110000000000"|"0001000110000000000",
   --FORMAT 2
116.                  "0010" WHEN "1100000000010000000",         --FORMAT 3
117.                  "0011" WHEN "1100000000000000110",         --FORMAT 4
118.                  "0100" WHEN "0100000010000000000",         --FORMAT 5
119.                  "0101" WHEN "1100000000000000010",         --FORMAT 6
120.                  "0110" WHEN"0010000000000000000"|"0010000100000000000"|
121.                   "0010100000000000000"|"0010100100000000000",--FORMAT 7
122.                  "0111"WHEN"0001000010000000000"|"0001010010000000000",-
   -FORMAT 8
123.                  "1000" WHEN "1100000000000000100",         --FORMAT 9
124.                  "1001" WHEN "1100000000011111000",         --FORMAT 10
125.                  "1010" WHEN "0000100000000000000",         --FORMAT 11
126.                  "1011" WHEN "1100000000000000000",         --FORMAT 12
127.                  "1100" WHEN"0000010010000000000"|"0000011010000000000",
   --FORMAT 13
128.                  "1101" WHEN "0100000001100000000",        --FORMAT 14
129.                  "1110" WHEN "0000001000000000000",        --FORMAT 15
130.                  "1111" WHEN "1100000000011101001",        --FORMAT 16
131.                  "0000" WHEN OTHERS;
132.
133.  -------Amb el format, es pot saber on es troben les components------
134.  ------------------D, K, S, N i B de la instrucció-------------------
135.
136.              WITH FORMATAUX SELECT D <=
137.                  INSTR(8 DOWNTO 4) WHEN "0000"|"0001"|"0011"|"0100"|
   "1010"|"0101",
138.                  '1' & INSTR(7 DOWNTO 4) WHEN  "1000"|"0010" ,
139.                  (OTHERS => '0') WHEN OTHERS;
140.
141.              WITH FORMATAUX SELECT R <=
142.                  INSTR(9) & INSTR(3 DOWNTO 0) WHEN "0000",
143.                  "11110" WHEN "0011",
144.                  INSTR(8 DOWNTO 4) WHEN "0100",
145.                  (OTHERS => '0') WHEN OTHERS;
146.
147.
148.              WITH FORMATAUX SELECT K <=
149.                  "0000" & INSTR(11 DOWNTO 8) & INSTR(3 DOWNTO 0) WHEN
   "0010",
150.                  "0000000" & INSTR(11 DOWNTO 10) & INSTR(2 DOWNTO 0)
   WHEN "0011",
151.                  "0000" & INSTR(11 DOWNTO 8) & INSTR(3 DOWNTO 0) WHEN
   "1001"|"1000",
152.                  INSTR(11 DOWNTO 0) WHEN "1011",
153.                  INSTR(9) &INSTR(9) &INSTR(9) &INSTR(9) & INSTR(9) &
154.                  INSTR(9 DOWNTO 3) WHEN "1101",
155.                  (OTHERS => '0') WHEN OTHERS;
156.
157.              WITH FORMATAUX SELECT S <=
158.                  INSTR(6 DOWNTO 4) WHEN "0111",
159.                  INSTR(2 DOWNTO 0) WHEN "1101",
160.                  (OTHERS => '0') WHEN OTHERS;
161.
```

```
162.              WITH FORMATAUX SELECT B <=
163.                  INSTR(2 DOWNTO 0) WHEN "0100",
164.                  (OTHERS => '0') WHEN OTHERS;
165.
166.              WITH FORMATAUX SELECT N <=
167.                  INSTR(13 DOWNTO 10) WHEN "0000",
168.                  INSTR(3 DOWNTO 0) WHEN "0001",
169.                  "00" & INSTR(13 DOWNTO 12) WHEN "0010",
170.                  "00" & INSTR(9) & INSTR(3) WHEN "0011",
171.                  "00" & INSTR(10 DOWNTO 9) WHEN "0100",
172.                  "000" & INSTR(11) WHEN "0101",
173.                  "000" & INSTR(7) WHEN "0111",
174.                  "000" & INSTR(12) WHEN "1011",
175.                  '0' & INSTR(8) & INSTR(4) & INSTR(0) WHEN "1100",
176.                  "000" & INSTR(10) WHEN "1101",
177.                  "00" & INSTR(5 DOWNTO 4) WHEN "1110",
178.                  (OTHERS => '0') WHEN OTHERS;
179.
180.
181.  END BEHAVIORAL;
```

D.6 Banc de Registres (*BANCREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY BANCREG IS
4.     PORT ( CLK : IN STD_LOGIC;
5.            WRITEEN : IN STD_LOGIC;
6.            A1 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
7.            A2 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
8.            A3 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
9.            WD3 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
10.              RD1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
11.              RD2 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
12.  END BANCREG;
13.
14.  ARCHITECTURE BEHAVIORAL OF BANCREG IS
15.  SIGNAL REG0 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
16.  SIGNAL REG1 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
17.  SIGNAL REG2 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
18.  SIGNAL REG3 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
19.  SIGNAL REG4 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
20.  SIGNAL REG5 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
21.  SIGNAL REG6 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
22.  SIGNAL REG7 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
23.  SIGNAL REG8 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
24.  SIGNAL REG9 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
25.  SIGNAL REG10 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
26.  SIGNAL REG11 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
27.  SIGNAL REG12 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
28.  SIGNAL REG13 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
29.  SIGNAL REG14 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
30.  SIGNAL REG15 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
31.  SIGNAL REG16 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
32.  SIGNAL REG17 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
33.  SIGNAL REG18 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
```

```
34.   SIGNAL REG19 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
35.   SIGNAL REG20 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
36.   SIGNAL REG21 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
37.   SIGNAL REG22 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
38.   SIGNAL REG23 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
39.   SIGNAL REG24 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
40.   SIGNAL REG25 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
41.   SIGNAL REG26 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
42.   SIGNAL REG27 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
43.   SIGNAL REG28 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
44.   SIGNAL REG29 : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
45.   SIGNAL REGZLO : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
46.   SIGNAL REGZHI : STD_LOGIC_VECTOR (7 DOWNTO 0) := "00000000";
47.   SIGNAL A1AUX : STD_LOGIC_VECTOR (4 DOWNTO 0);
48.   SIGNAL A2AUX : STD_LOGIC_VECTOR (4 DOWNTO 0);
49.
50.   SIGNAL REG : STD_LOGIC_VECTOR (7 DOWNTO 0)  := "00000000";
51.   BEGIN
52.   PROCESS(CLK)
53.   BEGIN
54.        IF RISING_EDGE(CLK) THEN
55.            CASE A1 IS
56.               WHEN "00000" => RD1 <= REG0;
57.               WHEN "00001" => RD1 <= REG1;
58.               WHEN "00010" => RD1 <= REG2;
59.               WHEN "00011" => RD1 <= REG3;
60.               WHEN "00100" => RD1 <= REG4;
61.               WHEN "00101" => RD1 <= REG5;
62.               WHEN "00110" => RD1 <= REG6;
63.               WHEN "00111" => RD1 <= REG7;
64.               WHEN "01000" => RD1 <= REG8;
65.               WHEN "01001" => RD1 <= REG9;
66.               WHEN "01010" => RD1 <= REG10;
67.               WHEN "01011" => RD1 <= REG11;
68.               WHEN "01100" => RD1 <= REG12;
69.               WHEN "01101" => RD1 <= REG13;
70.               WHEN "01110" => RD1 <= REG14;
71.               WHEN "01111" => RD1 <= REG15;
72.               WHEN "10000" => RD1 <= REG16;
73.               WHEN "10001" => RD1 <= REG17;
74.               WHEN "10010" => RD1 <= REG18;
75.               WHEN "10011" => RD1 <= REG19;
76.               WHEN "10100" => RD1 <= REG20;
77.               WHEN "10101" => RD1 <= REG21;
78.               WHEN "10110" => RD1 <= REG22;
79.               WHEN "10111" => RD1 <= REG23;
80.               WHEN "11000" => RD1 <= REG24;
81.               WHEN "11001" => RD1 <= REG25;
82.               WHEN "11010" => RD1 <= REG26;
83.               WHEN "11011" => RD1 <= REG27;
84.               WHEN "11100" => RD1 <= REG28;
85.               WHEN "11101" => RD1 <= REG29;
86.               WHEN "11110" => RD1 <= REGZLO;
87.               WHEN "11111" => RD1 <= REGZHI;
88.               WHEN OTHERS => RD1 <= (OTHERS => '0');
89.            END CASE;
90.
91.            CASE A2 IS
92.               WHEN "00000" => RD2 <= REG0;
93.               WHEN "00001" => RD2 <= REG1;
94.               WHEN "00010" => RD2 <= REG2;
```

```
95.               WHEN "00011" => RD2 <= REG3;
96.               WHEN "00100" => RD2 <= REG4;
97.               WHEN "00101" => RD2 <= REG5;
98.               WHEN "00110" => RD2 <= REG6;
99.               WHEN "00111" => RD2 <= REG7;
100.              WHEN "01000" => RD2 <= REG8;
101.              WHEN "01001" => RD2 <= REG9;
102.              WHEN "01010" => RD2 <= REG10;
103.              WHEN "01011" => RD2 <= REG11;
104.              WHEN "01100" => RD2 <= REG12;
105.              WHEN "01101" => RD2 <= REG13;
106.              WHEN "01110" => RD2 <= REG14;
107.              WHEN "01111" => RD2 <= REG15;
108.              WHEN "10000" => RD2 <= REG16;
109.              WHEN "10001" => RD2 <= REG17;
110.              WHEN "10010" => RD2 <= REG18;
111.              WHEN "10011" => RD2 <= REG19;
112.              WHEN "10100" => RD2 <= REG20;
113.              WHEN "10101" => RD2 <= REG21;
114.              WHEN "10110" => RD2 <= REG22;
115.              WHEN "10111" => RD2 <= REG23;
116.              WHEN "11000" => RD2 <= REG24;
117.              WHEN "11001" => RD2 <= REG25;
118.              WHEN "11010" => RD2 <= REG26;
119.              WHEN "11011" => RD2 <= REG27;
120.              WHEN "11100" => RD2 <= REG28;
121.              WHEN "11101" => RD2 <= REG29;
122.              WHEN "11110" => RD2 <= REGZLO;
123.              WHEN "11111" => RD2 <= REGZHI;
124.              WHEN OTHERS => RD2 <= (OTHERS => '0');
125.           END CASE;
126.
127.           IF WRITEEN = '1' THEN
128.              CASE A3 IS
129.              WHEN "00000" => REG0 <= WD3;
130.              WHEN "00001" => REG1 <= WD3;
131.              WHEN "00010" => REG2 <= WD3;
132.              WHEN "00011" => REG3 <= WD3;
133.              WHEN "00100" => REG4 <= WD3;
134.              WHEN "00101" => REG5 <= WD3;
135.              WHEN "00110" => REG6 <= WD3;
136.              WHEN "00111" => REG7 <= WD3;
137.              WHEN "01000" => REG8 <= WD3;
138.              WHEN "01001" => REG9 <= WD3;
139.              WHEN "01010" => REG10 <= WD3;
140.              WHEN "01011" => REG11 <= WD3;
141.              WHEN "01100" => REG12 <= WD3;
142.              WHEN "01101" => REG13 <= WD3;
143.              WHEN "01110" => REG14 <= WD3;
144.              WHEN "01111" => REG15 <= WD3;
145.              WHEN "10000" => REG16 <= WD3;
146.              WHEN "10001" => REG17 <= WD3;
147.              WHEN "10010" => REG18 <= WD3;
148.              WHEN "10011" => REG19 <= WD3;
149.              WHEN "10100" => REG20 <= WD3;
150.              WHEN "10101" => REG21 <= WD3;
151.              WHEN "10110" => REG22 <= WD3;
152.              WHEN "10111" => REG23 <= WD3;
153.              WHEN "11000" => REG24 <= WD3;
154.              WHEN "11001" => REG25 <= WD3;
```

```
155.              WHEN "11010" => REG26 <= WD3;
156.              WHEN "11011" => REG27 <= WD3;
157.              WHEN "11100" => REG28 <= WD3;
158.              WHEN "11101" => REG29 <= WD3;
159.              WHEN "11110" => REGZLO <= WD3;
160.              WHEN "11111" => REGZHI <= WD3;
161.              WHEN OTHERS => RD1 <= (OTHERS => '0');
162.              END CASE;
163.           END IF;
164.       END IF;
165.  END PROCESS;
166.  END BEHAVIORAL;
```

D.7 Registre d'entrada de dades (*INREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
3.
4.
5. ENTITY IREG IS
6.     PORT ( CLK : IN STD_LOGIC;
7.            RESET : IN STD_LOGIC;
8.            RESETWD : IN STD_LOGIC;
9.            IREGEN : IN STD_LOGIC;
10.              IREGWR : IN STD_LOGIC;
11.              DATAIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
12.              DATAREG : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
13.  END IREG;
14.
15.  ARCHITECTURE BEHAVIORAL OF IREG IS
16.  SIGNAL DATAAUX : STD_LOGIC_VECTOR (7 DOWNTO 0);
17.  BEGIN
18.  PROCESS(CLK)
19.  BEGIN
20.      IF RESET = '1' OR RESETWD = '1' THEN
21.          DATAAUX <= (OTHERS => ('0'));
22.          DATAREG <= (OTHERS => ('0'));
23.      ELSIF IREGEN = '1' THEN
24.          IF IREGWR = '1' THEN
25.              DATAAUX <= DATAIN;
26.          ELSE
27.              DATAREG <= DATAAUX;
28.          END IF;
29.      END IF;
30.  END PROCESS;
31.  END BEHAVIORAL;
```

D.8 Registre de sortida de dades (OREG)

```
0. LIBRARY IEEE;
```

```
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
3.
4.
5. ENTITY OREG IS
6.     PORT ( CLK : IN STD_LOGIC;
7.            RESET : IN STD_LOGIC;
8.            RESETWD : IN STD_LOGIC;
9.            OREGEN : IN STD_LOGIC;
10.              REGDATA : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
11.              DATAOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
12.   END OREG;
13.
14.   ARCHITECTURE BEHAVIORAL OF OREG IS
15.   --SIGNAL DATAAUX : STD_LOGIC_VECTOR (7 DOWNTO 0);
16.   BEGIN
17.   PROCESS(CLK)
18.   BEGIN
19.       IF RESET = '1' OR RESETWD = '1' THEN
20.           DATAOUT <= "00000000";
21.       ELSIF OREGEN = '1' THEN
22.           DATAOUT <= REGDATA;
23.           --DATAAUX <= (OTHERS => ('0'));
24.           --DATAOUT <= (OTHERS => ('0'));
25.       END IF;
26.   END PROCESS;
27.   END BEHAVIORAL;
```

D.9 Multiplexor de l'operand A (*MUXALUSRCA*)

```
0. ARCHITECTURE BEHAVIORAL OF MUXALUSRCA IS
1. SIGNAL RD1AUX : STD_LOGIC_VECTOR(8 DOWNTO 0);
2. BEGIN
3. RD1AUX <= PC - '1';
4.         WITH ALUSRCA SELECT ALUA <=
5.             "000" & PC WHEN "00",
6.             "000" & RD1AUX WHEN "01",
7.             ("0000" & RD1) WHEN "11",
8.             (OTHERS => '0') WHEN OTHERS;
9. END BEHAVIORAL;
```

D.10 Multiplexor de l'operand B (*MUXALUSRCB*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
```

```
2.
3. ENTITY MUXALUSRCB IS
4.     PORT ( K : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
5.            RD2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
6.            B : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7.            ALUSRCB : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8.            ALUB : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
9. END MUXALUSRCB;
10.
11.   ARCHITECTURE BEHAVIORAL OF MUXALUSRCB IS
12.   BEGIN
13.
14.           WITH ALUSRCB SELECT ALUB <=
15.               K WHEN "000",
16.               "0000" & RD2 WHEN "001",
17.               "000000000001" WHEN "010",
18.               "000000000010" WHEN "011",
19.               "0000" & B WHEN "100",
20.               "000000000000" WHEN OTHERS;
21.   END BEHAVIORAL;
```

D.11 ALU (*ALU*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
4.
5. ENTITY ALU IS
6.     PORT (
7.            FLAGSIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
8.            ALUA : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
9.            ALUB : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
10.              ALUCONTROL : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
11.              N : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
12.              FLAGS : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
13.              ALUOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
14.
15.   END ALU;
16.
17.   ARCHITECTURE BEHAVIORAL OF ALU IS
18.   SIGNAL ALUOP : STD_LOGIC_VECTOR (4 DOWNTO 0)     ;
19.   SIGNAL RESULT : STD_LOGIC_VECTOR (11 DOWNTO 0)       ;
20.   SIGNAL SREGAUX : STD_LOGIC_VECTOR (7 DOWNTO 0):= "00000000";
21.   SIGNAL FORMATAUX : STD_LOGIC_VECTOR (6 DOWNTO 0)     ;
22.   SIGNAL SIGNAL1 : STD_LOGIC;
23.   SIGNAL SIGNAL2 : STD_LOGIC;
24.   SIGNAL ALUAAUX : STD_LOGIC_VECTOR (7 DOWNTO 0):= "00000000";
25.   SIGNAL ALUBAUX : STD_LOGIC_VECTOR (7 DOWNTO 0):= "00000000";
26.   SIGNAL ALUOPAUX : STD_LOGIC_VECTOR (5 DOWNTO 0) := "000000";
27.
28.   BEGIN
29.           FORMATAUX <= ALUCONTROL & N;
30.             WITH FORMATAUX SELECT ALUOP <=
31.                   ----FORMAT 1
32.                   "10001" WHEN "0000001",
33.                   "10001" WHEN "0000010",
34.                   "00000" WHEN "0000011",
35.                   "00001" WHEN "0000100",
36.                   "00001" WHEN "0000101",
```

```
37.                     "00001" WHEN "0000110",
38.                     "10000" WHEN "0000111",
39.                     "00010" WHEN "0001000",
40.                     "00100" WHEN "0001001",
41.                     "00011" WHEN "0001010",
42.                     --FORMAT 2
43.                     "00101" WHEN "0010000",
44.                     "00110" WHEN "0010001",
45.                     "01110" WHEN "0010010",
46.                     "00111" WHEN "0010011",
47.                     "01011" WHEN "0010101",
48.                     "01001" WHEN "0010110",
49.                     "01010" WHEN "0010111",
50.                     "01000" WHEN "0011010",
51.                     --FORMAT 3
52.                     "10001" WHEN "0100000",
53.                     "00001" WHEN "0100001",
54.                     "00011" WHEN "0100010",
55.                     "00010" WHEN "0100011",
56.                     --PASS A
57.                     "01111" WHEN "0110000",
58.                     "01111" WHEN "0110001",
59.                     "01111" WHEN "0110010",
60.                     "01111" WHEN "0110011",
61.                     "01111" WHEN "0110100",
62.                     "01111" WHEN "0110101",
63.                     "01111" WHEN "0110110",
64.                     "01111" WHEN "0110111",
65.                     "01111" WHEN "0111000",
66.                     "01111" WHEN "0111001",
67.                     "01111" WHEN "0111010",
68.                     "01111" WHEN "0111011",
69.                     "01111" WHEN "0111100",
70.                     "01111" WHEN "0111101",
71.                     "01111" WHEN "0111110",
72.                     "01111" WHEN "0111111",
73.                     --AND
74.                     "00010" WHEN "1000000",
75.                     "00010" WHEN "1000001",
76.                     "00010" WHEN "1000010",
77.                     "00010" WHEN "1000011",
78.                     "00010" WHEN "1000100",
79.                     "00010" WHEN "1000101",
80.                     "00010" WHEN "1000110",
81.                     "00010" WHEN "1000111",
82.                     "00010" WHEN "1001000",
83.                     "00010" WHEN "1001001",
84.                     "00010" WHEN "1001010",
85.                     "00010" WHEN "1001011",
86.                     "00010" WHEN "1001100",
87.                     "00010" WHEN "1001101",
88.                     "00010" WHEN "1001110",
89.                     "00010" WHEN "1001111",
90.                     --OR
91.                     "00011" WHEN "1010000",
92.                     "00011" WHEN "1010001",
93.                     "00011" WHEN "1010010",
94.                     "00011" WHEN "1010011",
95.                     "00011" WHEN "1010100",
96.                     "00011" WHEN "1010101",
```

```
97.                   "00011" WHEN "1010110",
98.                   "00011" WHEN "1010111",
99.                   "00011" WHEN "1011000",
100.                  "00011" WHEN "1011001",
101.                  "00011" WHEN "1011010",
102.                  "00011" WHEN "1011011",
103.                  "00011" WHEN "1011100",
104.                  "00011" WHEN "1011101",
105.                  "00011" WHEN "1011110",
106.                  "00011" WHEN "1011111",
107.
108.                  --ADD
109.                  "00000" WHEN "1100000",
110.                  "00000" WHEN "1100001",
111.                  "00000" WHEN "1100010",
112.                  "00000" WHEN "1100011",
113.                  "00000" WHEN "1100100",
114.                  "00000" WHEN "1100101",
115.                  "00000" WHEN "1100110",
116.                  "00000" WHEN "1100111",
117.                  "00000" WHEN "1101000",
118.                  "00000" WHEN "1101001",
119.                  "00000" WHEN "1101010",
120.                  "00000" WHEN "1101011",
121.                  "00000" WHEN "1101100",
122.                  "00000" WHEN "1101101",
123.                  "00000" WHEN "1101110",
124.                  "00000" WHEN "1101111",
125.                  (OTHERS => '0') WHEN OTHERS;
126.
127.              ALUAAUX <= ALUA(7 DOWNTO 0);
128.              ALUBAUX <= ALUB(7 DOWNTO 0);
129.              ALUOPAUX <= ALUOP & ALUCONTROL(2);
130.
131.              WITH ALUOPAUX SELECT RESULT <=
132.                  (ALUA + ALUB) WHEN "000001",            --ADD 11b
133.                  (ALUA OR ALUB) WHEN "000111",           --OR 11b
134.                  (ALUA AND ALUB) WHEN "000101",          --AND 11b
135.                  ALUA WHEN "011111",                     --PASS 11b
136.                  (ALUA + ALUB) WHEN "000000",            --ADD
137.                  (ALUA - ALUB) WHEN "000010",            --SUB
138.                  "0000" & (ALUAAUX AND ALUBAUX) WHEN "000100",  --AND
139.                  "0000" & (ALUAAUX OR ALUBAUX) WHEN "000110",   --OR
140.                  "0000" & (ALUAAUX XOR ALUBAUX) WHEN "001000",  --EOR
141.                  "0000" & (NOT ALUAAUX) WHEN "001010",          --NEG
142.                  "0000" & ((NOT ALUAAUX) + '1') WHEN "001100", --COMP
143.                  "0000" & (ALUAAUX + '1') WHEN "001110",        --INC
144.                  "0000" & (ALUAAUX - '1') WHEN "010000",        --DEC
145.                  ("0000" & '0' & ALUAAUX(7 DOWNTO 1)) WHEN "010010",
    --LSR
146.                  ("0000" & FLAGSIN(0) & ALUAAUX(7 DOWNTO 1)) WHEN
    "010100",   --ROR
147.                  ("0000" & ALUAAUX(7) & ALUAAUX(7 DOWNTO 1)) WHEN
    "010110",      --ASR
148.                  "111111111111" WHEN "011000",                --SET
149.                  "000000000000" WHEN "011010",                --CLR
 150.                   "0000" & (ALUAAUX(3 DOWNTO 0) & ALUAAUX(7 DOWNTO 4))
                                                    WHEN "011100", --SWAP
151.                  "0000" & ALUAAUX WHEN "011110",           --PASS A
152.                  (ALUA + ALUBAUX + FLAGSIN(0)) WHEN "100000",  --ADC
153.                  (ALUA - ALUB - FLAGSIN(0)) WHEN "100010",     --SBC
```

```
154.                    (OTHERS => '0') WHEN OTHERS;
155.
156.
157.           WITH (ALUOP & ALUAAUX(3) & ALUBAUX(3)& RESULT(3)) SELECT
   SREGAUX(5) <=  --H
158.                    '1' WHEN "00000110"|"00000111"|"00001001",
   -- '1' + '1' = '(1)0'
159.
160.                    '0' WHEN OTHERS;
161.  SREGAUX(4) <= SREGAUX(2) XOR SREGAUX(3);                          --S
162.
163.             WITH (ALUOP & ALUAAUX(7) & ALUBAUX(7) & RESULT(7)) SELECT
   SREGAUX(3) <=   --V
164.                    '1' WHEN "00000001",              -- '+' + '+' = '-'
165.                    '1' WHEN "00000110",              -- '-' + '-' = '+'
166.                    '0' WHEN OTHERS;
167.
168.             WITH RESULT(7) SELECT SREGAUX(2) <=                    --N
169.                    '1' WHEN '1',
170.                    '0' WHEN OTHERS;
171.
172.             WITH RESULT SELECT SREGAUX(1) <=                       --Z
173.                    '1' WHEN "000000000000",
174.                    '0' WHEN OTHERS;
175.
176.           WITH RESULT(8) SELECT SIGNAL1 <=                         --C
177.                    '1' WHEN '1',
178.                    '0' WHEN OTHERS;
179.             WITH ALUOP SELECT SIGNAL2 <=
180.                   ALUAAUX(0) WHEN "01010",
181.                    '0' WHEN OTHERS;
182.          SREGAUX(0) <= SIGNAL1 OR SIGNAL2;
183.
184.             FLAGS <= SREGAUX;
185.             ALUOUT <= RESULT;
186.  END BEHAVIORAL;
```

D.12 Registre a la sortida de ALU (*ALUREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY ALUREG IS
4.     PORT (  ALUOUT : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
5.             CLK : IN STD_LOGIC;
6.             RESET : IN STD_LOGIC;
7.             RESETWD: IN STD_LOGIC;
8.             ALUREGOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
9.             ALUREG2: OUT STD_LOGIC_VECTOR(11 DOWNTO 0));
10.   END ALUREG;
11.
12.   ARCHITECTURE BEHAVIORAL OF ALUREG IS
13.   SIGNAL ALUREGAUX: STD_LOGIC_VECTOR (11 DOWNTO 0);
14.   BEGIN
15.   PROCESS (CLK)
16.      BEGIN
17.        IF RISING_EDGE(CLK) THEN
18.           IF RESET = '0' AND RESETWD = '0'THEN
19.
20.               ALUREGAUX <= ALUOUT;
21.           ELSE
22.               ALUREGAUX <= (OTHERS=>'0');
23.
24.           END IF;
25.        END IF;
26.   END PROCESS;
27.   ALUREGOUT <= ALUREGAUX;
28.   ALUREG2 <= ALUREGAUX;
29.   END BEHAVIORAL;
```

D.13 Multiplexor d'origen de PC, a la ALU (*MUXPC1*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY MUXPC1 IS
5.     PORT ( PCREG : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
6.             PCALU : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
7.             PCSRC : IN STD_LOGIC;
8.             PCMUXOUT : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
9. END MUXPC1;
10.
11.   ARCHITECTURE BEHAVIORAL OF MUXPC1 IS
12.
13.   BEGIN
14.       WITH PCSRC SELECT PCMUXOUT <=
15.               PCREG(8 DOWNTO 0) WHEN '0',
16.               PCALU(8 DOWNTO 0) WHEN '1',
17.               (OTHERS => '0') WHEN OTHERS;
18.
19.   END BEHAVIORAL;
```

D.14 Multiplexor d'origen de PC, a la *Stack* (*MUXPC2*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY MUXPC2 IS
5.     PORT ( PCSTACK : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
6.            PC1M : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
7.            PC2SRC : IN STD_LOGIC;
8.            PC2MUXOUT : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
9. END MUXPC2;
10.
11.   ARCHITECTURE BEHAVIORAL OF MUXPC2 IS
12.
13.   BEGIN
14.       WITH PC2SRC SELECT PC2MUXOUT <=
15.               PCSTACK(8 DOWNTO 0) WHEN '1',
16.               PC1M WHEN '0',
17.               (OTHERS => '0') WHEN OTHERS;
18.
19.   END BEHAVIORAL;
```

D.15 Memòria RAM (*RAMMEM*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3.
4. ENTITY RAMMEM IS
5. PORT ( CLK : IN STD_LOGIC; --RELLOTGE
6.        RAMEN : IN STD_LOGIC; --HABILITACI3GLOBAL
7.        MEMWRITE : IN STD_LOGIC; -- HABILITACI3D'ESCRIPTURA
8.        A: IN STD_LOGIC_VECTOR (5 DOWNTO 0); -- ADRE‖ LECTURA/ESCRIPTURA
9.        WD : IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- ENTRADA DADA
10.          RD_RAM : OUT STD_LOGIC_VECTOR (7 DOWNTO 0) ; --SORTIDA DADA
11.          RAM32: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
12.
13.   END RAMMEM;
14.
15.   ARCHITECTURE BEHAVIORAL OF RAMMEM IS
16.   TYPE RAM64X8 IS ARRAY (0 TO 63) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
17.       SIGNAL RAM : RAM64X8 := (OTHERS => (OTHERS => '0'));
18.       SIGNAL READ_A : UNSIGNED(5 DOWNTO 0) := (OTHERS => ('0'));
19.       SIGNAL AAUX : UNSIGNED(5 DOWNTO 0);
20.       SIGNAL AUXRAM: STD_LOGIC:='0';
21.       SIGNAL RDRAMAUX: STD_LOGIC_VECTOR( 7 DOWNTO 0) := (OTHERS =>
   ('0'));
22.   BEGIN
23.
24.   PROCESS (CLK)
25.   BEGIN
26.   AAUX <= UNSIGNED(A);
27.   IF (CLK'EVENT AND CLK = '1') THEN
28.       IF (RAMEN = '1') THEN
29.           IF (MEMWRITE = '1') THEN
30.               RAM(TO_INTEGER(AAUX)) <= WD;
31.               AUXRAM <= '1';
```

```
32.           ELSE
33.               RDRAMAUX <= RAM(TO_INTEGER(AAUX));
34.           END IF;
35.       ELSE AUXRAM<='0';
36.       END IF;
37.   END IF;
38.   END PROCESS;
39.   RD_RAM <= RDRAMAUX;
40.   RAM32 <= RAM(32);
41.   END BEHAVIORAL;
```

D.16 Multiplexor d'entrada de dades al banc de registres (*MUXREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY MUXREG IS
5.     PORT ( RD2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
6.            K : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
7.            ALUOUT : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
8.            RAMOUT : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
9.            REGSRCIN : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
10.              IREGOUT : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
11.              MUXREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
12.   END MUXREG;
13.
14.   ARCHITECTURE BEHAVIORAL OF MUXREG IS
15.   BEGIN
16.       WITH REGSRCIN SELECT MUXREGOUT <=
17.                RD2 WHEN "000",
18.                K(7 DOWNTO 0) WHEN "001",
19.                ALUOUT(7 DOWNTO 0) WHEN "010",
20.                RAMOUT WHEN "011",
21.                IREGOUT WHEN "100",
22.                (OTHERS => '0') WHEN OTHERS;
23.   END BEHAVIORAL;
```

D.17 Memòria *Stack* (*STACKMEM*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY STACKMEM IS
4.     PORT (  CLK : IN STD_LOGIC;
5.             STACKWR : IN STD_LOGIC;
6.             STACKEN : IN STD_LOGIC;
7.             STACKIN : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
8.             STACKOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
9. END STACKMEM;
10.
11.   ARCHITECTURE BEHAVIORAL OF STACKMEM IS
12.   TYPE STACK3X8 IS ARRAY (0 TO 2) OF STD_LOGIC_VECTOR (8 DOWNTO 0);
13.       SIGNAL STACKAUX : STACK3X8 :=
   ("000000000","000000000","000000000");
14.       SIGNAL STACKOUTAUX : STD_LOGIC_VECTOR(11 DOWNTO 0) :=
   "000000000000";
```

```
15.         SIGNAL SIGNAL1 : STD_LOGIC_VECTOR(1 DOWNTO 0);
16.     BEGIN
17.
18.     PROCESS (CLK)
19.     BEGIN
20.         IF (CLK'EVENT AND CLK = '1') THEN
21.             IF STACKEN = '1' THEN
22.                 IF STACKWR = '1' THEN
23.                     STACKAUX(2) <= STACKAUX(1);
24.                     STACKAUX(1) <= STACKAUX(0);
25.                     STACKAUX(0) <= STACKIN(8 DOWNTO 0);
26.                 ELSE
27.                     STACKOUTAUX <= "000" & STACKAUX(0);
28.                     STACKAUX(0) <= STACKAUX(1);
29.                     STACKAUX(1) <= STACKAUX(2);
30.
31.                 END IF;
32.             END IF;
33.         END IF;
34.     END PROCESS;
35.     STACKOUT <= STACKOUTAUX;
36.     END BEHAVIORAL;
```

D.18 Multiplexor de selecció de l'entrada *S* (*MUXSREGA*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY MUXSREGA IS
5.     PORT ( S : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
6.            SREGSRCA : IN STD_LOGIC;
7.            SSSMUXOUT : OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
8. END MUXSREGA;
9.              BITSET WHEN '1',
10.                '0' WHEN OTHERS;
11.     END BEHAVIORAL;
```

D.19 Multiplexor de selecció de l'entrada *BitSet* (*MUXSREGB*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY MUXSREGB IS
5.     PORT ( BITSET : IN STD_LOGIC;
6.            RR : IN STD_LOGIC;
7.            SREGSRCB : IN STD_LOGIC;
8.            BITSETMUXOUT : OUT STD_LOGIC);
9. END MUXSREGB;
10.
11.     ARCHITECTURE BEHAVIORAL OF MUXSREGB IS
12.
13.     BEGIN
14.         WITH SREGSRCB SELECT BITSETMUXOUT <=
15.                 RR WHEN '0',
```

D.20 Multiplexor de selecció de l'entrada de SREG (*MUXSREGC*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY MUXSREGC IS
4.     PORT ( SREGDECOD : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
5.            SREGALU : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
6.            SREGSRCC : IN STD_LOGIC;
7.            SREGMUXOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
8. END MUXSREGC;
9.
10.   ARCHITECTURE BEHAVIORAL OF MUXSREGC IS
11.   BEGIN
12.       WITH SREGSRCC SELECT SREGMUXOUT <=
13.               SREGALU WHEN '0',
14.               SREGDECOD WHEN '1',
15.               (OTHERS => '0') WHEN OTHERS;
16.   END BEHAVIORAL;
```

D.21 Descodificador R (*RRDECOD*)

```
17.   LIBRARY IEEE;
18.   USE IEEE.STD_LOGIC_1164.ALL;
19.
20.   ENTITY RRDECOD IS
21.       PORT ( RR : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
22.              B : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
23.              RROUT : OUT STD_LOGIC);
24.   END RRDECOD;
25.
26.   ARCHITECTURE BEHAVIORAL OF RRDECOD IS
27.   BEGIN
28.       WITH B SELECT RROUT <=
29.           RR(0) WHEN "000",
30.           RR(1) WHEN "001",
31.           RR(2) WHEN "010",
32.           RR(3) WHEN "011",
33.           RR(4) WHEN "100",
34.           RR(5) WHEN "101",
35.           RR(6) WHEN "110",
36.           RR(7) WHEN "111",
37.           '0' WHEN OTHERS;
38.   END BEHAVIORAL;
```

D.22 Codificador B (*BDECOD*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY BDECOD IS
5.     PORT ( B : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
6.            T : IN STD_LOGIC;
7.            BOUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
8. END BDECOD;
9. ARCHITECTURE BEHAVIORAL OF BDECOD IS
10.   SIGNAL BDECAUX : STD_LOGIC_VECTOR(3 DOWNTO 0);
11.   BEGIN
12.       BDECAUX <= T & B;
13.       WITH BDECAUX SELECT BOUT <=
14.           "00000001" WHEN "1000",
15.           "00000010" WHEN "1001",
16.           "00000100" WHEN "1010",
17.           "00001000" WHEN "1011",
18.           "00010000" WHEN "1100",
19.           "00100000" WHEN "1101",
20.           "01000000" WHEN "1110",
21.           "10000000" WHEN "1111",
22.           "11111110" WHEN "0000",
23.           "11111101" WHEN "0001",
24.           "11111011" WHEN "0010",
25.           "11110111" WHEN "0011",
26.           "11101111" WHEN "0100",
27.           "11011111" WHEN "0101",
28.           "10111111" WHEN "0110",
29.           "01111111" WHEN "0111",
30.           "01010101" WHEN OTHERS;
31.   END BEHAVIORAL;
```

D.23 Codificador de SREG (*SREGDECOD*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.NUMERIC_STD.ALL;
3.
4. ENTITY SREGDECOD IS
5.     PORT ( SREGIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
6.            SIN : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
7.            BITSETIN : IN STD_LOGIC;
8.            SREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9. END SREGDECOD;
10.
11.   ARCHITECTURE BEHAVIORAL OF SREGDECOD IS
12.   SIGNAL SREGAUX : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
13.   SIGNAL SREGDECAUX : STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');
14.
15.   BEGIN
16.       SREGDECAUX <= BITSETIN & SIN;
17.       WITH SREGDECAUX SELECT SREGAUX <=
18.           "00000001" WHEN "1000",
19.           "00000010" WHEN "1001",
```

```
20.           "00000100" WHEN "1010",
21.           "00001000" WHEN "1011",
22.           "00010000" WHEN "1100",
23.           "00100000" WHEN "1101",
24.           "01000000" WHEN "1110",
25.           "10000000" WHEN "1111",
26.           "11111110" WHEN "0000",
27.           "11111101" WHEN "0001",
28.           "11111011" WHEN "0010",
29.           "11110111" WHEN "0011",
30.           "11101111" WHEN "0100",
31.           "11011111" WHEN "0101",
32.           "10111111" WHEN "0110",
33.           "01111111" WHEN "0111",
34.           "01010101" WHEN OTHERS;
35.
36.       WITH SREGDECAUX(3) SELECT SREGOUT <=
37.           SREGIN OR SREGAUX WHEN '1',
38.           SREGIN AND SREGAUX WHEN '0',
39.           (OTHERS => '0') WHEN OTHERS;
40.   END BEHAVIORAL;
```

D.24 Registre d'estat (*SREG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY SREG IS
4.     PORT ( RESET : IN STD_LOGIC;
5.            RESETWD: IN STD_LOGIC;
6.            SREGIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
7.            SREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
8.            CLK : IN STD_LOGIC;
9.            SREGLOAD : IN STD_LOGIC;
10.              I : OUT STD_LOGIC;
11.              T : OUT STD_LOGIC;
12.              H : OUT STD_LOGIC;
13.              S : OUT STD_LOGIC;
14.              V : OUT STD_LOGIC;
15.              N : OUT STD_LOGIC;
16.              Z : OUT STD_LOGIC;
17.              C : OUT STD_LOGIC);
18.   END SREG;
19.
20.   ARCHITECTURE BEHAVIORAL OF SREG IS
21.   SIGNAL SREGAUX : STD_LOGIC_VECTOR(7 DOWNTO 0);
22.   BEGIN
23.   PROCESS (CLK)
24.      BEGIN
25.        IF RISING_EDGE(CLK) THEN
26.           IF RESET='1' OR RESETWD = '1' THEN
27.               SREGAUX <= (OTHERS=>'0');
28.           ELSIF SREGLOAD='1' THEN
29.               SREGAUX <= SREGIN;
30.           END IF;
31.        END IF;
32.   END PROCESS;
33.   I <= SREGAUX(7);
34.   T <= SREGAUX(6);
35.   H <= SREGAUX(5);
```

```
36.    S <= SREGAUX(4);
37.    V <= SREGAUX(3);
38.    N <= SREGAUX(2);
39.    Z <= SREGAUX(1);
40.    C <= SREGAUX(0);
41.    SREGOUT <= SREGAUX;
42.    END BEHAVIORAL;
```

D.25 Multiplexor de selecció de *flag* (*FLAGMUX*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3. ENTITY MUXFLAGS IS
4.     PORT ( I : IN STD_LOGIC;
5.            T : IN STD_LOGIC;
6.            H : IN STD_LOGIC;
7.            S : IN STD_LOGIC;
8.            V : IN STD_LOGIC;
9.            N : IN STD_LOGIC;
10.              Z : IN STD_LOGIC;
11.              C : IN STD_LOGIC;
12.              MUXFLAGOUT : OUT STD_LOGIC;
13.              FLAGSELECT : IN STD_LOGIC_VECTOR (2 DOWNTO 0));
14.   END MUXFLAGS;
15.
16.   ARCHITECTURE BEHAVIORAL OF MUXFLAGS IS
17.
18.   BEGIN
19.       WITH FLAGSELECT SELECT MUXFLAGOUT <=
20.           C WHEN "000",
21.           Z WHEN "001",
22.           N WHEN "010",
23.           V WHEN "011",
24.           S WHEN "100",
25.           H WHEN "101",
26.           T WHEN "110",
27.           I WHEN "111",
28.           '0' WHEN OTHERS;
29.   END BEHAVIORAL;
30.   LIBRARY IEEE;
31.   USE IEEE.STD_LOGIC_1164.ALL;
32.   USE IEEE.NUMERIC_STD.ALL;
```

D.26 Datapath (DATAPATHTINY)

```
0. ENTITY DATAPATH IS
1.     PORT(
2.         ---Senyals de TOP
3.         DATAIN : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
4.         DATAOUT: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
5.         CLK, RESET, RESETWD: IN STD_LOGIC;
6.         --Senyals de la Unitat de Control
7.         ALUCONTROL, ALUSRCB, FLAGSELECT: IN STD_LOGIC_VECTOR(2 DOWNTO
   0);
```

```
8.          REGSRC: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
9.          ALUSRCA: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
10.            IRWRITE, PCWRITE, PCSRC, PC2SRC, REGWRITE, SREGSRCA, SREGSRCB
   :IN STD_LOGIC;
11.            SREGSRCC, SREGLOAD, BITSET, MEMWRITE, STACKEN, STACKWR,
   RAMEN: IN STD_LOGIC;
12.            INSTRSRC, IREGEN, IREGWR, OREGEN: IN STD_LOGIC;
13.            FORMAT, N: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
14.            S : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
15.            STATUSREG: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
16.    END DATAPATH;
17.
18.    ARCHITECTURE STRUCT OF DATAPATH IS
19.
20.        COMPONENT ALU
21.            PORT (
22.               FLAGSIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
23.               ALUA : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
24.               ALUB : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
25.               ALUCONTROL : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
26.               N : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
27.               FLAGS : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
28.               ALUOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
29.        END COMPONENT;
30.
31.        COMPONENT ALUREG
32.            PORT (  ALUOUT : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
33.                CLK : IN STD_LOGIC;
34.                RESET : IN STD_LOGIC;
35.                RESETWD: IN STD_LOGIC;
36.                ALUREGOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
37.                ALUREG2 : OUT STD_LOGIC_VECTOR(11 DOWNTO 0));
38.        END COMPONENT;
39.
40.        COMPONENT BDECOD
41.            PORT ( B : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
42.               T : IN STD_LOGIC;
43.               BOUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
44.        END COMPONENT;
45.
46.        COMPONENT BANCREG
47.            PORT ( CLK : IN STD_LOGIC;
48.               WRITEEN : IN STD_LOGIC;
49.               A1 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
50.               A2 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
51.               A3 : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
52.               WD3 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
53.               RD1 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
54.               RD2 : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
55.         END COMPONENT;
56.
57.         COMPONENT FLASHMEM
58.            PORT(
59.                A : IN STD_LOGIC_VECTOR(8 DOWNTO 0);
60.                SPO : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
61.         END COMPONENT;
62.
63.         COMPONENT INSTDECOD
64.            PORT ( INSTR : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
65.               D : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
66.               R : OUT STD_LOGIC_VECTOR (4 DOWNTO 0);
```

```
67.               K : OUT STD_LOGIC_VECTOR (11 DOWNTO 0);
68.               S : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
69.               B : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
70.               N : OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
71.               FORMAT : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
72.               );
73.         END COMPONENT;
74.
75.         COMPONENT INSTREG
76.            PORT (  CLK : IN STD_LOGIC;
77.                 RESET : IN STD_LOGIC;
78.                 RESETWD: IN STD_LOGIC;
79.                 IRWRITE : IN STD_LOGIC;
80.                 SPO : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
81.                 INSTR : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
82.         END COMPONENT;
83.
84.         COMPONENT IREG
85.            PORT ( CLK : IN STD_LOGIC;
86.               RESET : IN STD_LOGIC;
87.               RESETWD : IN STD_LOGIC;
88.               IREGEN : IN STD_LOGIC;
89.               IREGWR : IN STD_LOGIC;
90.               DATAIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
91.               DATAREG : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
92.         END COMPONENT;
93.
94.         COMPONENT MUXALUSRCA
95.            PORT ( PC : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
96.               RD1 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
97.               ALUSRCA : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
98.               ALUA : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
99.         END COMPONENT;
100.
101.        COMPONENT MUXALUSRCB
102.           PORT ( K : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
103.              RD2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
104.              B : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
105.              ALUSRCB : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
106.              ALUB : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
107.        END COMPONENT;
108.
109.        COMPONENT MUXFLAGS
110.           PORT ( I : IN STD_LOGIC;
111.              T : IN STD_LOGIC;
112.              H : IN STD_LOGIC;
113.              S : IN STD_LOGIC;
114.              V : IN STD_LOGIC;
115.              N : IN STD_LOGIC;
116.              Z : IN STD_LOGIC;
117.              C : IN STD_LOGIC;
118.              MUXFLAGOUT : OUT STD_LOGIC;
119.              FLAGSELECT : IN STD_LOGIC_VECTOR (2 DOWNTO 0));
120.        END COMPONENT;
121.
122.        COMPONENT MUXINSTR
123.           PORT ( INSTRSRC : IN STD_LOGIC;
124.              INSTRREG : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
125.              INSTRMEM : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
126.              INSTROUT : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));
```

```
127.        END COMPONENT;
128.
129.        COMPONENT MUXPC1
130.           PORT ( PCREG : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
131.              PCALU : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
132.              PCSRC : IN STD_LOGIC;
133.              PCMUXOUT : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
134.        END COMPONENT;
135.
136.        COMPONENT MUXPC2 IS
137.           PORT ( PCSTACK : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
138.              PC1M : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
139.              PC2SRC : IN STD_LOGIC;
140.              PC2MUXOUT : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
141.        END COMPONENT;
142.
143.        COMPONENT MUXREG
144.           PORT ( RD2 : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
145.              K : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
146.              ALUOUT : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
147.              RAMOUT : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
148.              REGSRCIN : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
149.              IREGOUT : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
150.              MUXREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
151.        END COMPONENT;
152.
153.        COMPONENT MUXSREGA
154.           PORT ( S : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
155.              SREGSRCA : IN STD_LOGIC;
156.              SSSMUXOUT : OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
157.        END COMPONENT;
158.
159.        COMPONENT MUXSREGB
160.           PORT ( BITSET : IN STD_LOGIC;
161.              RR : IN STD_LOGIC;
162.              SREGSRCB : IN STD_LOGIC;
163.              BITSETMUXOUT : OUT STD_LOGIC);
164.        END COMPONENT;
165.
166.        COMPONENT MUXSREGC
167.           PORT ( SREGDECOD : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
168.              SREGALU : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
169.              SREGSRCC : IN STD_LOGIC;
170.              SREGMUXOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
171.        END COMPONENT;
172.
173.        COMPONENT OREG
174.           PORT ( CLK : IN STD_LOGIC;
175.              RESET : IN STD_LOGIC;
176.              RESETWD : IN STD_LOGIC;
177.              OREGEN : IN STD_LOGIC;
178.              REGDATA : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
179.              DATAOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
180.       END COMPONENT;
181.
182.       COMPONENT PCREG
183.           PORT (  CLK : IN STD_LOGIC;
184.               RESET : IN STD_LOGIC;
185.               RESETWD: IN STD_LOGIC;
186.               PCWRITE : IN STD_LOGIC;
187.               PC : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
```

```
188.                PCN : OUT STD_LOGIC_VECTOR (8 DOWNTO 0));
189.       END COMPONENT;
190.
191.       COMPONENT RAMMEM
192.           PORT ( CLK : IN STD_LOGIC; --RELLOTGE
193.                  RAMEN : IN STD_LOGIC; --HABILITACI3GLOBAL
194.                  MEMWRITE : IN STD_LOGIC; -- HABILITACI3D'ESCRIPTURA
195.                  A: IN STD_LOGIC_VECTOR (5 DOWNTO 0); -- ADRE‖
   LECTURA/ESCRIPTURA
196.                  WD : IN STD_LOGIC_VECTOR (7 DOWNTO 0); -- ENTRADA DADA
197.                  RD_RAM : OUT STD_LOGIC_VECTOR (7 DOWNTO 0); --SORTIDA DADA
198.                  RAM32: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
199.       END COMPONENT;
200.
201.       COMPONENT RRDECOD
202.          PORT ( RR : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
203.                 B : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
204.                 RROUT : OUT STD_LOGIC);
205.       END COMPONENT;
206.
207.       COMPONENT SREG
208.           PORT ( RESET : IN STD_LOGIC;
209.                  RESETWD: IN STD_LOGIC;
210.                  SREGIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
211.                  SREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
212.                  CLK : IN STD_LOGIC;
213.                  SREGLOAD : IN STD_LOGIC;
214.                  I : OUT STD_LOGIC;
215.                  T : OUT STD_LOGIC;
216.                  H : OUT STD_LOGIC;
217.                  S : OUT STD_LOGIC;
218.                  V : OUT STD_LOGIC;
219.                  N : OUT STD_LOGIC;
220.                  Z : OUT STD_LOGIC;
221.                  C : OUT STD_LOGIC);
222.       END COMPONENT;
223.
224.       COMPONENT SREGDECOD
225.           PORT ( SREGIN : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
226.                  SIN : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
227.                  BITSETIN : IN STD_LOGIC;
228.                  SREGOUT : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
229.       END COMPONENT;
230.
231.       COMPONENT STACKMEM
232.          PORT (  CLK : IN STD_LOGIC;
233.                  STACKWR : IN STD_LOGIC;
234.                  STACKEN : IN STD_LOGIC;
235.                  STACKIN : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
236.                  STACKOUT : OUT STD_LOGIC_VECTOR (11 DOWNTO 0));
237.       END COMPONENT;
238.
239.       SIGNAL PCMXO     : STD_LOGIC_VECTOR(8 DOWNTO 0);
240.       SIGNAL PCMXO2    : STD_LOGIC_VECTOR(8 DOWNTO 0);
241.       SIGNAL PCNOU     : STD_LOGIC_VECTOR(8 DOWNTO 0);
242.       SIGNAL FLASHOUT  : STD_LOGIC_VECTOR(15 DOWNTO 0);
243.       SIGNAL INSTOUT   : STD_LOGIC_VECTOR(15 DOWNTO 0);
244.       SIGNAL FORMATAUX: STD_LOGIC_VECTOR(3 DOWNTO 0);
245.       SIGNAL DD        : STD_LOGIC_VECTOR(4 DOWNTO 0);
246.       SIGNAL RR        : STD_LOGIC_VECTOR(4 DOWNTO 0);
```

```
247.      SIGNAL KK        : STD_LOGIC_VECTOR(11 DOWNTO 0);
248.      SIGNAL SS        : STD_LOGIC_VECTOR(2 DOWNTO 0);
249.      SIGNAL BB        : STD_LOGIC_VECTOR(2 DOWNTO 0);
250.      SIGNAL NN        : STD_LOGIC_VECTOR(3 DOWNTO 0);
251.      SIGNAL REGMUXOUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
252.      SIGNAL REGRD1    : STD_LOGIC_VECTOR(7 DOWNTO 0);
253.      SIGNAL REGRD2    : STD_LOGIC_VECTOR(7 DOWNTO 0);
254.      SIGNAL ALUINPUTA: STD_LOGIC_VECTOR(11 DOWNTO 0);
255.      SIGNAL ALUINPUTB: STD_LOGIC_VECTOR(11 DOWNTO 0);
256.      SIGNAL BDECOUT   : STD_LOGIC_VECTOR(7 DOWNTO 0);
257.      SIGNAL ALURESULT: STD_LOGIC_VECTOR(11 DOWNTO 0);
258.      SIGNAL FLAGSOUT : STD_LOGIC_VECTOR(7 DOWNTO 0);
259.      SIGNAL REGALUOUT: STD_LOGIC_VECTOR(11 DOWNTO 0);
260.      SIGNAL REG2      : STD_LOGIC_VECTOR(11 DOWNTO 0);
261.      SIGNAL RAMOUTPUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
262.      SIGNAL MUXSOUT   : STD_LOGIC_VECTOR(2 DOWNTO 0);
263.      SIGNAL MUXRROUT : STD_LOGIC;
264.      SIGNAL DECRROUT : STD_LOGIC;
265.      SIGNAL MUXFLOUT : STD_LOGIC_VECTOR(7 DOWNTO 0);
266.      SIGNAL MUXINSTROUT : STD_LOGIC_VECTOR(15 DOWNTO 0);
267.      SIGNAL FLSELEOUT: STD_LOGIC;
268.      SIGNAL SREGDECOUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
269.      SIGNAL SREGSEROUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
270.      SIGNAL STACKREGOUT: STD_LOGIC_VECTOR(11 DOWNTO 0);
271.      SIGNAL II : STD_LOGIC;
272.      SIGNAL TT : STD_LOGIC;
273.      SIGNAL HH : STD_LOGIC;
274.      SIGNAL SSIGN : STD_LOGIC;
275.      SIGNAL VV : STD_LOGIC;
276.      SIGNAL NNEG : STD_LOGIC;
277.      SIGNAL ZZ : STD_LOGIC;
278.      SIGNAL CC : STD_LOGIC;
279.      SIGNAL RAMENABLE : STD_LOGIC;
280.      SIGNAL RAMOUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
281.      SIGNAL DATAREGAUX : STD_LOGIC_VECTOR(7 DOWNTO 0);
282.      SIGNAL DATAOUTAUX : STD_LOGIC_VECTOR(7 DOWNTO 0);
283.
284.  BEGIN
285.      PC:  PCREG PORT MAP (
286.              CLK => CLK,
287.              RESET => RESET,
288.              RESETWD => RESETWD,
289.              PCWRITE => PCWRITE,
290.              PC => PCMXO2,
291.              PCN => PCNOU
292.              );
293.      MEMFLASH : FLASHMEM PORT MAP(
294.              A => PCNOU,
295.              SPO => FLASHOUT
296.              );
297.
298.      REGINST : INSTREG PORT MAP(
299.              CLK => CLK,
300.              RESET => RESET,
301.              RESETWD => RESETWD,
302.              IRWRITE => IRWRITE,
303.              SPO => FLASHOUT,
304.              INSTR => INSTOUT
305.              );
```

```
306.      INSTMUX : MUXINSTR PORT MAP(
307.             INSTRSRC => INSTRSRC,
308.             INSTRREG => INSTOUT,
309.             INSTRMEM => FLASHOUT,
310.             INSTROUT => MUXINSTROUT
311.             );
312.
313.      DECODINST: INSTDECOD PORT MAP (
314.              INSTR => MUXINSTROUT,
315.              D => DD,
316.              R => RR,
317.              K => KK,
318.              S => SS,
319.              B => BB,
320.              N => NN,
321.              FORMAT => FORMATAUX
322.              );
323.
324.      BANCREGISTRES: BANCREG PORT MAP (
325.              CLK => CLK,
326.              WRITEEN => REGWRITE,
327.              A1 => DD,
328.              A2 => RR,
329.              A3 => DD,
330.              WD3 => REGMUXOUT,
331.              RD1 => REGRD1,
332.              RD2 => REGRD2
333.              );
334.
335.      INREG: IREG PORT MAP(
336.              CLK => CLK,
337.              RESET => RESET,
338.              RESETWD => RESETWD,
339.              IREGEN => IREGEN,
340.              IREGWR => IREGWR,
341.              DATAIN => DATAIN,
342.              DATAREG => DATAREGAUX);
343.
344.      OUTREG: OREG PORT MAP(
345.              CLK => CLK,
346.              RESET => RESET,
347.              RESETWD => RESETWD,
348.              OREGEN => OREGEN,
349.              DATAOUT => DATAOUTAUX,
350.              REGDATA => REGRD1);
351.
352.      ALUMUXA:MUXALUSRCA PORT MAP (
353.              PC => PCNOU,
354.              RD1 => REGRD1,
355.              ALUSRCA => ALUSRCA,
356.              ALUA => ALUINPUTA
357.              );
358.
359.      ALUMUXB : MUXALUSRCB PORT MAP (
360.            K => KK,
361.            RD2 => REGRD2,
362.            B => BDECOUT,
363.            ALUSRCB => ALUSRCB,
364.            ALUB => ALUINPUTB
```

```
365.          );
366.
367.      ALUCOMP: ALU PORT MAP (
368.            FLAGSIN => SREGSEROUT,
369.            ALUA => ALUINPUTA,
370.            ALUB => ALUINPUTB,
371.            ALUCONTROL => ALUCONTROL,
372.            N => NN,
373.            FLAGS => FLAGSOUT,
374.            ALUOUT => ALURESULT
375.          );
376.
377.      REGALU: ALUREG PORT MAP (
378.            ALUOUT => ALURESULT,
379.            CLK => CLK,
380.            RESET => RESET,
381.            RESETWD => RESETWD,
382.            ALUREGOUT => REGALUOUT,
383.            ALUREG2 => REG2
384.          );
385.
386.      PCMUX1: MUXPC1 PORT MAP (
387.            PCREG => REG2,
388.            PCALU => ALURESULT,
389.            PCSRC => PCSRC,
390.            PCMUXOUT => PCMXO
391.          );
392.
393.      PCMUX2: MUXPC2 PORT MAP (
394.            PCSTACK => STACKREGOUT,
395.            PC1M => PCMXO,
396.            PC2SRC => PC2SRC,
397.            PC2MUXOUT => PCMXO2
398.          );
399.
400.      MEMRAM: RAMMEM PORT MAP (
401.            CLK => CLK,
402.            RAMEN => RAMEN,
403.            MEMWRITE => MEMWRITE,
404.            A => REGRD2(5 DOWNTO 0),
405.            WD => REGRD1,
406.            RD_RAM => RAMOUTPUT,
407.            RAM32 => RAMOUT
408.          );
409.
410.      REGMUX: MUXREG PORT MAP (
411.            RD2 => REGRD2,
412.            K => KK,
413.            ALUOUT => REG2,
414.            RAMOUT => RAMOUTPUT,
415.            REGSRCIN => REGSRC,
416.            IREGOUT => DATAREGAUX,
417.            MUXREGOUT => REGMUXOUT
418.          );
419.
420.      MEMSTACK: STACKMEM PORT MAP (
421.            CLK => CLK,
422.            STACKWR => STACKWR,
423.            STACKEN => STACKEN,
424.            STACKIN => PCNOU,
425.            STACKOUT => STACKREGOUT
```

```
426.          );
427.
428.      SREGA: MUXSREGA PORT MAP (
429.            S => SS,
430.            SREGSRCA => SREGSRCA,
431.            SSSMUXOUT => MUXSOUT
432.          );
433.
434.      SREGB: MUXSREGB PORT MAP (
435.            BITSET => BITSET,
436.            RR => DECRROUT,
437.            SREGSRCB => SREGSRCB,
438.            BITSETMUXOUT => MUXRROUT
439.          );
440.
441.      SREGC: MUXSREGC PORT MAP (
442.            SREGDECOD => SREGDECOUT,
443.            SREGALU => FLAGSOUT,
444.            SREGSRCC => SREGSRCC,
445.            SREGMUXOUT => MUXFLOUT
446.          );
447.
448.      RRDEC: RRDECOD PORT MAP (
449.            RR => REGRD2,
450.            B => BB,
451.            RROUT => DECRROUT
452.          );
453.
454.      BDEC: BDECOD PORT MAP (
455.            B => BB,
456.            T => FLSELEOUT,
457.            BOUT => BDECOUT
458.          );
459.
460.      SREGDEC: SREGDECOD PORT MAP (
461.            SREGIN => SREGSEROUT,
462.            SIN => MUXSOUT,
463.            BITSETIN => MUXRROUT,
464.            SREGOUT => SREGDECOUT
465.          );
466.
467.      STATREG: SREG PORT MAP (
468.            RESET => RESET,
469.            RESETWD => RESETWD,
470.            SREGIN => MUXFLOUT,
471.            SREGOUT => SREGSEROUT,
472.            CLK => CLK,
473.            SREGLOAD => SREGLOAD,
474.            I => II,
475.            T => TT,
476.            H => HH,
477.            S => SSIGN,
478.            V => VV,
479.            N => NNEG,
480.            Z => ZZ,
481.            C => CC
482.          );
```

```
483.       FLAGMUX : MUXFLAGS PORT MAP(
484.             I => II,
485.             T => TT,
486.             H => HH,
487.             S => SSIGN,
488.             V => VV,
489.             N => NNEG,
490.             Z => ZZ,
491.             C => CC,
492.             MUXFLAGOUT => FLSELEOUT,
493.             FLAGSELECT => FLAGSELECT
494.             );
495.
496.     FORMAT <= FORMATAUX;
497.     N <= NN;
498.     STATUSREG <= SREGSEROUT;
499.     S <= SS;
500.     DATAOUT <= DATAOUTAUX;
501.   END STRUCT;
```

D.27 Unitat de Control (*MAINCONTROLLER*)

```
0. library IEEE;
1. use IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
3. USE IEEE.NUMERIC_STD.ALL;
4.
5. ENTITY MAINCONTROLLER IS
6.      PORT (CLK, RESET, RESETWD: IN STD_LOGIC;
7.          N,FORMAT: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
8.          S : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
9.          STATUSREG: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
10.
11.             ALUCONTROL, ALUSRCB, FLAGSELECT: OUT STD_LOGIC_VECTOR(2
   DOWNTO 0);
12.             REGSRC: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
13.             ALUSRCA: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
14.             IRWRITE, PCWRITE, SREGSRCA, SREGSRCB:OUT STD_LOGIC;
15.             SREGSRCC, MEMWRITE, STACKEN, STACKWR, RAMEN: OUT STD_LOGIC;
16.             PCSRC, PC2SRC, REGWRITE, SREGLOAD, BITSET: OUT STD_LOGIC;
17.             IREGEN, IREGWR, OREGEN: OUT STD_LOGIC;
18.             INSTRSRC, FLAGWD : OUT STD_LOGIC);
19.   END MAINCONTROLLER;
20.
21.   ARCHITECTURE BEHAVIORAL OF MAINCONTROLLER IS
22.
23.   TYPE STATE_TYPE IS (S0, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
24.   S12, S13, S14, S15, S16, S17, S18, S19, S20, S21, S22, S23, S24, S25,
25.   S26, S27, S28, S0N, S1N);
26.   SIGNAL ESTAT : STATE_TYPE;
27.   SIGNAL VEC_CONTROL: STD_LOGIC_VECTOR(32 DOWNTO 0);
28.   SIGNAL SYS: STD_LOGIC;
29.   SIGNAL SYSCALL_AUX: STD_LOGIC := '0';
30.   SIGNAL FORMATAUX, NAUX : STD_LOGIC_VECTOR(3 DOWNTO 0);
31.   SIGNAL STATUSREGAUX: STD_LOGIC_VECTOR(7 DOWNTO 0);
32.   SIGNAL SAUX: UNSIGNED(2 DOWNTO 0);
```

```
33.   TYPE VECTORSCONTROL IS ARRAY(0 TO 30) OF STD_LOGIC_VECTOR(32 DOWNTO
   0);
34.   SIGNAL CV : VECTORSCONTROL := (
35.   "00010111000000010110000000000000", --S0
36.   "00000000000000000000000000000000", --S1
37.   "00000000000011000011000100000000", --S2
38.   "00000000000000000000000000000010", --S3
39.   "00000000101100000000000000000000", --S4
40.   "00000000000000000000000010000010", --S5
41.   "00000000001011001000001000000000", --S6
42.   "00000000101000000000000000000000", --S7
43.   "00000000100000000000000000000000", --S8
44.   "00000000000001011110000000000000", --S9
45.   "00000010000000000000000000000000", --S10
46.   "00000000000011000001000100000000", --S11
47.   "00000000000011000010000100000000", --S12
48.   "00000010000000000000000000000001", --S13
49.   "00000000000001000110000000001000", --S14
50.   "00000000000011100100000001100000", --S15
51.   "00000000000011100101000001100000", --S16
52.   "00000000000000000000000000000000", --S17
53.   "00000000000000000001011000000000", --S18
54.   "10000000110000000000000000000000", --S19
55.   "00000000000000000000111100000000", --S20 bitset = n
56.   "00000000000000000000111000000000"
57.   "00000000100100000000000000000000", --S21
58.   "00000010000000000110000000001000", --S22
59.   "00000011000000000110000000001100", --S23
60.   "00000000000000000000000000001001", --S24
61.   "00000010000001000000000000000000", --S25 (S->6 DT 4)
62.   "00000000000000000000000000000000", --S26
63.   "00100000000000000000000000000000", --S27
64.   "11000000000000000000000000000000", --S28
65.   "00011101000000010110000000000000", --S0N
66.   "00001000000000000000000000000000");--S1N
67.
68.   begin
69.
70.   FSM: PROCESS(CLK)
71.   BEGIN
72.   FORMATAUX <= FORMAT;
73.   NAUX <= N;
74.   STATUSREGAUX <= STATUSREG;
75.   SAUX <= UNSIGNED(S);
76.
77.       IF RISING_EDGE(CLK) THEN
78.           IF (RESET ='1') OR RESETWD = '1' THEN
79.               ESTAT <= S0;
80.               VEC_CONTROL <= CV(0);                          --FETCH
81.           ELSE
82.               CASE ESTAT IS
83.                   WHEN S0 =>
84.                       ESTAT <= S1;
85.                       VEC_CONTROL <= CV(1);                  --DECODE
86.                   WHEN S1 =>
87.                       IF FORMATAUX = "0011" THEN
88.                           ESTAT <= S2;
89.                           VEC_CONTROL <= CV(2);
90.                       ELSIF FORMATAUX = "0000" THEN
91.                           IF NAUX = "1011" THEN
```

```
92.                         ESTAT <= S8;
93.                         VEC_CONTROL <= CV(8);
94.                     ELSE
95.                         ESTAT <= S6;
96.                         VEC_CONTROL <= CV(6);
97.                     END IF;
98.                 ELSIF FORMATAUX = "0001" THEN
99.                     ESTAT <= S11;
100.                    VEC_CONTROL <= CV(11);
101.                ELSIF FORMATAUX = "0010" OR FORMATAUX = "1001"
102.                                                        THEN
103.                    ESTAT <= S12;
104.                    VEC_CONTROL <= CV(12);
105.                ELSIF FORMATAUX = "0100" THEN
106.                    IF N = "0000" THEN
107.                        IF STATUSREGAUX(6) = '0' THEN
108.                            ESTAT <= S15;
109.                            VEC_CONTROL <= CV(15);
110.                        ELSE
111.                            ESTAT <= S16;
112.                            VEC_CONTROL <= CV(16);
113.                        END IF;
114.                    ELSE
115.                        ESTAT <= S17;
116.                        VEC_CONTROL <= CV(17);
117.                    END IF;
118.                ELSIF FORMATAUX = "0101" THEN
119.                    IF N = "0000" THEN --IN
120.                        ESTAT <= S28;
121.                        VEC_CONTROL <= CV(28);
122.                    ELSE
123.                        ESTAT <= S27;
124.                        VEC_CONTROL <= CV(27);
125.                    END IF;
126.
127.                ELSIF FORMATAUX = "0111" THEN
128.                    ESTAT <= S20;
129.                    IF N <= "0000" THEN
130.                        VEC_CONTROL <= CV(20)(32 DOWNTO 9) & '1'
131.                                       & CV(20)(7 DOWNTO 0) ;
132.                    ELSIF N <= "0001" THEN
133.                        VEC_CONTROL <= CV(20)(32 DOWNTO 9) & '0'
134.                                       & CV(20)(7 DOWNTO 0) ;
135.                    END IF;
136.                ELSIF FORMATAUX = "1000" THEN
137.                    ESTAT <= S21;
138.                    VEC_CONTROL <= CV(21);
139.                ELSIF FORMATAUX = "1011" THEN
140.                    IF N = "0000" THEN
141.                        ESTAT <= S22;
142.                        VEC_CONTROL <= CV(22);
143.                    ELSIF N = "0001" THEN
144.                        ESTAT <= S23;
145.                        VEC_CONTROL <= CV(23);
146.                    END IF;
147.                ELSIF FORMATAUX = "1100" THEN
148.                    ESTAT <= S24;
149.                    VEC_CONTROL <= CV(24);
150.                ELSIF FORMATAUX = "1101" THEN
151.                    ESTAT <= S25;
152.                    VEC_CONTROL <= CV(25)(32 DOWNTO 7) & S &
```

```
153.                                                CV(25)(3 DOWNTO 0) ;
154.
155.                        ELSIF FORMATAUX <= "1111" OR FORMATAUX <= "1110"
156.                                                                    THEN
157.                            ESTAT <= S26;
158.                            VEC_CONTROL <= CV(26);
159.                        ELSE
160.                            VEC_CONTROL <= (OTHERS => '0');
161.                        END IF;
162.                    WHEN S2 =>
163.                        IF N = "0000" THEN
164.                            ESTAT <= S3;
165.                            VEC_CONTROL <= CV(3);
166.                        ELSIF N = "0010" THEN
167.                            ESTAT <= S5;
168.                            VEC_CONTROL <= CV(5);
169.                        END IF;
170.                    WHEN S3 =>
171.                        ESTAT <= S4;
172.                        VEC_CONTROL <= CV(4);
173.                    WHEN S4 =>
174.                        ESTAT <= S0;
175.                        VEC_CONTROL <= CV(0);
176.                    WHEN S5 =>
177.                        ESTAT <= S0;
178.                        VEC_CONTROL <= CV(0);
179.                    WHEN S6 =>
180.                        IF N = "0100" THEN
181.                            IF STATUSREGAUX(1) = '1' THEN
182.                                ESTAT <= S9;
183.                                VEC_CONTROL <= CV(9);
184.                            ELSE
185.                                ESTAT <= S1;
186.                                VEC_CONTROL <= CV(1);
187.                            END IF;
188.                        ELSE
189.                            ESTAT <= S7;
190.                            VEC_CONTROL <= CV(7);
191.                        END IF;
192.                    WHEN S7 =>
193.                        ESTAT <= S0;
194.                        VEC_CONTROL <= CV(0);
195.
196.                    WHEN S8 =>
197.                        ESTAT <= S0;
198.                        VEC_CONTROL <= CV(0);
199.                    WHEN S9 =>
200.                        ESTAT <= S10;
201.                        VEC_CONTROL <= CV(10);
202.                    WHEN S10 =>
203.                        IF FORMAT = "1100" OR FORMAT = "1101" OR
204.                  FORMAT = "0111" OR (FORMAT = "0000" AND N = "0100") THEN
205.                            ESTAT <= S0N;
206.                            VEC_CONTROL <= CV(29);
207.                        ELSE
208.                            ESTAT <= S0;
209.                            VEC_CONTROL <= CV(0);
210.                        END IF;
211.                    WHEN S0N =>
212.                        ESTAT <= S1N;
```

```
213.                    VEC_CONTROL <= CV(30);
214.
215.                WHEN S11 =>
216.                    ESTAT <= S7;
217.                    VEC_CONTROL <= CV(7);
218.                WHEN S12 =>
219.                    ESTAT <= S7;
220.                    VEC_CONTROL <= CV(7);
221.                WHEN S13 =>
222.                    ESTAT <= S0N;
223.                    VEC_CONTROL <= CV(29);
224.
225.                WHEN S14 =>
226.                    ESTAT <= S10;
227.                    VEC_CONTROL <= CV(10);
228.
229.                WHEN S15 =>
230.                    ESTAT <= S7;
231.                    VEC_CONTROL <= CV(7);
232.                WHEN S16 =>
233.                    ESTAT <= S7;
234.                    VEC_CONTROL <= CV(7);
235.                WHEN S17 =>
236.                    IF N = "0001" THEN
237.                        ESTAT <= S18;
238.                        VEC_CONTROL <= CV(18);
239.                    ELSE
240.                        ESTAT <= S9;
241.                        VEC_CONTROL <= CV(9);
242.                    END IF;
243.                WHEN S18 =>
244.                    ESTAT <= S0;
245.                    VEC_CONTROL <= CV(0);
246.
247.                WHEN S19 =>
248.                    ESTAT <= S0;
249.                    VEC_CONTROL <= CV(0);
250.
251.                WHEN S20 =>
252.                    ESTAT <= S0;
253.                    VEC_CONTROL <= CV(0);
254.                WHEN S21 =>
255.                    ESTAT <= S0;
256.                    VEC_CONTROL <= CV(0);
257.                WHEN S22 =>
258.                    ESTAT <= S10;
259.                    VEC_CONTROL <= CV(10);
260.                WHEN S23 =>
261.                    ESTAT <= S26;
262.                    VEC_CONTROL <= CV(26);
263.                WHEN S24 =>
264.                    ESTAT <= S13;
265.                    VEC_CONTROL <= CV(13);
266.                WHEN S25 =>
267.                    --SAUX <= UNSIGNED(S);
268.                    IF (STATUSREGAUX(TO_INTEGER(SAUX)) = '1'
269.                                                AND N = "0000") OR
270.                        (STATUSREGAUX(TO_INTEGER(SAUX)) = '0'
271.                                                AND N = "0001")THEN
272.                        ESTAT <= S14;
273.                        VEC_CONTROL <= CV(14);
```

```
274.
275.                       ELSE
276.                           ESTAT <= S0;
277.                           VEC_CONTROL <= CV(0);
278.                       END IF;
279.                   WHEN S26 =>
280.                       IF FORMATAUX = "1111" OR FORMATAUX = "1011" THEN
281.                           ESTAT <= S0;
282.                           VEC_CONTROL <= CV(0);
283.                       END IF;
284.
285.                   WHEN S27 =>
286.                       ESTAT <= S0;
287.                       VEC_CONTROL <= CV(0);
288.
289.                   WHEN S28 =>
290.                       ESTAT <= S19;
291.                       VEC_CONTROL <= CV(19);
292.
293.                   WHEN S1N =>
294.                       IF FORMATAUX = "0011" THEN
295.                               ESTAT <= S2;
296.                               VEC_CONTROL <= CV(2);
297.                           ELSIF FORMATAUX = "0000" THEN
298.                               IF NAUX = "1011" THEN
299.                                   ESTAT <= S8;
300.                                   VEC_CONTROL <= CV(8);
301.                               ELSE
302.                                   ESTAT <= S6;
303.                                   VEC_CONTROL <= CV(6);
304.                               END IF;
305.                           ELSIF FORMATAUX = "0001" THEN
306.                               ESTAT <= S11;
307.                               VEC_CONTROL <= CV(11);
308.                           ELSIF FORMATAUX = "0010" OR
309.                                                 FORMATAUX = "1001" THEN
310.                               ESTAT <= S12;
311.                               VEC_CONTROL <= CV(12);
312.                           ELSIF FORMATAUX = "0100" THEN
313.                               IF N = "0000" THEN
314.                                   IF STATUSREGAUX(6) = '0' THEN
315.                                       ESTAT <= S15;
316.                                       VEC_CONTROL <= CV(15);
317.                                   ELSE
318.                                       ESTAT <= S16;
319.                                       VEC_CONTROL <= CV(16);
320.                                   END IF;
321.                               ELSE
322.                                   ESTAT <= S17;
323.                                   VEC_CONTROL <= CV(17);
324.                               END IF;
325.
326.                           ELSIF FORMATAUX = "0101" THEN
327.                           IF N = "0000" THEN --IN
328.                               ESTAT <= S28;
329.                               VEC_CONTROL <= CV(28);
330.                           ELSE
331.                               ESTAT <= S27;
332.                               VEC_CONTROL <= CV(27);
333.                           END IF;
```

```
334.
335.                         ELSIF FORMATAUX = "0111" THEN
336.                             ESTAT <= S20;
337.                             IF N <= "0000" THEN
338.                             VEC_CONTROL <= CV(20)(32 DOWNTO 9) & '1'
339.                                           & CV(20)(7 DOWNTO 0) ;
340.                             ELSIF N <= "0001" THEN
341.                             VEC_CONTROL <= CV(20)(32 DOWNTO 9) & '1'
342.                                           & CV(20)(7 DOWNTO 0) ;
343.                             END IF;
344.                         ELSIF FORMATAUX = "1000" THEN
345.                             ESTAT <= S21;
346.                             VEC_CONTROL <= CV(21);
347.                         ELSIF FORMATAUX = "1011" THEN
348.                             IF N = "0000" THEN
349.                                 ESTAT <= S22;
350.                                 VEC_CONTROL <= CV(22);
351.                             ELSIF N = "0001" THEN
352.                                 ESTAT <= S23;
353.                                 VEC_CONTROL <= CV(23);
354.                             END IF;
355.                         ELSIF FORMATAUX = "1100" THEN
356.                             ESTAT <= S24;
357.                             VEC_CONTROL <= CV(24);
358.                         ELSIF FORMATAUX = "1101" THEN
359.                             ESTAT <= S25;
360.                             VEC_CONTROL(32 DOWNTO 0) <= CV(25)
361.                              (32 DOWNTO 7) & S & CV(25)(3 DOWNTO 0) ;
362.                         ELSIF FORMATAUX <= "1111" THEN
363.                             ESTAT <= S26;
364.                             VEC_CONTROL <= CV(26);
365.                         ELSE
366.                             VEC_CONTROL <= (OTHERS => '0');
367.                         END IF;
368.                 END CASE;
369.             END IF;
370.         END IF;
371.     END PROCESS;
372.     IREGEN <= VEC_CONTROL(32);
373.     IREGWR <= VEC_CONTROL(31);
374.     OREGEN <= VEC_CONTROL(30);
375.     FLAGWD <= VEC_CONTROL(29);
376.     INSTRSRC <= VEC_CONTROL(28);
377.     IRWRITE <= VEC_CONTROL(27);
378.     PCWRITE <= VEC_CONTROL(26);
379.     PCSRC <= VEC_CONTROL(25);
380.     REGWRITE <= VEC_CONTROL(24);
381.     REGSRC<= VEC_CONTROL(23 DOWNTO 21);
382.     ALUSRCA <= VEC_CONTROL(20 DOWNTO 19);
383.     ALUSRCB <= VEC_CONTROL(18 DOWNTO 16);
384.     ALUCONTROL <= VEC_CONTROL(15 DOWNTO 13);
385.     SREGSRCA <= VEC_CONTROL(12);
386.     SREGSRCB <= VEC_CONTROL(11);
387.     SREGSRCC <= VEC_CONTROL(10);
388.     SREGLOAD <= VEC_CONTROL(9);
389.     BITSET <= VEC_CONTROL(8);
390.     MEMWRITE <= VEC_CONTROL(7);
391.     FLAGSELECT <= VEC_CONTROL(6 DOWNTO 4);
392.     STACKEN <= VEC_CONTROL(3);
393.     STACKWR <= VEC_CONTROL(2);
394.     RAMEN <= VEC_CONTROL(1);
```

```
395.       PC2SRC <= VEC_CONTROL(0);
396.   END BEHAVIORAL;
```

D.28 Comptador *Watchdog* (*WATCHDOG*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2. USE IEEE.STD_LOGIC_UNSIGNED.ALL;
3.
4. ENTITY WATCHDOG IS
5.     PORT ( CLK : IN STD_LOGIC;
6.            RESET : IN STD_LOGIC;
7.            RESETWD : OUT STD_LOGIC;
8.            WDFLAG : IN STD_LOGIC);
9. END WATCHDOG;
10.
11.   ARCHITECTURE BEHAVIORAL OF WATCHDOG IS
12.   SIGNAL WDAUX : STD_LOGIC_VECTOR(18 DOWNTO 0);
13.   BEGIN
14.   PROCESS(CLK)
15.   BEGIN
16.       IF RISING_EDGE(CLK) THEN
17.           IF RESET = '1' OR WDFLAG = '1' THEN
18.               WDAUX <= (OTHERS => ('0'));
19.           ELSE
20.               WDAUX <= WDAUX + '1';
21.               IF WDAUX = "11000011010011111111" THEN --399999
22.                   RESETWD <= '1';
23.               ELSE
24.                   RESETWD <= '0';
25.               END IF;
26.           END IF;
27.       END IF;
28.   END PROCESS;
29.   END BEHAVIORAL;
```

D.29 Microprocessador Complet (*TOP*)

```
0. LIBRARY IEEE;
1. USE IEEE.STD_LOGIC_1164.ALL;
2.
3.
4. ENTITY TOP IS
5.     PORT( CLK_IN, RESET, RESETCLK: IN STD_LOGIC;
6.           DATAOUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
7.           DATAIN : IN STD_LOGIC_VECTOR(7 DOWNTO 0));
8. END TOP;
9.
10.   ARCHITECTURE BEHAVIORAL OF TOP IS
11.
12.   COMPONENT WATCHDOG
13.       PORT( CLK : IN STD_LOGIC;
14.             RESET : IN STD_LOGIC;
15.             RESETWD : OUT STD_LOGIC;
16.             WDFLAG : IN STD_LOGIC);
17.   END COMPONENT;
18.
19.   COMPONENT CLK_WIZ_0
```

```
20.         PORT(    CLK_IN : IN STD_LOGIC;
21.                  CLK : OUT STD_LOGIC;
22.                  RESET : IN STD_LOGIC;
23.                  LOCKED : OUT STD_LOGIC);
24.     END COMPONENT;
25.     COMPONENT DATAPATH
26.         PORT(
27.             --Senyals de Top
28.             DATAIN : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
29.             DATAOUT: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
30.             CLK, RESET, RESETWD: IN STD_LOGIC;
31.             --Senyals de la Unitat de Control
32.             ALUCONTROL, ALUSRCB: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
33.             FLAGSELECT: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
34.             REGSRC: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
35.             ALUSRCA: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
36.             IRWRITE, PCWRITE, SREGSRCA, SREGSRCB :IN STD_LOGIC;
37.             SREGSRCC, SREGLOAD, BITSET, MEMWRITE, STACKWR,: IN STD_LOGIC;
38.             PCSRC, PC2SRC, REGWRITE, STACKEN, RAMEN: IN STD_LOGIC;
39.             INSTRSRC, IREGEN, IREGWR, OREGEN: IN STD_LOGIC;
40.             FORMAT, N: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
41.             S : OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
42.             STATUSREG: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
43.     END COMPONENT;
44.
45.     COMPONENT MAINCONTROLLER
46.         PORT (CLK, RESET, RESETWD: IN STD_LOGIC;
47.             N,FORMAT: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
48.             S : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
49.             STATUSREG: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
50.             ALUCONTROL, ALUSRCB: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
51.             FLAGSELECT: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
52.             REGSRC: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
53.             ALUSRCA: OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
54.             IRWRITE, PCWRITE, SREGSRCA, SREGSRCB:OUT STD_LOGIC;
55.             SREGSRCC, SREGLOAD, BITSET, MEMWRITE, STACKEN: OUT STD_LOGIC;
56.             PCSRC, PC2SRC, REGWRITE, STACKWR, RAMEN: OUT STD_LOGIC;
57.             IREGEN, IREGWR, OREGEN: OUT STD_LOGIC;
58.             INSTRSRC, FLAGWD : OUT STD_LOGIC);
59.     END COMPONENT;
60.
61.             SIGNAL CLK: STD_LOGIC;
62.             SIGNAL RESETWD: STD_LOGIC := '0';
63.             SIGNAL ALUCONTROL, ALUSRCB: STD_LOGIC_VECTOR(2 DOWNTO 0);
64.             SIGNAL FLAGSELECT: STD_LOGIC_VECTOR(2 DOWNTO 0);
65.             SIGNAL REGSRC : STD_LOGIC_VECTOR(2 DOWNTO 0);
66.             SIGNAL ALUSRCA: STD_LOGIC_VECTOR(1 DOWNTO 0);
67.             SIGNAL IRWRITE, SREGSRCA, SREGSRCB, FLAGWD : STD_LOGIC;
68.             SIGNAL STACKEN, STACKWR, RAMEN, INSTRSRC: STD_LOGIC;
69.             SIGNAL SREGSRCC, SREGLOAD, PCWRITE, PCSRC : STD_LOGIC;
70.             SIGNAL BITSET, MEMWRITE, REGWRITE, PC2SRC : STD_LOGIC;
71.             SIGNAL IREGEN, IREGWR, OREGEN: STD_LOGIC;
72.             SIGNAL NOUT,FORMATOUT:  STD_LOGIC_VECTOR(3 DOWNTO 0);
73.             SIGNAL SOUT :  STD_LOGIC_VECTOR(2 DOWNTO 0);
74.             SIGNAL STATUSREGOUT : STD_LOGIC_VECTOR(7 DOWNTO 0);
75.             SIGNAL DATAOUTAUX : STD_LOGIC_VECTOR(7 DOWNTO 0);
76.     BEGIN
77.
78.         WD:      WATCHDOG PORT MAP(
79.                  CLK => CLK,
80.                  RESET => RESET,
```

```
81.                 RESETWD => RESETWD,
82.                 WDFLAG => FLAGWD);


83.         CLKMOD: CLK_WIZ_0 PORT MAP(
84.                 CLK_IN => CLK_IN,
85.                 CLK => CLK,
86.                 RESET => RESETCLK);
87.
88.         DATAPATHTINY : DATAPATH PORT MAP(
89.                 CLK => CLK,
90.                 RESET => RESET,
91.                 RESETWD => RESETWD,
92.                 ALUCONTROL => ALUCONTROL,
93.                 ALUSRCB => ALUSRCB,
94.                 FLAGSELECT => FLAGSELECT,
95.                 IRWRITE => IRWRITE,
96.                 PCWRITE => PCWRITE,
97.                 PCSRC => PCSRC,
98.                 PC2SRC => PC2SRC,
99.                 REGWRITE => REGWRITE,
100.                ALUSRCA => ALUSRCA,
101.                SREGSRCA => SREGSRCA,
102.                SREGSRCB => SREGSRCB,
103.                SREGSRCC => SREGSRCC,
104.                SREGLOAD => SREGLOAD,
105.                BITSET => BITSET,
106.                MEMWRITE => MEMWRITE,
107.                STACKEN => STACKEN,
108.                STACKWR => STACKWR,
109.                RAMEN => RAMEN,
110.                FORMAT => FORMATOUT,
111.                REGSRC => REGSRC,
112.                INSTRSRC => INSTRSRC,
113.                N => NOUT,
114.                STATUSREG => STATUSREGOUT,
115.                S => SOUT,
116.                IREGEN => IREGEN,
117.                IREGWR => IREGWR,
118.                OREGEN => OREGEN,
119.                DATAIN => DATAIN,
120.                DATAOUT => DATAOUTAUX);
121.
122.        CONTROLLER: MAINCONTROLLER PORT MAP (
123.              CLK => CLK,
124.              RESET => RESET,
125.              RESETWD => RESETWD,
126.              N => NOUT,
127.              FORMAT => FORMATOUT,
128.              S => SOUT,
129.              STATUSREG => STATUSREGOUT,
130.              ALUCONTROL => ALUCONTROL,
131.              ALUSRCB => ALUSRCB,
132.              FLAGSELECT => FLAGSELECT,
133.              REGSRC => REGSRC,
134.              IRWRITE => IRWRITE,
135.              PCWRITE => PCWRITE,
136.              PCSRC => PCSRC,
137.              PC2SRC => PC2SRC,
138.              REGWRITE => REGWRITE,
```

```
139.            ALUSRCA => ALUSRCA,
140.            SREGSRCA => SREGSRCA,
141.            SREGSRCB => SREGSRCB,
142.            SREGSRCC => SREGSRCC,
143.            SREGLOAD => SREGLOAD,
144.            BITSET => BITSET,
145.            MEMWRITE => MEMWRITE,
146.            STACKEN => STACKEN,
147.            STACKWR => STACKWR,
148.            RAMEN => RAMEN,
149.            INSTRSRC => INSTRSRC,
150.            FLAGWD => FLAGWD,
151.            IREGEN => IREGEN,
152.            IREGWR => IREGWR,
153.            OREGEN => OREGEN);
154.  DATAOUT <= DATAOUTAUX;
155.  END BEHAVIORAL;
```

D.30 *TestBench* del processador (*TOP_tb*)

```
0. library IEEE;
1. use IEEE.STD_LOGIC_1164.ALL;
2.
3. entity TOP_tb is
4. end TOP_tb;
5. architecture Behavioral of TOP_tb is
6. COMPONENT TOP
7.     PORT( CLK_IN, RESET, RESETCLK: IN STD_LOGIC;
8.           DATAIN: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
9.           DATAOUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10.       END COMPONENT;
11.      signal clk_in, RESET, RESETCLK: std_logic := '0';
12.      SIGNAL DATAIN,DATAOUT: STD_LOGIC_VECTOR(7 DOWNTO 0);
13.      constant clk_period : time := 10 ns;
14.   BEGIN
15.      uut: TOP PORT MAP (
16.             clk_in => clk_in,
17.             reset => reset,
18.             RESETCLK => RESETCLK,
19.             DATAIN => DATAIN,
20.             DATAOUT => DATAOUT
21.             );
22.      CLK_process: process
23.              begin
24.                  CLK_in <= '0';
25.                  wait for CLK_period/2;
26.                  CLK_in <= '1';
27.                  wait for CLK_period/2;
28.              end process;
29.             -- Stimulus process
30.      stim_proc: process
31.           begin
32.              reset <= '1';
33.              wait for clk_period*4;
34.              reset <= '0';
35.              datain <= "01011100";
36.           wait;
37.           end process;
38.   end Behavioral;
```

Annex E: Fitxer de Restriccions

```
## Fitxer de restriccions
## Autor: Tomàs Hudson
## Data: 05/12/2019

## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports {
CLK_IN }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10 -waveform {0 5} [get_ports
{CLK_IN}];
#create_clock -add -name clk80M      -period 12.5    -waveform{0 5}
[get_ports {CLK}];

##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 } [get_ports {
CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {
RESET }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports {
RESETCLK }]; #IO_L4N_T0_D05_14 Sch=btnu

##Pmod Header JA

set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[0] }]; #IO_L20N_T3_A19_15 Sch=ja[1]
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[1] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[2] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[3] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[4] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[5] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[6] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {
DATAIN[7] }]; #IO_L22P_T3_A17_15 Sch=ja[10]


##Pmod Header JB

set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[0] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[1] }]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[2] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[3] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[4] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[5] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
```

```
set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[6] }]; #IO_0_15 Sch=jb[9]
set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports {
DATAOUT[7] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]
```