Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Electrònica Industrial i Automàtica

Títol: Estudi i implementació de sistemes de comunicació sèrie utilitzant dispositius lògics programables: la placa DE0-Nano i el llenguatge Verilog

Document: 1. Memòria

Alumne: Naïm Baulenas Sanglas

Tutor: Lluís Pacheco Valls
Departament: Arquitectura i Tecnologia de Computadors
Àrea: Arquitectura i Tecnologia de Computadors

Convocatòria (mes/any): setembre/2017

ÍNDEX

1 INTRODUCCIÓ 4
1.1 Antecedents 4
1.2 Objecte 4
1.3 Especificacions i abast 5
2 DE0-NANO D'ALTERA 7
3 INTRODUCCIÓ AL VERILOG 12
3.1 Tipus de variables i operacions amb Verilog 13
3.2 Processos 15
3.3 Tipus d'estructures 17
3.4 Disseny pel sistema de portes 19
3.4.1 Multiplexor amb portes lògiques 19
3.5 Disseny pel sistema de transferència de registres 20
3.5.1 Comptador 21
3.5.2 Registre de desplaçament 22
3.6 Disseny pel sistema de comportament 24
4 COMUNICACIONS 25
4.1 Comunicacions sèrie 26
4.1.1 Comunicació SPI 27
4.1.2 Comunicació I2C 31
4.1.3 Comunicació SPI combinada amb la comunicació I2C 34
5 COMUNICACIONS EN SÈRIE A LA PLACA DE0-Nano 35
5.1 La memòria EEPROM 24LC02B 36
5.1.1 Comunicació de la memòria 24LC02B amb la Cyclone IV 36
5.2 El convertidor analògic a digital 128S022 38
5.2.1 Comunicació del convertidor analògic a digital 128S022 amb la Cyclone IV .. 39
5.3 L'acceleròmetre ADXL345 40
5.3.1 Comunicació de l'acceleròmetre amb la Cyclone IV 41
6 PROGRAMA DE0_NANO_EEPROM 44

6.1 Modificació del programa DE0_NANO_EEPROM 49
6.2 Proves, simulacions i resultats 51
7 PROGRAMA DE0_NANO 54
7.1 Mòdul DE0_NANO 54
7.2 Mòdul SPIPLL 56
7.3 Mòdul ADC_CTRL 57
7.4 Proves, simulacions i resultats del programa DE0_NANO 59
8 PROGRAMA DE0_NANO_G_SENSOR 62
8.1 Configuració de l'acceleròmetre ADXL345 62
8.1.1 Configuració de l'adreça 0x24 63
8.1.2 Configuració de l'adreça 0x25 63
8.1.3 Configuració de l'adreça 0x26 64
8.1.4 Configuració de l'adreça 0x27 64
8.1.5 Configuració de l'adreça 0x28 65
8.1.6 Configuració de l'adreça 0x29 65
8.1.7 Configuració de l'adreça 0x2C 66
8.1.8 Configuració de l'adreça 0x2D 66
8.1.9 Configuració de l'adreça 0x31 67
8.1.10 Configuració de l'adreça 0x2E 68
8.1.11 Configuració de l'adreça 0x2F 68
8.2 Mòdul DE0_NANO_G_Sensor 68
8.3 Mòdul reset_delay 71
8.4 Mòdul spipll 73
8.5 Mòdul spi_ee_config 74
8.5.1 Mòdul spi_controler 81
8.6 Mòdul led_driver 85
8.7 Proves, simulacions i resultats del programa DE0_NANO_G_Sensor 88
9 RESUM DEL PRESSUPOST 91
10 CONCLUSIONS 92

11 RELACIÓ DE DOCUMENTS 95
12 BIBLIOGRAFIA 96
13 GLOSSARI 98
A PROGRAMES 101
A.1 DE0_NANO_EEPROM 101
A.2 DE0_NANO_EEPROM_V2 104
A.2.1 I2cpll 105
A.2.2 EEPROM_CONTROL 107
A.3 DE0_NANO 110
A.3.1 SPIPLL 111
A.3.2 ADC_CTRL 114
A.4 DE0_NANO_G_Sensor 116
A.4.1 Spi_param 118
A.4.2 Spi_ee_config 119
A.4.3 Spi_controler 122
A.4.4 Reset_delay 123
A.4.5 Led_driver 124
A.4.6 Spipll 125
A.5 Comptador 128
A.6 Flip flop d 129
A.7 Multiplexor 129
A.8 Registre_d 130
A.9 DE0_Nano_PLL 131

1 INTRODUCCIÓ

Amb aquest projecte es pretén analitzar una part del funcionament de la placa DE0-Nano perquè qualsevol persona interessada en el desenvolupament de projectes d'investigació o aplicacions electròniques li serveixi com a guia bàsica.

1.1 Antecedents

L'empresa Altera és una companyia pionera en dispositius programables. Aquesta empresa ofereix al consumidor un gran ventall de productes aptes per a diferents aplicacions. Alguns dels productes que comercialitza són: FPGAs, CPLDs i ASICs. Tots aquests tenen diferents qualitats per adaptar-se a les necessitats de l'usuari.

Altera ofereix la sèrie de productes DE, pensats principalment com a material educatiu i que permet iniciar-se en el l'ús de les FPGA. L'Escola Politècnica Superior de la Universitat de Girona disposa de la placa DE1 per a realitzar les pràctiques de l'assignatura d'Electrònica Digital i Dispositius Programables i recentment ha comprat la DE0-Nano que pot servir per desenvolupar diferents projectes i aplicacions electròniques.

A Europa el llenguatge més utilitzat per a programar els productes DE és el VHDL i a Amèrica és el Verilog.

El meu punt de partida d'aquest projecte és el que he après en l'assignatura d'Electrònica Digital i Dispositius Programables on s'han explicat diferents circuits digitals i realitzat diferents exercicis amb VHDL utilitzant la placa DE1 però no el llenguatge Verilog, llenguatge que he tingut que estudiar per a el desenvolupament d'aquest projecte.

1.2 Objecte

La placa DE0-Nano ofereix moltes possibilitats, des de les aplicacions bàsiques, com la d'encendre i apagar un LED fins a crear una consola per jugar a videojocs NES. Tot això és possible perquè la placa incorpora components bàsics com pot ser una memòria o polsadors, a més a més, disposa de dos connectors que permet ampliar les seves capacitats.

L'objecte d'aquest projecte és analitzar una part del funcionament de la placa perquè serveixi com a guia bàsica per a comprendre les possibilitats d'aplicacions del sistema. Es pretén que tot l'estudi realitzat pugui significar el punt de partida per desenvolupar futures aplicacions utilitzant la placa DE0-Nano. Per fer-ho s'interpretarà part dels programes de demostració que inclou la placa i es farà una introducció al llenguatge Verilog.

1.3 Especificacions i abast

En aquest projecte es pretén estudiar i realitzar la comunicació sèrie I2C, la SPI de tres senyal i la SPI de quatre senyals que es troben presents a la placa DE0_Nano. Per portar-ho a terme s'utilitzen tres programes de demostració proporcionats pel fabricant Altera Corporation. Aquest programes estan fets amb el llenguatge Verilog. Per aquest motiu es realitza una introducció d'aquest llenguatge i és presenten alguns exemples, els quals amb alguns d'ells es pot comprovar el seu correcte funcionament mitjançant la placa DE0-Nano. Tanmateix es realitzaran esquemes o bloc per poder interpretar més fàcilment els programes i s'expliquen dos modes de programació de la Cyclone IV, és a dir, dues maneres diferents per carregar un programa a la Cyclone IV.

Per portar a terme aquest projecte s'utilitzen els components de la placa DE0_Nano següents: la FPGA Cyclone IV d'Altera model EP4CE22F17C6N, l'acceleròmetre ADXL345, el convertidor analògic a digital 128S022, la memòria EEPROM 24LC02B, els vuit LEDs, els quatre interruptors, els dos polsadors i el connector JP3.

El programa DE0_NANO_EEPROM s'utilitza per estudiar i portar a terme la comunicació sèrie I2C entre la Cyclone IV i la memòria 24LC02B. El programa assigna l'adreça on es pretén escriure les dades i el seu valor. Per poder comprovar que realment s'ha fet aquesta operació s'utilitza l'aplicació DE0 Nano Control Panel proporcionada per la mateixa empresa Altera. Aquest programa es modifica per poder seleccionar el valor que es vol assignar a l'adreça utilitzant els 4 interruptors. La memòria disposa d'un mode de múltiple escriptura. Caldrà realitzar els canvis necessaris en el programa DE0_NANO_EEPROM per poder dur a terme aquesta comunicació.

El programa DE0_NANO s'utilitza per estudiar i portar a terme la comunicació sèrie SPI de quatre senyals entre la Cyclone IV i el convertidor analògic a digital. Per fer-ho s'utilitza els interruptors SW0, SW1 i SW2 per seleccionar l'entrada del convertidor que es vol llegir. En aquest projecte es fa servir l'entrada Analog_In0. El resultat de la conversió es representa

mitjançant els vuit LEDs de la placa. El voltatge que s'aplica a l'entrada Analog_In0 és escollit per l'usuari.

El programa DE0_NANO_G_Sensor s'utilitza per estudiar i portar a terme la comunicació sèrie SPI de tres senyals entre la Cyclone IV i l'acceleròmetre ADXL345. En funció de la inclinació de la placa respecta l'eix X s'encenen els LEDs corresponents. Per altra banda de la funcionalitat del programa també s'ha estudiat el funcionament respecte a l'eix Y i Z.

2 DE0-NANO D'ALTERA

Altera ens ofereix la placa DE0-Nano. Es tracta d'un dispositiu que l'usuari pot programar-lo mitjançant un software específic el qual permet fer una programació adient perquè els projectes no acumulin retards innecessaris i proporcionin la màxima flexibilitat a l'usuari, tot això s'aconsegueix entre altres coses fent passar totes les connexions a través de la FPGA tal com es pot veure a la Figura 1.

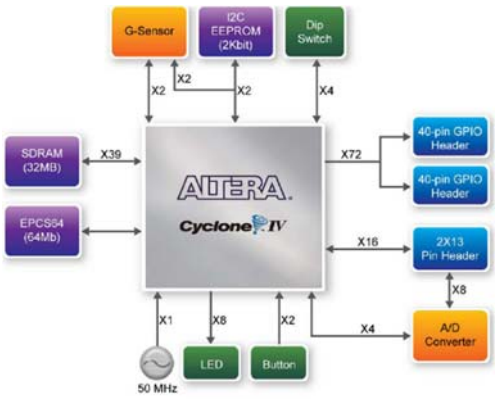


Figura 1. Diagrama de blocs de la placa DE0-Nano

El disseny de la placa DE0-Nano és compacte i disposa d'elements d'entrada i de sortida per a la comunicació i per a la programació. Tal com es pot veure a la Taula 1 alguns d'aquets components són els següents:

Component	Descripció
Cyclone IV EP4CE22F17C6N	FPGA
ADXL345	Acceleròmetre
128S022B	Convertidor A/D
IS42S16160G-7TLI	Memòria SDRAM de 32 MBytes
24AA02	Memòria EEPROM de 2 kBytes
S25FL064P	Memòria FLASH de 8 MBytes
FT245BL	USB-FIFO
Rellotge 50MHz	Rellotge
SN74AUC17	Buffer Trigger Schmitt
EMP240	Dispositiu de lògica programable complexa
8 LEDs verd	Elements visualitzadors de sortida
2 polsadors	Elements d'entrada
4 interruptors	Elements d'entrada
2 connectors de 40 pins	Connectors d'entrada i/o sortida
1 connector de 26 pins	Connectors d'entrada i/o sortida
1 connector de 2 pins	Connector per alimentació externa
Port USB mini A-B	Per la configuració, la programació i l'alimentació

Taula 1. Especificacions tècniques de la placa DE0-Nano

L'alimentació de la placa es realitza mitjançant el connector USB que proporciona un voltatge de 5 Vdc, quan aquest està connectat a un ordinador o utilitzant una font externa de 5 V. La placa també es pot alimentar utilitzant els dos pins del connector JP4 amb una tensió compresa entre 3,6 i 5,7 volts. Un LED amb el nom de POWER serveix per verificar que el sistema està convenientment connectat.

La placa disposa de tres connectors, dos dels quals són de 40 pins i l'altre de 26 pins. En el conjunt dels dos connectors de 40 pins hi ha un total de 72 pins d'entrada o de sortida de senyals que van connectats directament a la Cyclone IV, 4 pins de referència a massa, 2 pins de sortida de voltatge 3,3 V i finalment 2 de pins sortida de voltatge de 5 V. El connector de 26 pins disposa d'1 pin d'alimentació de 3,3 V, d'1 pin de referència a massa, de 8 pins que s'utilitzen per a les entrades analògiques del convertidor analògic a digital i de 16 pins per a senyals d'entrada o sortida connectats directament a la Cyclone IV.

La placa disposa d'un total de dos polsadors, assignats amb la nomenclatura KEY0 i KEY1. Aquest poden generar dos estats binaris, 0 i 1, depenent de la seva posició. La senyal que proporcionen els polsadors passa per un buffer Trigger Schmitt, per evitar rebots que podrien ocasionar lectures errònies, abans d'entrar a la FPGA. Els polsadors, quan no es pressionen, proporcionen un senyal de nivell lògic alt de 3,3 V i quan es premem proporcionen un senyal de nivell lògic baix de 0 V. En els programes que s'estudiaran més endavant el polsador KEY0 s'utilitzarà per reinicia els programes.

La placa també disposa de 4 interruptors, assignats amb la nomenclatura de SW0 a SW3. Aquest poden generar dos estat binaris, 0 i 1, depenent de si estan oberts o tancats. A diferència dels polsadors, els senyals que proporcionen van directament connectats a la FPGA. En el programa que utilitza el convertidor analògic a digital els interruptors serveixen per seleccionar l'entrada d'aquest convertidor. A la Figura 2 s'observa la distribució dels components a la placa.

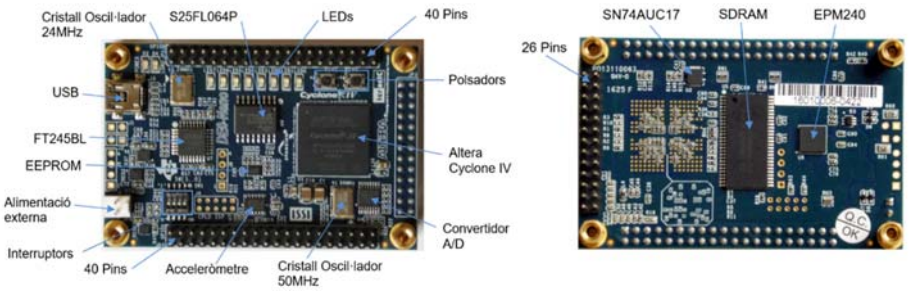


Figura 2. Components de la placa DE0-Nano

La placa utilitza una FPGA de la gama FPGA Cyclone IV d'Altera model EP4CE22F17C6N. La gamma de dispositius de la sèrie Cyclone IV segueix la mateixa arquitectura de la resta de les sèries de Cyclone, basada en files i columnes. A la Taula 2 es detalla algunes de les seves característiques tècniques.

Element	Descripció
Número d'elements lògics (LEs)	22.320
Bits totals de RAM	60.8256
Multiplicadors embedded (18 bits x 18 bits)	66
PLLs (Phase-Loked Loop)	4
Pins I/O de l'usuari	153
Tipus d'encapsulat	FBGA-256
Blocs de memòria M9K	66
Memòria embedded (kbits)	594

Taula 2. Característiques tècniques de la Cyclone IV model EP4CE22F17C6N

Per dissenyar una aplicació el fabricant Altera ens ofereix el programa Quartus II. Per programar la FPGA cal utilitzar l'entrada USB i un LED amb el nom LOAD que ens indicarà quan s'està enviant el programa. Una vegada aquest és transferit romandrà actiu sempre i quan no es desconnectí l'alimentació ni es torni a configurar la FPGA amb un altre programa. Per transferir el programa s'utilitza l'eina *Programmer* del Quartus II i el tipus d'arxiu que es transfereix és sof. Tal com es mostra a la Figura 3, aquest arxiu es transfereix directament a la Cyclone IV mitjançant el USB.

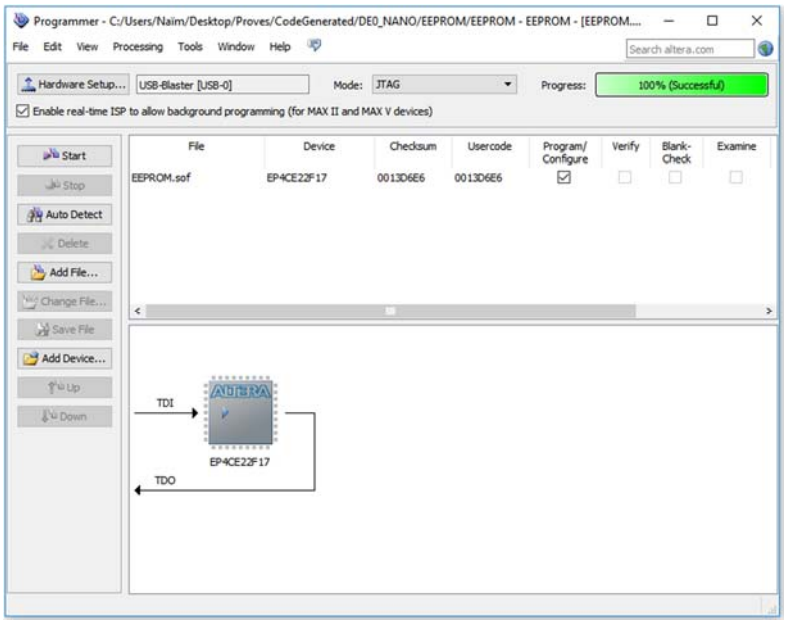


Figura 3. Eina *Programmer* per transferir un programa a la Cyclone IV

Una altra manera de programar la FPGA és guardar el programa al dispositiu S25FL064P, el qual té un emmagatzematge no volàtil, és a dir, que manté tota la informació encara que la placa no estigui alimentada. Aquest component també rep el nom de EPCS64. Quan es torna alimentar la placa, el programa guardat a S25FL064P es carrega automàticament a la Cyclone IV. Tal com es mostra a la Figura 4 aquest es transfereix directament a la Cyclone IV i seguidament a EPCS64 mitjançant el USB.

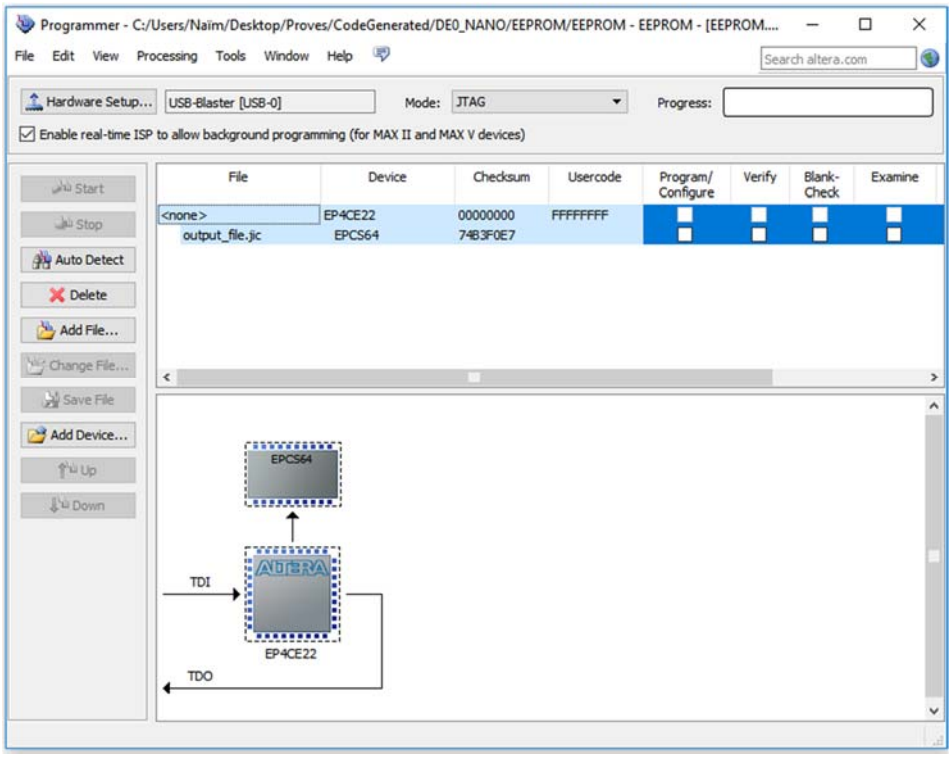


Figura 4. Eina *Programmer* per transferir un programa el EPCS64

Per transferir el programa també s'utilitza l'eina *Programmer* del Quartus II i el tipus d'arxiu que es transfereix és jic. Aquest arxiu jic no es crea automàticament sinó què, és el propi usuari que el crea fent ús de l'eina *Convert Programming File* del Quartus II. Tal com mostra la Figura 5 per crear l'arxiu cal definir el tipus de fitxer de programació, el dispositiu que es configura, el tipus de FPGA que s'utilitza i l'arxiu sof que es vol convertir.

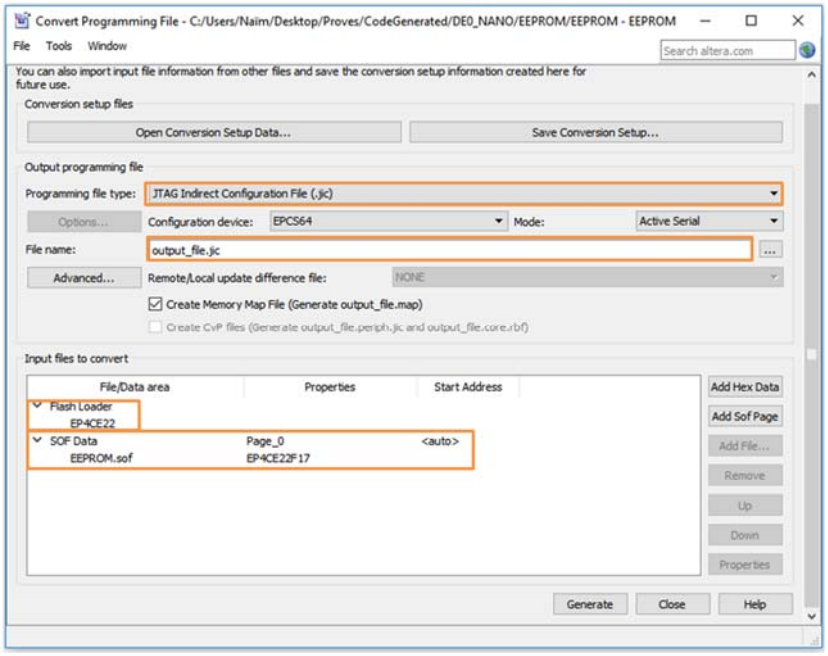


Figura 5. Eina *Convert Programming* File per converitr un fitxer sof a jic

Quan es crea un nou projecte amb el Quartus II hi ha una sèrie d'arxius que són necessaris per evitar que apareguin errors. Aquests són: l'arxiu del projecte del Quartus II (.qpf), l'arxiu de configuració del Quartus II (.qsf) on s'especifica els pins de la FPGA que van connectats com entrades o sortides, els arxius amb Verilog (.v) on es descriuen les entrades i sortides del disseny, l'arxiu de restriccions de temps de Synopsys (.sdc) i finalment l'arxiu d'assignació de pins (.htm).

3 INTRODUCCIÓ AL VERILOG

El Verilog és un llenguatge de programació especialitzat que s'utilitza per definir l'estructura, el disseny i les operacions de circuits electrònics, i més comunament de circuits electrònics digitals. Aquest llenguatge és un model de HDL semblant a el llenguatge C però es diferencia d'aquest perquè les execucions de les accions no són sempre lineals.

El Verilog ens permet dissenyar sistemes digitals molt complexos de manera molt pràctica i fàcil. Els dissenys poden ser simulats prèviament, per tal de verificar que es compleixen les especificacions lògiques i temporals requerides abans de ser transferits en el dispositiu programable.

El llenguatge permet l'ús de números reals i enters. Aquests es poden definir com a decimals, hexadecimals, octals o binaris. Si un número és negatiu, es representa amb el complement a2. Per escriure un número s'utilitza la sintaxis següent: en primer lloc s'especifica el numero de bits expressat amb decimal, en cas que no s'especifiqui el valor per defecte serà 32. A continuació es detalla la base amb que s'escriu el valor, "b" per binari, "d" per decimal, "h" per hexadecimal i "o" per octal, en cas de no especificar-ho el valor per defecte és en base decimal. I finalment s'especifica el número. Quan els números són molt llargs es pot utilitzar el caràcter "_" per una representació més clara.

Quan es defineix un número s'han de tenir en compte dos conceptes. Primer, que si la grandària del numero definit és menor que la del número que es vol expressar, es perden els bits més significatius. Segon, si la grandària del número definit és més gran que la del número que si vol expressar, els espais en vuit s'ompliran automàticament en funció del valor del bit més significatiu; si el valor és "0" o "1", els espais buits s'omplen amb "0" i si el valor és "z" o "x", s'omplen amb "z" i "x" respectivament. A la Taula 3 hi ha representats alguns exemples.

Número	Representació	Descripció
1	00000000000000000000000000000001	El número 1 amb decimal amb 32 bits
'o10	00000000000000000000000000001010	El número 10 amb octal amb 32 bits
8'b1	00000001	El número 1 amb binari amb 8 bits
6'b10_0011	100011	El número 35 amb binari amb 6 bits
8'hF	00001111	El número 15 amb hexadecimal amb 8 bits
8'b1x	0000001x	Valor 1x en binari en 8 bits
16'bz	zzzzzzzzzzzzzz	Valor z en binari amb 16 bits
-8'd2	11111110	Numero -2 amb decimal amb 8 bits

Taula 3. Exemples de definició de numero enters amb Verilog

La “x” representa un valor desconegut no es pot considerar ni 0 ni 1. Quan el valor és “z” és valor d’alta impedància com un buffer de tres estats. Això passa quan el senyal de control no està definit o bé quan una variable tipus *wire* no està connectat.

A la placa DE0-Nano el valor de “0” en binari és igual a 0 V i el valor de “1” en binari és igual a +3,3 V.

Un disseny en Verilog esta constituït per una jerarquia de mòduls. Quan es defineix un mòdul es defineix el seu nom i els noms dels respectius ports de les entrades i de les sorties d’aquest. Internament el mòdul conté una llista de variables i registres. Els diferents processos que hi han defineixen el comportament del mòdul, descrivint les relacions entre els ports i els registres. Les accions seqüencials normalment van col·locades dins d'un bloc “begin / end” i són executades en ordre seqüencial. Un mòdul pot contenir altres mòduls dintre seu i les accions d’un poden afectar a la resta.

Quan s’inicia un projecte cal decidir quin és el mètode més convenient per dissenyar un programa. Hi han tres mètodes i són les següents: pel sistema de portes lògiques, pel sistema de transferència de registres i pel sistema de comportament; considerant el primer com el més simple i l’últim el més complex.

3.1 Tipus de variables i operacions amb Verilog

El Verilog distingeix entre majúscules i minúscules per tant, és recomanable escriure sempre amb el mateix tipus de lletra. Les variables tenen que començar amb una lletra, un número o amb el símbol “_” o “$”. Les variables poden tenim un màxim de 1.024 caràcters.

En un mòdul es distingeixen dos tipus de variables, les externes i les internes. Les externes són les que actuaran com a senyals d’entrada o de sortida del mòdul i les internes, anomenades nodes, serveixen per fer operacions amb les dades que entren dins el mòdul. Les senyals d’entrada no es defineixen i per defecte son de tipus *wire*, en canvi, les senyals de sortida i els nodes poden ser de tipus *wire* o *reg*. També es pot donar el cas que algun d’aquests senyals sigui bidireccional.

Per definir una variable s’ha de seguir una sintaxis concreta. Primer es determina el tipus de variable, seguidament s’especifica la mida, és a dir, el nombre de bits que té, el qual

s'expressa dins un parèntesis quadrat, que pot ser un número o una variable i finalment es posa el nom de la variable.

Les variables més utilitzades són les variables *reg* que tenen la capacitat de guardar informació i les *wire* que estableixen una connexió entre dos components i no tenen la capacitat de guardar les dades. Altres tipus de variables són: les *integer*, que són registres de 32 bits, les *real*, que són capaces de guardar números amb coma flotant i les *time*, que són registres sense signe de 64 bits.

La Figura 6, mostra un exemple de definició de diferents variables. En aquest cas hi ha una variable d'entrada amb el nom "entrada", una de sortida amb el nom "sortida" de tipus *wire* de 8 bits, una de tipus entrada-sortida amb en nom "entrada_sortida" d'1 bit, un node tipus *wire* amb el nom de "a" d'1 bit i un altre amb el nom "b" tipus *reg* de 4 bits.

```
input                   entrada;
output reg [7:0]        sortida;
inout                   entrada_sortida;
wire                    a;
reg        [3:0]        b;
```

Figura 6. Diferents tipus de variables

Una vegada estan definides les variables es pot determinar l'estructura del programa. Per portar-ho a terme s'utilitza diferents operadors per construir les expressions, que poden ser: aritmètiques, racionals, d'igualtat, lògiques, bit a bit, de reducció, de desplaçament, de concatenació i condicional.

Els operadors aritmètics són: la suma "+", la resta "-", la multiplicació "*", la divisió "/", el residu "%" i l'exponencial "**". En tots els casos si algun dels bit val "x" el resultat serà "x".

Els operadors racionals permeten comparar dos operants, retornant 1 o 0, segons sigui verdader o fals respectivament. En cas que algun dels bits dels operants sigui "x" el resultat serà "x". Els tipus d'operadors racionals són: "<" més petit que, "<=" més petit o igual que, ">" més gran que, ">=" més gran o igual que.

Els operadors de igualtat s'utilitzen com a condicionals. Aquest permeten comparar dos operants bit a bit. Si un dels dos és més gran que l'altre en el més petit s'ha afegeixen 0 automàticament per igualar el nombre de bits. L'operador "==" retorna 1 si els dos operants són igual i 0 en cas contrari. L'operador "!=" retorna 1 quan els dos operants són diferents i 0

en cas contrari. Els operadors “==” i “!=” només comparen 1 i 0. Si algun bit és “x” o “z” el resultat serà “x”. En cas que calgui comparar les “x” i “z” i el resultat no es vol que sigui “x” aleshores s’ha de utilitzar “===” i “!==”.

Els operadors lògics són tres, la negació lògica “!”, la AND lògica “&&” i la OR lògica “||”. La negació lògica canvia el valor lògic de operant que hi ha el seu darrera. La AND i la OR lògica s’avaluen d’esquerra a dreta. Perquè el resultat de la AND sigui 1 els dos valors tenen que ser 1, en cas contrari serà 0 i en el cas de la OR només fa falta que un dels dos sigui 1.

La lògica bit a bit permet fer operacions lògiques amb tots els bits dels dos operants. Les diferents operacions que es poden fer són: negació bit a bit “~”, AND bit a bit “&”, OR bit a bit “|”, XOR bit a bit “^”, NAND bit a bit “~&”, NOR bit a bit “~|” i NOT XOR bit a bit “~^ “ o “^~”.

Les operacions lògiques de reducció fan operacions amb els bits de l’operant, és a dir, si un operant és de vuit bits la operació es farà amb aquest vuit i donarà com a resultat un únic bit. Les diferents operacions lògiques que hi han són: AND “&”, OR “|”, XOR “^”, NAND “~&”, NOR “~|” i NOT XOR “~^”.

Els operadors de desplaçament són dos el que desplaça a l’esquerra “<<” i desplaça a la dreta “>>”. Els bits desplaçats s’omplen amb “0”.

La concatenació serveix per ajuntar dos operants, sempre que ambdós estiguin definits. Els operants se separen amb comes i es troben dins claudàtors “{}”.

El condicional serveix per designar un valor depenent de si val 1 o 0.

L’ordre de prioritat dels operadors en una equació és: en primer lloc la multiplicació, la divisió i el mòdul, en segon lloc la suma, la resta i els desplaçaments, en tercer lloc els de igualtat i els racionals, en quart lloc els de reducció, en cinquè lloc els lògics i finalment en sisè lloc el condicional.

3.2 Processos

Una de les característiques fonamentals del llenguatge Verilog és que els processos sempre s'executen en paral·lel, a diferència d’altres llenguatges similars, com el llenguatge C.

Tota descripció de comportament en llenguatge Verilog ha de ser definit dins d'un procés excepte quan s'utilitza l'assignació continua que s'explicarà el final d'aquest apartat. En el llenguatge Verilog existeixen dos tipus de processos, el *initial* i *always*. Quan un procés engloba més d'una assignació o més d'una estructura de control aquests tenen que estar delimitats per *begin* i *end*.

El procés initial s'executa al començament del programa i només una vegada. Aquest procés no és sintetitzable. L'always s'executa contínuament en forma de bucle cada cert temps i es controla per temporitzacions o per esdeveniments.

Quan es controla per temporitzacions es crea un retard que s'especifica mitjançant el símbol "#" i a continuació el temps de retard. Com a exemple es presenta el següent procés:

```
initial
begin
      clk = 0;      // Primera assignació
      reset = 0;    // Segona assignació
      #5 reset = 1; // Tercera assignació
      #4 clk = 1;   // Quarta assignació
      reset = 0;    // Cinquena assignació
end
```

Les dues primeres assignacions s'executen a l'instant de temps zero, després de cinc unitats de temps més tard es realitza la tercera assignació i finalment quatre unitats de temps més tard s'executa la quarta i cinquena assignació. El resultat es pot veure a la Figura 7.

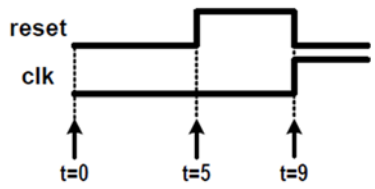


Figura 7. Representació gràfica del procés *initial*

L'assignació del procés *always* és per esdeveniment que es produeix pel canvi d'una variable. Per definir-ho s'utilitza el caràcter "@" i a continuació l'esdeveniment. Hi ha dos tipus d'esdeveniments, el de nivell, que es produeix quan hi ha un canvi de valor d'una o més variables i el de flancs, que es produeix quan una variable provoca un flanc de pujada o de baixada. A la Taula 4 hi ha alguns exemples dels tipus d'esdeveniments.

Tipus	Esdeveniment	Descripció
Nivell	always @ (a) b <= b+c;	Cada vegada varia el valor de (a) s'avalua l'expressió
Nivell	always@(a or b or c) b <= b+c;	Cada vegada varia el valor de (a), (b) o (c) s'avalua l'expressió
Flanc	always@(posedge clk or posedge jr) b <= b+c;	Cada vegada que es produeix un flanc de pujada de (clk) o de (jr) s'avalua l'expressió
Flanc	always@(posedge clk or negedge jr) b <= b+c:	Cada vegada que es produeix un flanc de pujada de (clk) o de baixada de (jr) s'avalua l'expressió

Taula 4. Exemples d'esdeveniments amb Verilog

La assignació continua s'utilitza amb la lògica combinacional. Totes les assignacions continues s'executen a la vegada. També es pot controlar el temps en que s'executa l'assignació i sempre ha d'estar fora d'un procés *initial* o *always*. A continuació es mostra un exemple corresponent a un buffer de tres estats.

```
assign #1 out = (enable) ? data : 8'bz;
```

A la variable *out* se li assigna el valor de data o d'alta impedància 8'bz depenen del valor de la variable *enable* cada cop que transcorre una unitat de temps. Si *enable* val 1, *out* té el valor de *data* i en cas contrari el valor és 8'bz.

3.3 Tipus d'estructures

Existeixen diferents estructures en el llenguatge Verilog, algunes d'aquestes són: la *conditional*, la *case*, la *casez*, la *casex*, la *forever*, la *repeat*, la *while* i la *for*. Totes elles van dins un procés *always* o *initial*. A continuació s-explicant les estructures *conditional* i *case* que són les que s'utilitzen en els programes de la DE0-Nano estudiats.

L'estructura *conditional* es defineix amb *if* al principi i s'utilitza *else* quan hi ha més d'una condició. Hi ha tres tipus d'estructura: la condicional simple, la condicional doble i la condicional múltiple. A continuació es mostra un exemple de cada tipus.

```
// Condició simple
if (a);  // Si a val 1, b val 1
      b=1;
//Condició doble
if (a); // Si a val 1, b val 0
      b=0;
else    // Si a no val 1, b val 0
      b=1;
//Condició múltiple
```

```
if (reset == 1'b1) // Si reset val 1, hi ha un retard de 1 unitat de temps
      count<= #1 8'b0; // el comptador de vuit bits val 0
else if (up == 1'b1) //Si reset val 0 i up val 1 se suma 1 al comptador
      count<= count + 1'b1;
else if (down == 1'b1) //Si reset i up val 0 i down val 1 es resta 1 al
comptador
      count<= count – 1'b1;
else                    //Si reset, up i down val 0 al valor del comptador
és igual.
      count<= count;
```

La condició simple sempre té una única acció. Quan la variable "a" tingui un valor 1 la variable "b" també té un valor 1. En la condició doble hi han dues accions possibles. Si el valor de "a" val 1 el de "b" és 0 i si "a" val 0 el de "b" és 1. Finalment el cas de condició múltiple poden passar quatre accions. Si el valor de *reset* és 1 el comptador, de vuit bits, val 0; si el valor de *reset* és 0 i el de *up* val 1 el valor del comptador s'incrementa amb 1 unitat; en cas que el valor de *reset* i de *up* sigui 0 i el de *down* sigui 1, el valor del comptador disminueix amb 1 unitat i finalment si el valor de *reset*, *up* i *down* val 0 el valor del comptador continua amb el mateix valor.

L'estructura *case* avalua una expressió i en funció del seu valor executa unes accions, anomenades casos. És possible que el valor que pot tenir l'expressió sigui més gran que el nombre d'accions que es defineixen en l'estructura, en aquest cas és necessari utilitzar el *defaut*. Aquesta acció s'executarà sempre que no es complexi cap dels casos anteriors. A continuació es mostra un exemple d'estructura *case*.

```
case (sel) // sel és l'expressió que s'avalua
      0       : a= 4'b0101;
      1       : a= 4'b0011;
      2       : a= 4'b1111;
      defaut  : a= 4'b0000;
endcase
```

L'expressió que s'avalua és la variable "sel" i depenen del valor que aquesta tingui s'assigna un valor a la variable "a" de quatre bits. Si "sel" val 0, "a" val 0110, si "sel" val 1 "a" val 0011, si "sel" val 2 "a" val 1111 i finalment si "sel" val qualsevol altre valor dels comentats "a" val 0000.

3.4 Disseny pel sistema de portes

El disseny pel sistema de portes permet representar estructures a baix nivell. Per fer el disseny s'utilitzen les funcions lògiques bàsiques per representar les portes lògiques, les connexions lògiques i els paràmetres de temps. Les senyals només podem tenir el valor de "0", "1" , "x" o "z".

Els noms en minúscula de les portes lògiques són paraules reservades per representar aquestes, alguns exemples són and, or, nor, buf i not. Per escriure una porta AND de dues entrades "a" i "b", una sortida "c" i amb el nom "porta2" la sintaxis és la següent:

```
and porta2 (c, a, b);
```

El nom és opcional i si es vol s'hi pot afegir un temps de retard posant el símbol "#" darrera de "and" i a continuació el temps desitjat. La Figura 8 representa la AND descrita anteriorment.

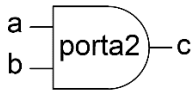


Figura 8. Porta AND

3.4.1 Multiplexor amb portes lògiques

El disseny en portes lògiques és de nivell baix. Aquest mètode de disseny no és apropiat per fer programes de gran envergadura. A continuació es presenta el programa *multiplexor* com exemple d'un multiplexor de dues entrades amb el codi Verilog. Transferint el programa a la placa DE0-Nano es pot comprovar el seu correcte funcionament.

```
module multiplexor(
output            [7:0] LED,
input             [3:0] SW
);
//Estructura del programa
and #5 porta0   (res1,SW[0],nsel);
and #5 porta1   (res2,SW[1],SW[2]);
or  #5 porta2   (LED, res2, res1);
not              (nsel,SW[2]);
endmodule
```

L'entrada SW correspon als quatre interruptors. El SW0 i el SW1 representen les entrades del multiplexor. El SW2 s'utilitza per seleccionar l'entrada que es vol fer servir. A la sortida LED se li assigna el resultat de SW0 o de SW1. La variable LED és de 8 bits i el valor que se li assigna és d'1 bit, per aquest motiu només s'utilitza el LED0. Si SW0 o SW1 estan a *ON* el resultat és 0 i el LED0 està apagat, en cas contrari s'il·luminarà.

Les portes lògiques "and" amb el nom "porta0" i "porta1" i la "or" amb el nom "porta2" tenen un retard de 5 ms pensat perquè la porta "not" tingui temps de negar l'entrada SW2. La porta lògica "not" no té cap nom assignat, ja que no és necessari definir un nom per les portes. La Figura 9, és l'esquema del programa multiplexor.

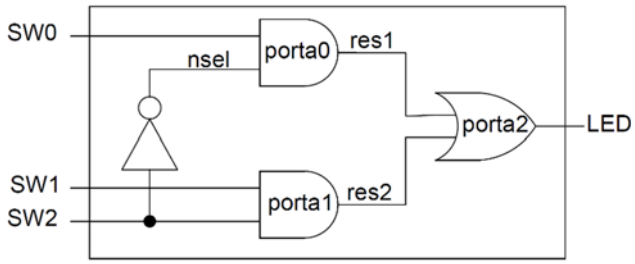


Figura 9. Esquema del programa multiplexor

3.5 Disseny pel sistema de transferència de registres

Quan el disseny és més complex i existeixen moltes operacions, treballar amb portes lògiques resulta molt complicat. La solució passa per dissenyar el programa amb el sistema de transferència de registres. Aquest sistema és anomenat també RTL.

El disseny amb RTL permet implementar lògica combinacional. Utilitza la assignació contínua per crear les relacions lògiques existents entre les entrades i les sortides. Aquesta disseny és sempre sintetitzables, és a dir, els programes de síntesi permeten crear els esquemàtics equivalent.

Els dissenys amb RTL s'ha de descriure amb precisió. Per fer-ho s'utilitzen operacions i transferència de dades entre els registres. Es fan servir especificacions de temps perquè les operacions es realitzin en uns instants determinats. Aquest sistema permet la simulació en determinades parts del disseny que requereixen un comportament molt específic.

Com exemple d'aquest sistema es descriu un filp-flop tipus D, sense *reset* i *preset* amb dues sortides. El codi amb Verilog és el que es detalla a continuació.

```
module flip_flop (entrada, rellotge, sortida, nsortida); //Definició del
mòdul
//Definició de les variables
input             entrada, rellotge;
output            sortida;
reg               q;
//Definició del procés
always @(posedge rellotge)// Sempre que hi hagi un flanc de pujada del
rellotge
begin
sortida     <= entrada;  //El valor de l'entrada s'assigna a la sortida
nsortida    <= !entrada; //El valor negat de l'entrada s'assigna a la
sortida.
end
endmodule
```

3.5.1 Comptador

Un comptador és un circuit seqüencial que s'utilitza en moltes aplicacions de sistemes digitals. En tots els programes que s'estudiaran més endavant en aquest projecte s'utilitza un comptador i per aquest motiu és convenient mostrar un exemple d'un comptador bàsic que ens ajudarà entendre el seu funcionament. Amb el Quartus II es crea el programa comptador i utilitzant la placa DE0-Nano es pot visualitzar el seu correcte funcionament. A la Figura 10 s'observa l'esquema del comptador.

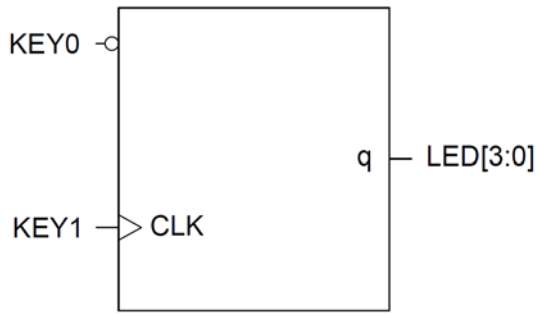


Figura 10. Representació gràfica del programa comptador

El codi descriu un comptador de 4 bits ascendent, amb una entrada de *reset*, activa a nivell baix per reiniciar-lo a 0, una entrada pel senyal del rellotge i una sortida pel valor. Cada vegada que arribi un flanc de pujada del senyal de rellotge, el valor del comptador augmenta en una unitat. Com que la variable sortida és de quatre bits un cop arribi a setze ja no puja més i

romandrà en aquest valor fins que es reinicií. El polsador KEY1 s'utilitza per generar el senyal de rellotge i el polsador KEY0 s'utilitza per reiniciar el comptador. De la sortida LED només s'utilitzen 4 bits corresponents als LEDs 0,1,2 i 3. Aquest representaran el valor del comptador amb codi binari. La variable interna "q" serveix per assignar a la sortida LED el valor de comptador. Per fer-ho s'utilitza la funció *assign*.

```
module comptador(
      output      [7:0] LED,
      input       [1:0] KEY
                 );
// Variables internes
reg [3:0]     q;
//Definició del procés
assign  LED[3:0] = q;
always @(posedge KEY[1] or negedge KEY[0])
if (!KEY[0])
q <= 8'b00000000;
else if (q<4'b1111)
q<=q+1;
else
q<=q;
endmodule
```

En aquest codi quan es defineixen les variables externes del mòdul també se'ls hi assigna si són entrada o sortida. La variable interna "q" és tipus *reg* perquè ha de recordar el valor del comptador.

3.5.2 Registre de desplaçament

Els registres de desplaçament són circuits seqüencials que permeten guardar un valor i desplaçar-lo. Són molt utilitzats en les comunicacions síncrones com la SPI. L'entrada de les dades normalment és en sèrie i la sortida pot ser en paral·lel o en sèrie.

Com exemple d'un registre de desplaçament s'ha utilitzat el programa Quartus II per crear el programa registre_d i la placa DE0-Nano per comprovar el seu correcta funcionament. El que es pretén amb aquest programa és enviar la dada amb el nom "a" formada per 3 bits a la sortida LED formada per 8 bits i veure el desplaçament de la dada dins el registre de desplaçament. Per veure el desplaçament és necessari que la sortida de les dades siguin en paral·lel i per simular el valor de les dades s'utilitzen els 8 LEDs de la placa.

El valor de la dada que es vol carregar se selecciona utilitzant els interruptors SW0, SW1 i SW2. El SW3 s'utilitza per habilitar el registre de desplaçament. Amb el polsador KEY0 es reinicia el registre de desplaçament posant tots els seus bits a 0. Com que el senyal de rellotge que porta incorporat la placa va molt ràpid s'utilitza el polsador KEY1 perquè en cada flanc de pujada executa els dos processos *always*. A la Figura 11 podem veure el registre de desplaçament de 8 bits.

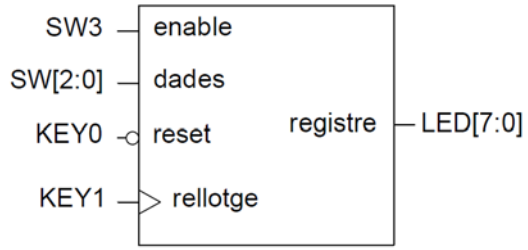


Figura 11. Representació gràfica del programa registre_d

El primer bit de la dada ocupa el bit de menys pes del registre de desplaçament i cada vegada que arriba un flanc de rellotge es desplaça cap a un bit de més pes. L'entrada "enable" ha de estar habilitada perquè funcioni el bloc, en cas contrari, el valor dels bits no canvia. El codi del registre de desplaçament descrit és el següent.

```
//Definició del mòdul de les variables externes
module registre_d(
      output      [7:0] LED,
      input       [1:0] KEY,
      input       [3:0] SW
);
// Definició de la variables internes
reg [7:0] registre;
wire      enable,a;
reg [3:0] count;
// Definició del procés
assign LED [7:0] = registre;
assign enable = SW[3];
assign a= enable&(count<3) ? SW[count]:1'b0;
always @(posedge KEY[1] or negedge KEY[0])
begin
if (!KEY[0])
registre <=0;
else
if (enable)
registre <={registre[6:0], a} ;
end
always @(posedge KEY[1] or negedge KEY[0])
begin
```

```
      if (!KEY[0])
      count =0;
      else if (count < 10)
      count <= count+1;
   else if (!enable & KEY0)
      count =0;
   else
      count =0;
 end
endmodule
```

A la sortida LED se li assigna el valor de la variable interna “registre”, a la variable interna “a” se li assigna el valor dels interruptors SW0, SW1 i SW2 i finalment a la variable “enable” se li assigna el valor de SW3.

L’assignació del valor a la variable “registre” i a la “count” s’ha optat per fer-ho amb dos processos *always* perquè quedin clarament diferenciats.

En el funcionament del procés always per la variable “registre” s’assigna per concatenació el seu valor, on els bits de menys pes es conserven i el de més pes s’elimina per deixar lloc a un nou valor. Això passa sempre que les entrades “enable” i “reset” estiguin a nivell alt.

En el funcionament del procés *always* s’augmenta el valor de la variable “count” en cada flanc de pujada de l’entrada de rellotge mentre aquesta variable sigui inferior a 10. S’ha optat posar aquesta limitació perquè no torni entrar el valor de la variable “a” que conté el valor dels interruptors SW0, SW1 i SW2 i alhora poder veure clarament el desplaçament dels bits sense confusions. El comptador torna al seu valor inicial sempre que es polsi el polsador KEY0 o el seu valor sigui més gran de 10 o que es polsi el polsador KEY0 i a més a més el valor de “enable” sigui 0.

3.6 Disseny pel sistema de comportament

La principal característica d’aquest sistema és la llibertat de disseny de l’estructura que s’utilitza mitjançant algoritmes en paral·lel. Cada algoritme executa un conjunt d’operacions de forma seqüencial. Aquest algoritmes poden utilitzar accions o estructures no sintetitzables. Un exemple de disseny pel sistema de comportament és el programa DE0_NANO_G_Sensor explicat més endavant.

4 COMUNICACIONS

Actualment l'intercanvi de informació digital entre dos o més components electrònics es pot realitzar de dues maneres, en paral·lel o en sèrie. Tant en un sistema com en l'altre, si la informació que s'ha d'enviar és diferent al codi binari, com per exemple el codi ASCII, aquesta es codifica caràcter a caràcter amb codi binari i s'envia.

En la comunicació en paral·lel es transmet la informació simultàniament utilitzant línies separades, és a dir, hi han tantes línies com bits que es volen enviar, veure Figura 12. El processador es veu obligat a utilitzar un gran nombre d'entrades amb una velocitat de transmissió de les dades molt ràpida.

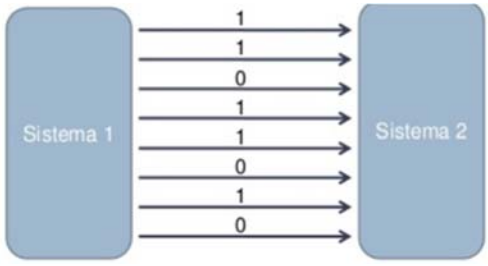


Figura 12. Representació de la comunicació paral·lel

La comunicació en sèrie és tot al contrari de la comunicació amb paral·lel atès que la informació és enviada bit a bit, veure Figura 13. Aquest sistema comporta que el receptor ha de reconstruir la informació que li arriba a sistema binari o qualsevol altra. En enviar les dades bit a bit la velocitat de transmissió d'aquestes és menor que la de comunicació en paral·lel.

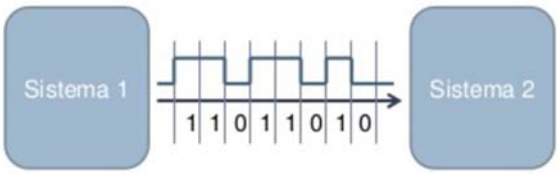


Figura 13. Representació de la comunicació en sèrie

Actualment la comunicació en sèrie és una de les més utilitzades per la majoria dels fabricants de components electrònics. Alguns exemples on s'utilitza aquest tipus de comunicació són: sensors de temperatura i d'humitat, pantalles, convertidors analògics a digitals, impressores, unitats d'emmagatzematge de informació, etc.

A la placa DE0-Nano quan s'utilitzen els connectors de pins es poden realitzar connexions en sèrie i en paral·lel amb l'ajuda de components electrònics addicionals i la programació adient de la Cyclone IV. Atès que aquest projecte es basa en l'estudi de la placa s'analitzen algunes de les comunicacions en sèrie que hi han entre els components de la placa.

4.1 Comunicacions sèrie

En un sistema de comunicació generalment les informacions es transmeten en blocs de caràcters. Per portar a terme una comunicació és necessari l'existència d'un emissor i d'un receptor que cal que aquests estiguin convenient sincronitzats. Entenem per sincronització de missatge quan hi ha una igualtat entre la fase i la freqüència de dos moviments periòdics d'un conjunt de paraules les quals seran interpretades correctament.

Existeixen diferents tipus de comunicació en sèrie, alguns exemples són: SCI, SPI, I2C, CAN, Profi*bus* i UART. La informació es transmet bit a bit per tant el sistema de codificació ha de distingir, sense problemes, quins bits formen una paraula. Normalment s'envien els bits de cada caràcter separats per una senyal de sincronisme. Per tenir aquest sincronisme es poden utilitzar diferents sistemes, com podria ser la utilització de línies addicionals a les dades per enviar els impulsos que indiquen el principi i el final del bloc de caràcters. Aquests impulsos identificaran el primer i l'últim bit de caràcters d'un bloc o missatge.

Els dos tipus de sistemes utilitzats en la comunicació en sèrie, són: la asíncrona i la síncrona.

La comunicació asíncrona és quan tots els dispositius tenen una base de temps pròpia que ha d'estar sincronitzats. La sincronització es realitza per programació, per maquinària o totes dues a la vegada i es porta a terme a través d'uns bits especials que defineixen l'entorn de cada codi. El caràcter que es vol enviar porta dos bits, un al principi, anomenat inici i un altre al final anomenat stop. Un exemple de comunicació asíncrona és la RS232.

La comunicació síncrona ha de tenir un senyal de rellotge que sincronitzi els diversos dispositius. Aquest senyal la genera el dispositiu mestre i l'envia a tots els dispositius esclaus. Aquest tipus de comunicació és molt utilitzat en memòries, acceleròmetres i DSP entre d'altres.

A diferència de la comunicació asíncrona, quan el receptor identifica una configuració apropiada per el sincronisme, comença la transferència de dades i a continuació es compten els bits per identificar els diferents blocs.

En aquest projecte s'explica les comunicacions en sèrie, que s'utilitzen per comunicar la Cyclone IV amb l'acceleròmetre, el convertidor analògic digital i la memòria EEPROM. Aquestes són les comunicacions síncrones SPI i I2C.

4.1.1 Comunicació SPI

La comunicació SPI és estàndard creat per Motorola a principis del 1980. Al llarg dels anys SPI s'ha convertit en un dels protocols més populars per treballar en comunicació en sèrie, gràcies a la velocitat de transmissió de dades, la simplicitat de funcionament i que una gran varietat de sensors, chips, mòduls de maquinària que es comuniquen mitjançant aquest protocol. Per això és molt important entendre com funciona i com programar-lo per poder crear infinitat de projectes.

És un protocol síncron que treballa en mode *full dúplex*. Aquest mode és capaç de mantenir una comunicació bidireccional, enviant i rebent dades de forma simultània. Per fer-ho cal que el mestre i l'esclau enviïn les dades en un flanc de rellotge i en el següent flanc les rebin. Com que les dades són enviades bit a bit, es guarden en un registre de desplaçament i quan està ple el mestre i l'esclau interpretaran les dades rebudes. La Figura 14, mostra el funcionament *full dúplex* de la comunicació SPI.

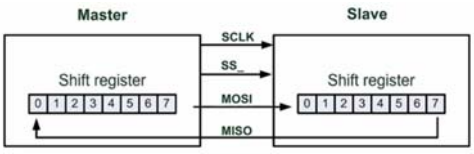


Figura 14. Funcionament de SPI

Aquest protocol defineix un mestre que és l'encarregat de transmetre informació a un o més esclaus que rebran i enviaran informació al mestre. La sincronització i la transmissió de dades entre l'emissor i el receptor es realitza utilitzant quatre senyals. El senyal SCLK és la del rellotge, el senyal MOSI és la que al mestre utilitza per enviar dades a l'esclau, el senyal MISO l'utilitza l'esclau per poder enviar dades al mestre i finalment, el senyal SS serveix perquè el

mestre pugui habilitat l'esclau amb el qual vol realitzar una comunicació, la Figura 15 mostra la connexió entre mestre i esclau.

Figura 15. Comunicació entre mestre i un esclau

El protocol SPI es comunica a través d'un *bus* o canal, que permet connectar més d'un esclau. Existeixen dues maneres de connectar múltiples esclaus a un *bus*, una és la configuració d'esclaus independents, en qué es necessiten sortides digitals addicionals controlades pel mestre per seleccionar l'esclau que es connecta al *bus*, tal com s'observa a la Figura 16, i l'altre, és la connexió en cascada on el mestre genera una senyal que habilita tots els esclaus alhora i les sortides MISO d'aquests es connecten a l'entrada MOSI del següent esclau, d'aquesta manera el mestre envia les dades només al primer esclau i rep resposta de l'últim esclau. Veure la línia vermella de la Figura 17.

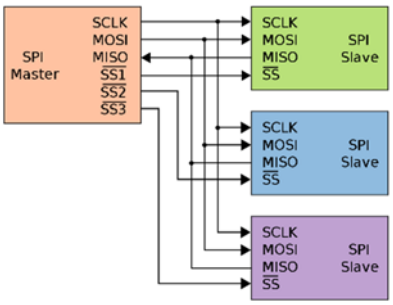


Figura 16. Comunicació SPI de quatre senyals amb un mestre i tres esclaus

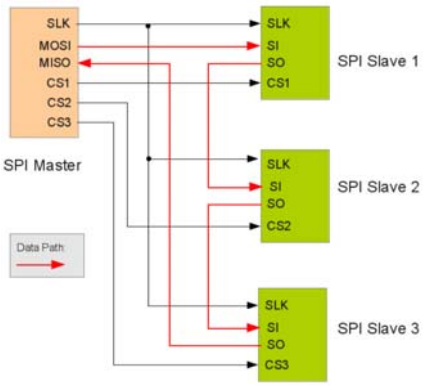


Figura 17. Connexió en cascada SPI de quatre senyals

La transferència del les dades és sincronitzada mitjançant la línia de rellotge SCKL. Un bit es transfereix a cada cicle de rellotge. El senyal SCKL la genera el mestre i és la mateixa per a tots els esclaus que hi estiguin connectats. Per enviar la informació existeixen diferents mètodes, en funció dels dos paràmetres relacionats amb el senyal de rellotge. El primer és la polaritat del rellotge anomenada CPOL, i és la que determina l'estat de la línia de rellotge està que pot ser en estat baix o estat alt. El segon és la fase anomenada CPHA, aquest determina en quin moment s'envien els bits. Per exemple, si CPHA val 0 les dades de la línia MOSI són detectades en el flanc de baixada i els de la línia MISO per el flanc de pujada.

Cada paràmetre té dos estats, per tant hi ha quatre combinacions possibles i són compatibles per realitzar la transmissió dels bits. La Taula 5 recull les diferents combinacions.

CPOL	CPHA	Mode
0	0	0
0	1	1
1	0	2
1	1	3

Taula 5. Configuració transmissió de les dades

Si el CPOL i el CPHA tenen valor zero, es defineix com a mode 0, inicialment el senyal de rellotge es troba a nivell baix i les dades s'envien en cada flanc de pujada del senyal de rellotge. Quan el CPOL val 1 i el CPHA val 0, es defineix com a mode 2, inicialment el senyal de rellotge està a nivell alt i les dades s'envien en el flanc de baixada. En el cas que el CPOL val 0 i el CPHA val 1, es defineix com a mode 1, inicialment el senyal de rellotge està a nivell baix i les dades s'envien en el flanc de baixada. I finalment, si el CPOL i el CPHA valen 1, es defineix com a mode 3, inicialment el senyal de rellotge està a nivell alt i les dades s'envien en el flanc de pujada. A la Figura 18 s'observa les diferents combinacions.

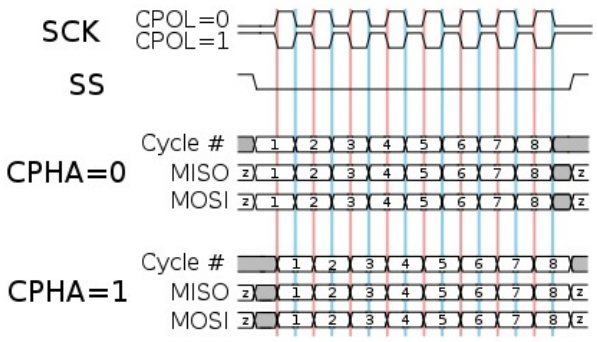


Figura 18. Temporitzacions del *bus* SPI

La configuració del CPHA i del CPOL de l'esclau ve donada pel fabricant així com també, l'orde en que es reben els bits, que pot ser que el primer bit que s'envia sigui el MSB o bé el LSB. Es pot donar el cas que es tingui més d'un esclau connectat a un mestre i que cada un tingui una configuració diferent. Això implica que el mestre té que adaptar-se a la configuració de cada esclau cada vegada que es comuniqui amb aquests. També pot ser que un dispositiu pugui admetre dos modes de configuració dels quatre possibles que hi han.

Les senyals MOSI i MISO s'utilitzen per enviar i rebre dades entre mestre i esclau. El mestre utilitza el MOSI per enviar les dades i l'esclau utilitza el MISO. En aquest tipus de comunicació només es pot tenir un mestre que s'encarrega d'iniciar la comunicació amb els diferents esclaus. Quan es comença una transmissió de informació entre mestre i esclau, no es fan servir adreces sinó que se selecciona l'esclau utilitzant el senyal SS, posant-lo a nivell baix. A la Figura 19 es pot veure una comunicació SPI en mode 0.

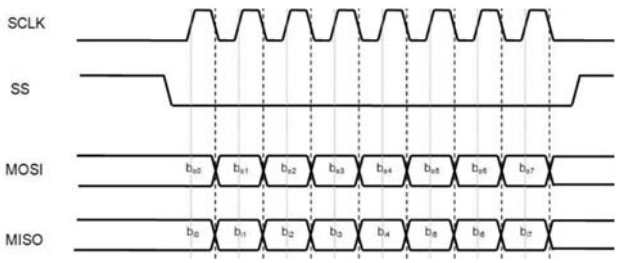


Figura 19. Comunicació SPI en mode 0

Una variant de la comunicació SPI és la connexió amb tres senyals en lloc de quatre. Les senyals MISO i MOSI utilitzen el mateix canal per a transmetre les dades; aquest mode de treball s'anomena *semi dúplex*. La transmissió *semi dúplex* permet transferir dades en els dos sentits, a diferencia de la *full dúplex* que nomes pot fer-ho en una direcció. De la mateixa manera que el mode *full dúplex* també hi ha quatre combinacions de transmissió de dades. A la Figura 20 es pot observar la connexió entre mestre i esclau en una comunicació SPI de tres senyals.

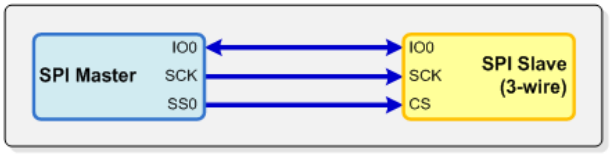


Figura 20. Connexió SPI de tres senyals

Els avantatges de la comunicació SPI en relació a altres protocols de comunicació síncrona és que és un protocol flexible atès que es pot tenir un control sobre els bits transmesos, com exemples, no està limitada la transferència de blocs de 8 bits, la implementació és molt simple i en comparació a la comunicació en paral·lel es necessiten molts menys senyals. A la placa DE0-Nano trobem la comunicació SPI de quatre i tres senyals.

4.1.2 Comunicació I2C

La comunicació I2C va ser desenvolupada per l'empresa Philips a la dècada dels anys 80. Avui en dia és un protocol tan popular com la comunicació SPI, i és utilitzat per molts fabricants de renom, com per exemple, les empreses Texas Instruments, Atmel i Analog Devices. L'I2C permet crear un *bus* de comunicació entre diferents dispositius en sèrie. És un protocol síncron, que només utilitza dues senyals per realitzar la comunicació. Un senyal és la de rellotge (SCL) i l'altre per a les dades (SDA). Això comporta que entre el mestre i l'esclau s'enviïn dades per el mateix canal i el mestre creï el senyal de rellotge. El I2C utilitza adreçament per seleccionar els diferents esclaus. Tots els dispositius porten una adreça fixada pel mateix fabricant i formada per 7 bits. Per aquest motiu es poden connectar 128 dispositius, cadascun d'ells amb una adreça diferent, tot i què, també existeix una versió de I2C al mercat que permeten el adreçament amb 10 bits malgrat no s'utilitza molt.

Per la gran diversitat de components que porten incorporats aquest protocol és molt possible que en un *bus*, dos o més components tinguin la mateixa adreça. Per superar aquesta limitació, els dispositius permeten modificar la seva adreça. A la Figura 21 es pot observar la connexió entre el microcontrolador que fa de mestre i diferents components que fan d'esclaus.

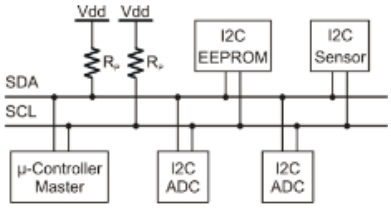


Figura 21. Connexió I2C amb un mestre i quatre esclaus

Fins ara s'ha esmentat que només cal dos fils per enviar i rebre les dades, però es necessiten dos requisits més, el primer és que els dos dispositius estiguin referenciats a massa i el segon, que s'utilitzin de dues resistències addicionals a les senyals SCL i SDA. Aquestes resistències no són especials, únicament es diferencien perquè es troben col·locades d'una manera

determinada. Depenent de la configuració se'ls anomena resistències *pull up* o *pull down*. La funció principal és determinar un estat lògic en el pin d'entrada del circuit quan aquest està en repòs i evitar lectures errònies produïdes per sorolls provinents d'altres components electrònics.

En la configuració pull down, quan el circuit està en repòs, com es mostra en la Figura 22, la caiguda de tensió que hi ha a la resistència és pràcticament nul·la (estat baix), en canvi si es prem el polsador P1, deixa passar el corrent i en conseqüència es crea una diferència de potencial de 5V(estat alt). Per altra banda, en la configuració *pull up*, quan el circuit està en repòs, P1 sense prémer, la caiguda de tensió és de 5V (estat alt), en canvi quan es prem el polsador P1 es deriva tot el corrent a massa i la caiguda de tensió és nul·la (estat baix). El valor d'aquestes resistències normalment es troba entre 1,8 kOhms i 47 kOhms. Com més baix sigui el valor de la resistència més s'incrementa el consum dels integrats però en canvi es disminueix la sensibilitat al soroll i es millora el temps entre els flancs de pujada i de baixada del senyal.

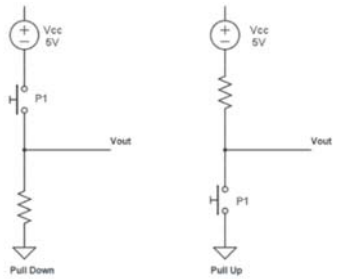


Figura 22. Configuració *pull up* i *pull down*

Un altre aspecte a tenir en compte en el I2C és que pot tenir més d'un mestre. Aquests són els únics que poden controlar la línia de SCL i només un d'aquest pot iniciar una transmissió i els altres hauran d'esperar que aquesta finalitzi, i així successivament fins que acabin totes les transmissions. Quan hi ha més d'un mestre, els que no controlen la línia SCL poden actuar com esclaus.

És una condició sine qua non que l'esclau o els esclaus, si n'hi han més d'un, hagin d'esperar que un mestre li enviï la seva adreça per poder iniciar una comunicació i enviar-li les dades.

En la trama d'enviament de dades el protocol I2C reserva alguns paràmetres per poder realitzar la transmissió. Una trama està formada per 6 camps. El primer és d'inici i és una condició que s'ha de complir per iniciar l'enviament de la trama. El segon és l'adreça, format

per set bits que són l'adreça del dispositiu amb el que es vol comunicar. El tercer anomenat llegir/escriure on hi ha un bit que estableix si el mestre vol llegir o enviar dades. El quart camp és el ACK, format per un bit que es posa al final de cada byte i permet assegurar que el byte ha arribat correctament i es pot continuar amb la trama. El cinquè camp és on hi han els bits de les dades de configuració de l'esclau enviades pel mestre o les dades que vol enviar l'esclau al mestre. Un com enviat aquest bits es tornar enviar un ACK i es repeteix tantes vegades com sigui necessari. El sisè camp conté la condició que indica la finalització de la transmissió deixant el *bus* lliure per a una altre transmissió. Veure la Figura 23.

Figura 23. Els diferents camps de la comunicació I2C

Per començar una comunicació el mestre posa el senyal SDA a nivell baix, que actua com un senyal d'atenció per a tots els dispositius connectats, mentre el senyal de rellotge està a nivell alt. A continuació el mestre envia la direcció del dispositiu a qui es vol adreçar i defineix si és una operació de lectura o escriptura. Aquests bits són llegits per tots els esclaus i comparen l'adreça enviada amb la seva pròpia. Si les adreces no coincideixen, l'esclau no fa cas de les dades enviades pel *bus* fins al bit de parar. Si les adreces coincideixen l'esclau envia el senyal de resposta ASK. Un cop el mestre rep aquesta senyal, pot començar a transmetre o a rebre dades. Si és el mestre el que envia les dades, al final d'aquestes envia la condició de parar, el qual indica que la transmissió ha finalitzat, deixant el *bus* lliure per començar una nova comunicació. Quan és el mestre que rep les dades li correspon a aquest interpretar-les per saber quan s'han acabat de transmetre i finalitza la comunicació. En la Figura 24, es pot observar la condició d'inici i finalització de la comunicació, l'adreça formada pels bits A0 fins a A7 i les dades que són col·locades en els 8 bits de D0 a D7.

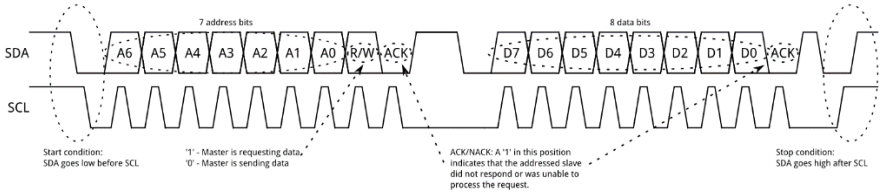


Figura 24. Comunicació I2C adreçament del mestre i enviament de les dades

De vegades, la demanda de dades per part del mestre supera la capacitat de l'esclau de proporcionar-les. Això pot ser degut al fet que les dades encara no estan preparades pel motiu que sigui, per exemple, quan l'esclau no ha completat encara una conversió analògic a digital

o bé, perquè una operació anterior encara no ha finalitzat, com pot ser el cas d'una EEPROM que no ha finalitzat d'escriure a la memòria no volàtil.

En aquest cas, alguns dispositius esclaus executen el que es coneix com a "estirament de rellotge". Normalment el rellotge és controlat pel dispositiu mestre i els esclaus simplement posen o prenen dades en el *bus* en resposta als impulsos del rellotge. En qualsevol punt del procés de transferència de dades, un esclau adreçat pot mantenir la línia SCL en estat baix després que el mestre l'alliberi. El mestre no enviarà polsos de rellotge addicionals o de transferència de dades fins que l'esclau alliberi la línia SCL.

4.1.3 Comunicació SPI combinada amb la comunicació I2C

A la placa DE0-Nano es combina la comunicació SPI amb la comunicació I2C. Això és possible per dues raons: una per la manera amb que se selecciona l'esclau i l'altre per com i quan s'envien les dades. Tal com mostra la Figura 25 el senyal CLK de l'acceleròmetre i el senyal SCL de la memòria que corresponen a les senyals de rellotge de cada un s'ajunten i van al pin F2 Cyclone IV. Les senyals per on es transmeten les dades SDIO i SDA també s'ajunten i van a parar al pin F1 de la Cyclone IV.

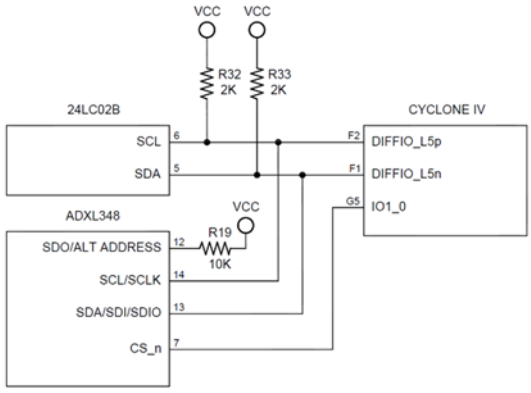


Figura 25. Comunicació de l'acceleròmetre i la memòria amb la Cyclone IV

El tipus de comunicació SPI que s'utilitza és de tres senyals, motiu per el qual el senyal SDO de l'acceleròmetre va connectada a l'alimentació VCC. El senyal d'habilitació de l'acceleròmetre està connectada al pin G1 de la Cyclone IV, fet que permet escollir amb quin dels dos components ens volem comunicar atès que un anirà per adreça i l'altre per habilitació. L'acceleròmetre no envia dades mentre el pin d'habilitació estigui a nivell alt i la memòria 24LC02B tampoc envia cap dada fins que no rebi l'adreça de la Cyclone IV.

5 COMUNICACIONS EN SÈRIE A LA PLACA DE0-Nano

En aquest projecte per desenvolupar les comunicacions en sèrie a la placa DE0-Nano utilitzarem tres dels programes que proporciona Altera. Aquest programes són: el DE0_NANO_EEPROM, el DE0_NANO i el DE0_NANO_G_Sensor.

El programa DE0_NANO_EEPROM és el que s'utilitza per estudiar i realitzar la comunicació sèrie I2C entre la memòria EEPROM 24LC02B i la Cyclone IV.

El programa DE0_NANO, és el que s'utilitza per estudiar i realitzar la comunicació sèrie SPI de quatre senyals entre el convertidor analògic a digital 128S022 i la Cyclone IV.

El programa DE0_NANO_G_Sensor, és el que s'utilitza per per estudiar i realitzar la comunicació sèrie SPI de tres senyals entre l'acceleròmetre ADXL345 i la Cyclone IV.

Els programes estan escrits amb llenguatge Verilog i per facilitar la seva comprensió s'ha cregut convenient representar-los en forma d'esquemes o blocs. Tots els programes tenen variables internes i externes. Les variables externes es representen com entrades o sortides dels blocs i les variables internes normalment en forma d'etiqueta rectangular acabada en fletxa en un dels extrems i amb el nom a dintre. La fletxa indica el sentit de sortida. A la Figura 26 es representa una variable interna.

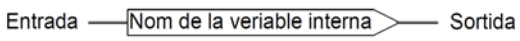


Figura 26. Representació de les variables internes

Per representar un *bus* de dades s'utilitzen línies més gruixudes. En cas que es vulgui assignar una dada del *bus* a una variable es grafia una etiqueta de forma rectangular on s'hi escriu el nom de la variable i entre claudàtors quina dada es pretent assignar. Tal com es pot observar a la Figura 27 per la part esquerra entra un *bus* de dades, on s'indica que és un *bus* d'entrada, format per 10 bits i s'assigna a la variable "fft" el bit número 5.

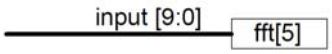


Figura 27. Representació d'una línia *bus* i la variable fft que se li assigna el bit 5 d'aquest

5.1 La memòria EEPROM 24LC02B

Una memòria EEPROM és una memòria RAM no volàtil, la qual es pot programar o esborrar el contingut utilitzant impulsos elèctrics. Aquesta té un direccionament individual de bytes per a escriure les dades. La memòria que hi ha a la placa és la 24LC02B de 2Kbits. Utilitza una comunicació sèrie I2C. Pel funcionament correcte s'ha d'alimentar amb una tensió de 2,5 V. A la Figura 28 s'observa els pins d'entrada i sortida de la 24LC02B i ha la Taula 6 una descripció de cada un d'ells.

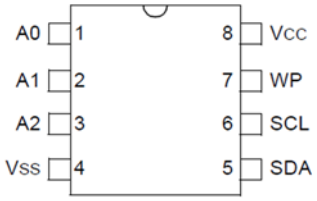


Figura 28. Memòria EEPROM 24LC02B

Número	PIN	Descripció
1	A0	No s'utilitza
2	A1	No s'utilitza
3	A2	No s'utilitza
4	VSS	Aquest pin va connectat a massa
5	SDA	Entrada-Sortida per la comunicació seria
6	SCL	Entrada de rellotge per a la comunicació sèrie
7	WP	Aquest pin va connectat a massa
8	VCC	Entrada d'alimentació

Taula 6. Descripció dels pins de la memòria EEPROM 24LC02B

5.1.1 Comunicació de la memòria 24LC02B amb la Cyclone IV

La memòria 24LC02B únicament permet la comunicació sèrie I2C que sempre actuarà com esclau. Aquesta comunicació és possible perquè la configuració de la placa tal com se subministra de fàbrica, la entrada-sortida SDA i l'entrada SCL van connectades directament a la Cyclone IV. L'adreça que porta per defecte és 8'hA0. A la Figura 29 s'observa com estan connectats els pins a la placa DE0-Nano. Les resistències *pull up* són de 2 kOhms per tant la màxima velocitat de comunicació és de 400 kHz segons especifica el full de característiques.

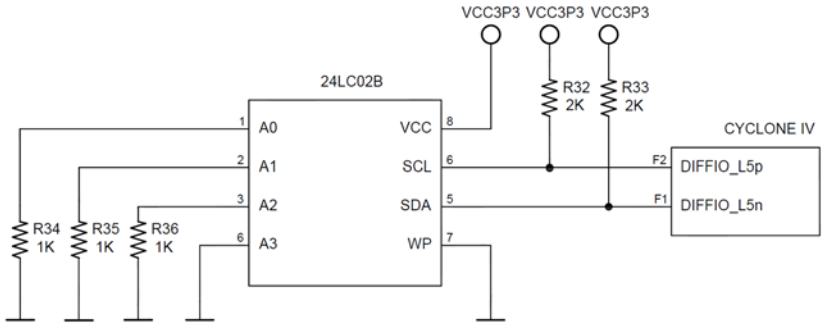


Figura 29. Connexió de la memòria 24LC02B a la placa DE0-Nano

A continuació s'explica la comunicació que s'utilitza en el programa DE0_NANO_EEPROM. Per iniciar una comunicació d'escriptura s'ha de complir la condició de INICI. Aquesta es compleix sempre que la línia de *bus* no estigui ocupada i que la senyal SDA canviï d'estat alt a baix mentre la senyal SCL es mantingui en estat alt. En el moment que es compleix aquesta condició s'envia el byte de control. Tal com es mostra a la Figura 30 aquest byte està format per l'adreça del la memòria EEPROM i per un bit que indica si es vol realitzar una operació d'escriptura o de lectura. L'adreça de la memòria EEPROM està formada per set bits i només s'utilitzen els quatre més significatius, el valor de la resta és indiferent.

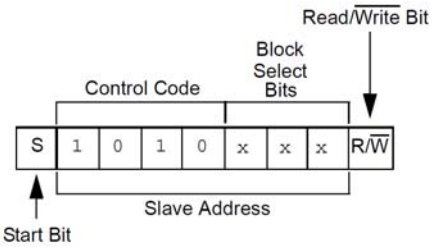


Figura 30. Format de l'enviament de l'adreça de la memòria 24LC02B

Un cop enviat el byte de control, la Cyclone IV deixa lliure la línia de *bus* fent que la memòria pugui posar la línia SDA a nivell baix, és a dir, que envia el bit ACK. Un cop enviat aquest bit el mestre torna agafar el control del *bus* i remet l'adreça a on es vol escriure les dades. Enviades les dades es torna a repetir el procés per enviar el bit ACK. Un cop acabat, el mestre pot enviar les dades a l'adreça seleccionada. Per últim, es repeteix el procés d'enviar el bit ACK i finalment el mestre dur a terme la condició de *stop* per acabar la comunicació. A la Figura 31 es pot observar tota la seqüencia.

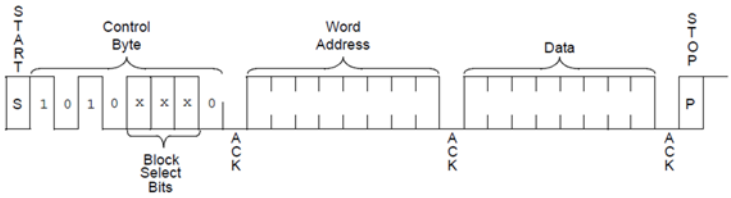


Figura 31. Seqüència per escriure a una adreça

5.2 El convertidor analògic a digital 128S022

El convertidor analògic a digital 128S022 és de 12 bits i disposa de vuit canals de baix consum. Aquest convertidor és especialment indicat per a velocitats de rendiment de conversió de 50 ksps a 200 ksps. La seva arquitectura es basa en el registre d'aproximació successiva. Les dades convertides en binari són compatibles amb diferents estàndards, com ara SPI, QSPI, Micro*wire* i moltes interfícies seriades DSP. A la Figura 32 es pot observar els pins d'entrada i sortida del 128S022 i a la Taula 7 una descripció de cada un d'ells.

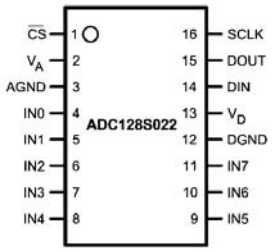


Figura 32. Convertidor analògic a digital 128S022

Número	PIN	Descripció
1	/CS	Entrada d'habilitació de l'acceleròmetre
2	VA	Entrada de tensió analògica
3	AGND	Aquest pin va connectat a la massa digital
4	IN0	Entrada analògica 0
5	IN1	Entrada analògica 1
6	IN2	Entrada analògica 2
7	IN3	Entrada analògica 3
8	IN4	Entrada analògica 4
9	IN5	Entrada analògica 5
10	IN6	Entrada analògica 6
11	IN7	Entrada analògica 7
12	DGND	Aquest pin va connectat a la massa analògica
13	VD	Entrada de tensió digital
14	DIN	Entrada de dades de la comunicació SPI
15	DOUT	Sortida de dades de la comunicació SPI
16	SCLK	Entrada de rellotge per les comunicacions sèrie SPI

Taula 7. Descripció dels pins del convertidor analògic a digital 128S022

5.2.1 Comunicació del convertidor analògic a digital 128S022 amb la Cyclone IV

El convertidor analògic a digital únicament permet la comunicació sèrie SPI de quatre senyals i aquest sempre actua com esclau. Aquesta comunicació és possible perquè la configuració de la placa tal com se subministra de fàbrica, les entrades CS_n, DIN, SCLK i la sortida DOUT del convertidor van connectades directament a la Cyclone IV. L'alimentació de les entrades analògiques del convertidor és 3,3 V i motiu pel qual el voltatge màxim que podem posar a les entrades és de 3,3 V. L'alimentació digital és de 3,3V. A la Figura 33 s'observa com estan connectats els pins a la placa DE0-Nano. Les entrades analògiques passen individualment per un filtre pas baix abans d'anar al seu pin corresponent del connector JP3.

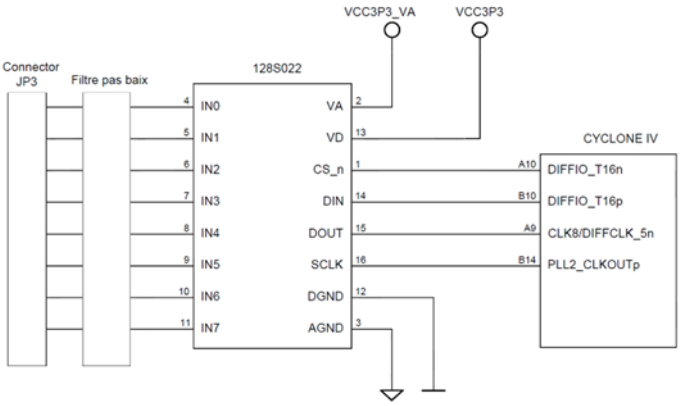


Figura 33. Connexió del convertidor analògic a digital 128S022 a la placa DE0-Nano

Aquesta comunicació sèrie comença quan l'entrada CS, anomenada habilitació, passa d'estat alt a estat baix i arriba un flanc de baixada del senyal de rellotge. La transmissió acaba després de 16 períodes de rellotge. Tal com es mostra a la Figura 34 la Cyclone IV escriu les dades de l'entrada DIN en cada flanc de baixada i llegeix les dades de la sortida DOUT en cada flanc de pujada. A la sortida DOUT els 3 primer períodes de rellotge són el temps d'adquisició de les dades i en els 13 períodes següents es realitza la conversió. Un cop finalitzar el procés de conversió es torna a començar sempre que l'entrada CS no estigui en estat alt, i si ho està la sortida DOUT val alta impedància.

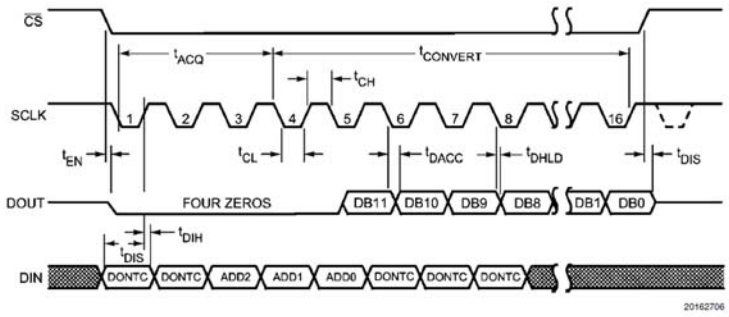


Figura 34. Comunicació SPI de quatre senyals

En el moment que l'entrada CS val 0 per l'entrada DIN tal com es mostra a la Figura 35 es rep el registre de control. Els valors dels bits D7,D6,D2,D1 i D0 no serveixen per res i els bits D5, D4 i D3 determinen quina entrada es vol seleccionar.

Bit 7 (MSB)	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DONTC	**DONTC**	**ADD2**	**ADD1**	**ADD0**	**DONTC**	**DONTC**	**DONTC**

Figura 35. Bits del registre de control

Les diferents combinacions que poden tenir ADD2, ADD1 i ADD0 són les que es mostren a la Taula 8. L'entrada que se selecciona per defecte és la INT0.

Entrada	ADD2	ADD1	ADD0
INT0	0	0	0
INT1	0	0	1
INT2	0	1	0
INT3	0	1	1
INT4	1	0	0
INT5	1	0	1
INT6	1	1	0
INT7	1	1	1

Taula 8. Entrades analògiques del convertidor

5.3 L'acceleròmetre ADXL345

L'acceleròmetre ADXL345 és de 3 eixos de molt baix consum, amb una resolució màxima de 13 bits i amb una mesura màxima de +/-16g. Els tipus de comunicacions que pot fer servir són: la SPI de 3 senyals, la SPI de 4 senyals i la I2C. Algunes de les aplicacions s'utilitzen en l'àmbit de la instrumentació mèdica, de la instrumentació industrial i amb els videojocs. L'acceleròmetre permet mètodes de baixa potència i una gestió intel·ligent d'energia basada

en l'enviament de dades i de moviment. Ofereix diverses funcions especials per a la detecció de moviment. La detecció d'activitat o inactivitat es detecta amb la presència o manca de moviment de l'acceleròmetre, mitjançant la comparació del llindar establert per l'usuari amb qualsevol dels eixos. També disposa de detecció de caiguda del dispositiu i la detecció de cops siguin individuals o dobles. Aquestes funcions es poden assignar individualment a qualsevol dels dos pins de sortida d'interrupció. També disposa d'un sistema integrat de gestió de memòria basat amb el FIFO. A la Figura 36 es pot observar els pins d'entrada i sortida de ADXL345 i la Taula 9 una descripció de cada un d'ells.

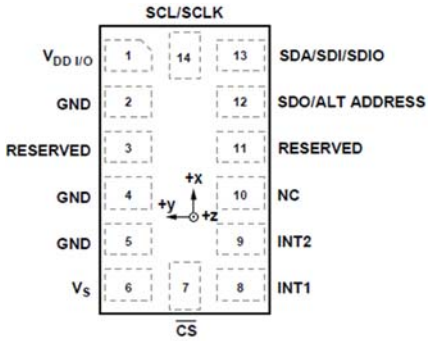


Figura 36. Acceleròmetre ADXL345

Número	PIN	Descripció
1	VDD I/O	Tensió d'alimentació per la interfície digital
2	GND	Aquest pin va connectat a massa
3	Reserved	Aquest pin ha d'anar connectat a Vs
4	GND	Aquest pin va connectat a massa
5	GND	Aquest pin va connectat a massa
6	Vs	Tensió d'alimentació
7	/CS	Entrada d'habilitació de l'acceleròmetre
8	INT1	Sortida d'interrupcions 1
9	INT2	Sortida d'interrupcions 2
10	NC	No s'utilitza
11	Reserved	Aquest pin ha d'anar connectat a GND
12	SDO/Alt Address	SDO per la comunicació SPI-4 i ALT ADDRESS per la I2C
13	SDA/SDI/SDIO	SDA per la comunicació I2C, SDI per la SPI-4 i SDIO per SPI-3
14	SCL/SCLK	Entrada de rellotge per les comunicacions sèrie SPI i I2C

Taula 9. Descripció dels pins de l'acceleròmetre ADXL345

5.3.1 Comunicació de l'acceleròmetre amb la Cyclone IV

L'acceleròmetre es pot comunicar per I2C, per SPI de tres senyals o per SPI de quatre senyals i en tots els casos sempre actua com esclau, mai com a mestre. Tot i que admet aquest tipus de comunicació segons la configuració de la placa tal com se subministra de fàbrica només

és possible realitzar la comunicació per I2C o per la SPI de tres senyals. A la Figura 37 es pot observar com va connectat a la placa DE0-Nano.

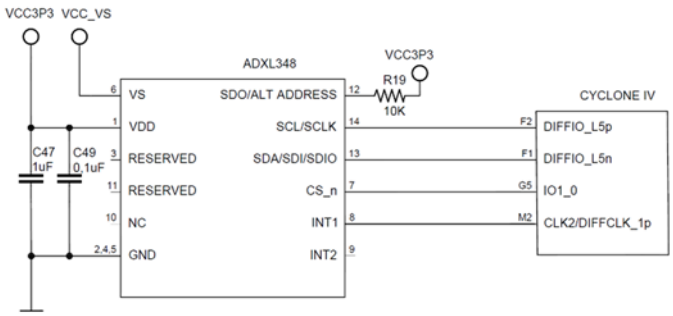


Figura 37. Connexió de l'acceleròmetre a la placa DE0-Nano

La comunicació I2C és possible quan l'entrada del senyal MOSI va connectada a massa o bé a l'alimentació VCC3P3, segons el full de característiques, i que l'entrada del senyal CS estigui sempre a nivell baix. D'acord amb la configuració de l'acceleròmetre l'entrada CS de la placa va connectada a una entrada de la Cyclone IV i l'hem de posar a nivell baix. Per altre banda, el senyal MOSI va connectat a l'alimentació mitjançant una resistència de 10 kOhms i en conseqüència, l'adreça de l'acceleròmetre per establir una comunicació I2C és 0x1D. Si es dóna el cas també es pot canviar l'adreça a 0x53 si es treu la resistència R19 i es connectar l'entrada del senyal MOSI a massa.

La comunicació SPI de quatre senyals no es pot portar a terme atès que l'entrada del senyal MOSI va connectat a VCC3P3 fent que aquesta sempre estigui a nivell alt i impedeix l'enviament de dades del mestre cap a l'esclau. Si pel motiu que sigui convingués realitzar aquesta comunicació s'haurien de fer els canvis següents: treure la resistència R19 i en el seu lloc connectar un cable fins a un dels pins GPIO de la Cyclone IV.

La configuració de la transmissió de les dades és en mode 3 perquè la configuració de fabrica del CPOL i del CPHA valen 1. Una trama de lectura i escriptura es transmet en 16 polsos de rellotge i es complementa si és el cas en múltiples de 8 quan s'envien múltiples bytes. La durada de temps d'un bit és entre dos flancs de baixada de rellotge. La Figura 38 mostra la transmissió d'una trama de lectura i escriptura. Quan el bit R/W val 0 els valors dels bits D7 a D0 s'escriuen a l'acceleròmetre i quan val 1 es llegeixen. Si el bit MB val 0 l'adreça del bits A5 a A0 es manté i quan val 1 s'incrementa l'adreça automàticament en el mode de múltiple bytes. Els bits A5 a A0 s'hi pot posar qualsevol adreça de l'acceleròmetre.

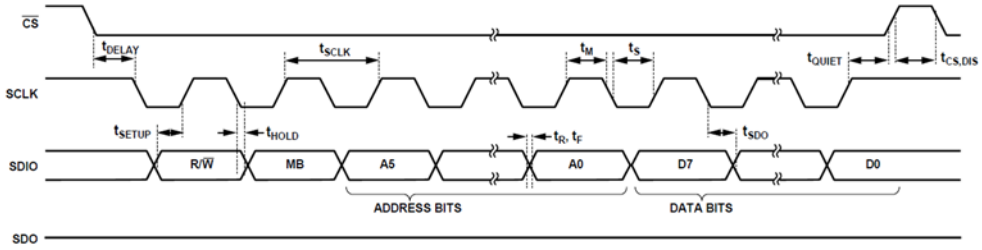


Figura 38. Comunicació SPI de tres senyals

El format de les dades canvia en funció del mode de funcionament. En el programa DE0_NANO_G_Sensor segons l'adreça 0x31 el format de les dades és justificada a la dreta amb una resolució de 10 bits i amb un rang de +/- 2g. Tal com es mostra a la Figura 39 amb aquesta configuració el bit D0 de l'adreça DATAx0 sempre té un valor 0, els bits D1 a D7 de l'adreça DATAX0 i els bits D0 i D1 de l'adreça DATAX1 contenen el valor de l'eix X.

D7	D6	D5	D4	D3	D2	D1	D0
D7	D6	D5	D4	D3	D2	D1	D0

D7	D6	D5	D4	D3	D2	D1	D0
D7	D6	D5	D4	D3	D2	D1	0

DADES DE L'EIX X

Figura 39. Format amb que s'envien les dades

6 PROGRAMA DE0_NANO_EEPROM

Primer de tot cal esmentar que s'ha modificat el programa per poder assignar el valor de la dada utilitzant els interruptors i s'ha configurat que el LED0 s'il·lumini quan s'han transmès les dades. Això s'ha fet perquè no sigui necessari modificar el programa cada vegada que es vol enviar una dada diferent i per tenir una referència de quan s'ha acabat d'executar el programa.

Per a la implementació de la comunicació I2C s'utilitza el programa DE0_NANO_EEPROM. Aquest programa utilitza els polsadors KEY0 i KEY1, el quatre interruptors, el LED0 i la memòria 24LC02B. De la memòria 22AA02B s'utilitza l'adreça 8'b0.

Tal com es pot observar a la Figura 40 el mòdul està format per tres entrades, dues sortides, una entrada-sortida i set variables internes. Les entrades són la CLOK_50, la KEY i la SW. Les sortides són la I2C_SCLK, la COUNT, la SD_COUNTER i la LED. L'entrada-sortida és la I2C_SDAT. Les variables internes són la reset_n, la GO, la SD_COUNTER, la SDI, la SCLK, la COUNT, la "final" i la "valor".

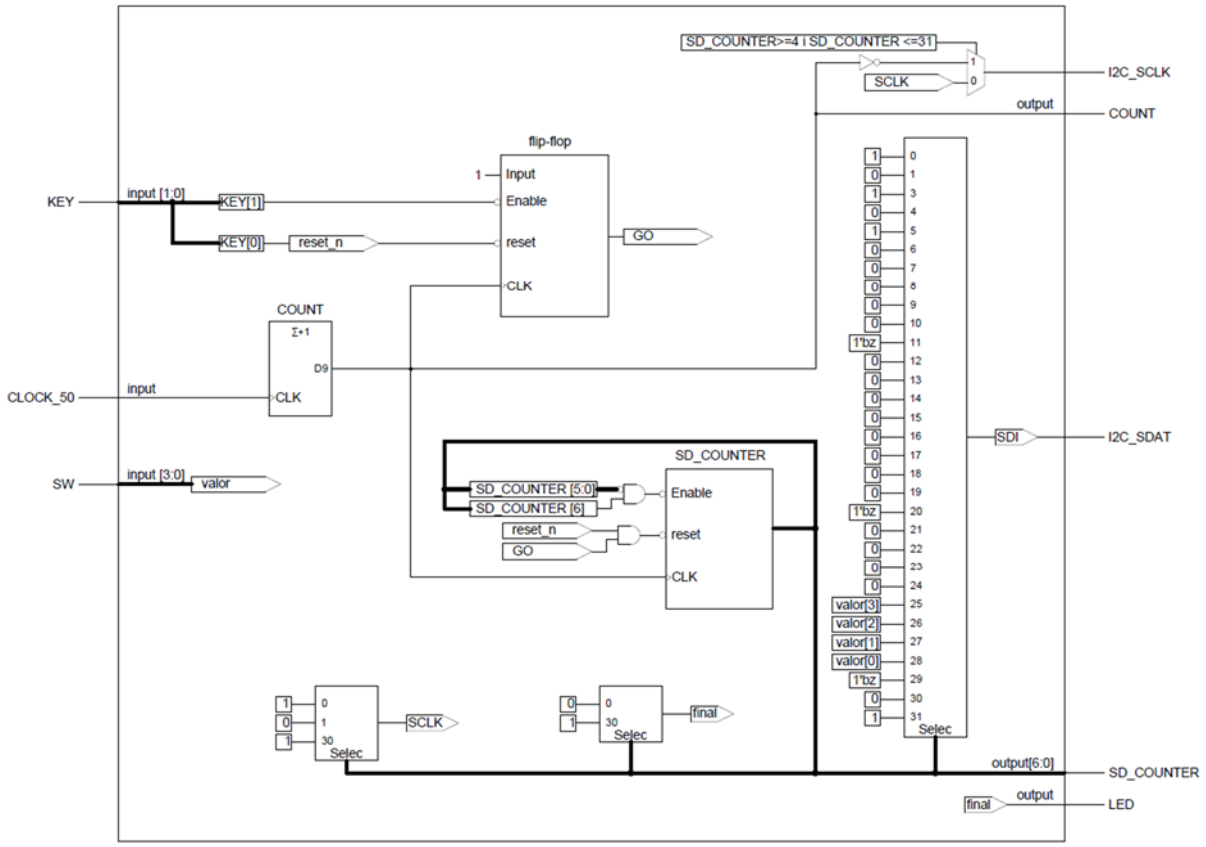


Figura 40. Estructura del mòdul DE0_NANO_EEPROM

La funció d'aquest mòdul és escollir el valor que es vol enviar a la memòria utilitzant el quatre interruptors, un cop enviada es llegirà aquesta adreça utilitzant l'aplicació Control Panel. Com que només es disposa de quatre interruptors el nombre màxim de valors que es podrà representar és de 16. Els valors amb l'aplicació Control Panel es representen amb hexadecimal. Per iniciar la comunicació i enviar aquesta dada a la memòria és imprescindible polsar el polsador KEY1. En aquest mòdul s'hi duen a terme quatre estructures *always*. Un cop les dades estiguin enviades s'il·lumina el LED0.

La primera estructura únicament serveix per augmentar el valor del comptador COUNT amb una unitat en cada flanc de pujada del rellotge CLOCK_50. La funció d'aquest comptador és establir un temps d'execució de les altres estructures *always*. El motiu és perquè la comunicació amb aquesta memòria, com especifica el full de característiques, no pot anar a la velocitat de rellotge que proporciona l'entrada CLOCK_50. A la Figura 41 es pot observar la representació gràfica de la primera estructura *always*.

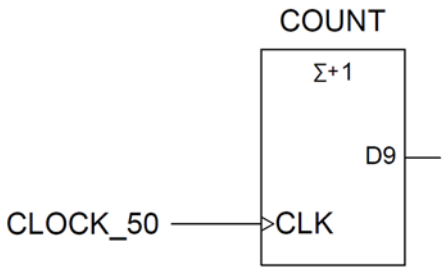


Figura 41. Primera estructura always

La segona estructura s'activa sempre que hi ha un flanc de pujada del bit 9 de la variable COUNT o un flanc de baixada de la variable reset_n. La funció d'aquest bloc és assignar a la variable GO el valor 1 indicant que comenci la comunicació I2C. Perquè això succeeixi s'ha de polsar el polsador KEY1. Si en algun moment es polsa el polsador KEY0 el valor de la variable GO és 0. A la Figura 42 es pot observar la representació gràfica de la segona estructura *always*.

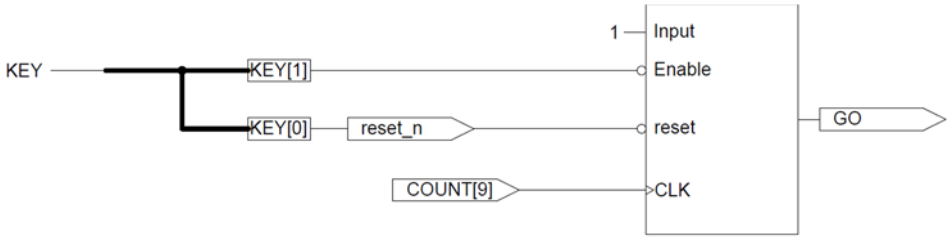


Figura 42. Segona estructura always

La tercera estructura s'activa sempre que hi ha un flanc de pujada del bit 9 de la variable COUNT o un flanc de baixada de la variable reset_n . Si el valor de la variable GO és 1 se suma una unitat a la variable SD_COUNTER sempre que el seu valor sigui inferior a 33, és a dir, el comptador pot arribar a tenir fins un valor de 32. En cas contrari el valor de SD_COUNTER és 0. Si en algun moment es polsa el polsador KEY0 es fa un *reset* del comptador. A la Figura 43 es pot observar la representació gràfica de la tercera estructura *always*.

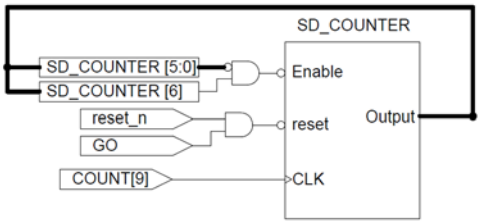


Figura 43. Tercera estructura always

Finalment la quarta estructura és per on s'envien i es reben les dades. En aquesta hi ha una estructura *case* i utilitzant el valor de SD_COUNTER es va entrant a cada un dels estats d'aquesta estructura. Per representar aquesta estructura s'utilitzen tres multiplexors.

El primer *case* és el que assigna valors a la variable SDI. Cada entrada correspon a un bit que s'ha de enviar per la entrada-sortida I2C_SDAT. Per aquesta variable s'envien les dades i al ser tipus *reg* guarda el seu valor fins que no se li assigna un de nou. A la Figura 44 es pot observar l'estructura *case* descrita.

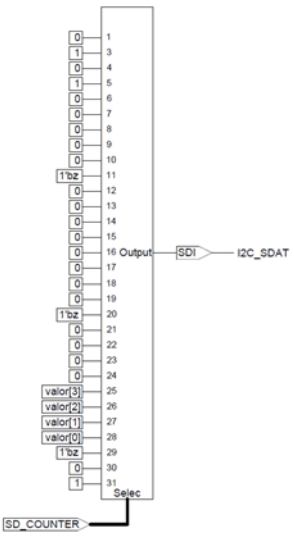


Figura 44. Primera estructura case

El segon *case* assigna valors a la variable SCLK. Només s'assigna valor en tres ocasions quan SD_COUNTER val 0,1 i 30. Com que la variable SCLK és tipus *reg* guarda l'últim valor que se li ha assignat. El valor d'aquesta variable s'envia per la sortida I2C_SCLK quan el valor del comptador SD_COUNTER és inferior a 4 i més gran que 31. Quan I2C_SCLK es troba dins d'aquest valors el valor de I2C_SCLK és el negat de la COUNT[9]. A la Figura 45 es pot observar l'estructura *case* descrita i a la Figura 46 com se selecciona el valor que surt per la sortida I2C_SCLK.

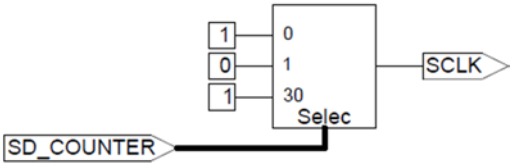


Figura 45. Segona estructura case

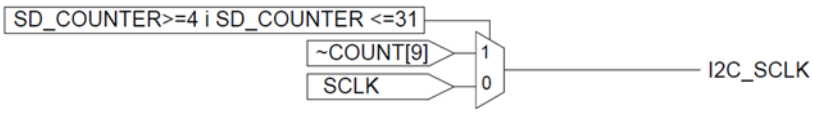


Figura 46. El valor de la sortida I2C_SCLK depèn de SD_COUNTER

El tercer *case* només s'utilitza per assignar els valors 0 o 1 a la variable "final". Aquesta variable indica que s'ha acabat la comunicació. Com que va connectada a la sortida LED quan el seu valor és 1 fa que s'il·lumini el LED0. Igual que els altres dos *case* la variable "final" és tipus *reg* i guarda el seu valor fins que no se li assigna un de nou. A la Figura 47 es pot observar l'estructura *case* descrita.

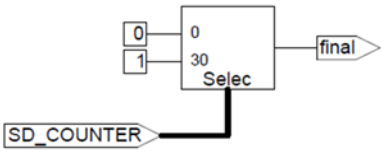


Figura 47. Tercera estructura *case*

El funcionament del mòdul és el següent: mentre no es premi el polsador KEY1 la variable GO val 0, per tant, el comptador SD_COUNTER val 0 i com a conseqüència les tres estructures *case* assignen a les corresponents variables el cas 0. En el moment que es polsa el polsador KEY1 el comptador SD_COUNT va augmentant el seu valor en cada flanc de pujada de

rellotge i comença la comunicació I2C. A les Figures 48 i 49 es mostra la seqüència d'un exemple amb el supòsit que el valor de SW és de 1010.

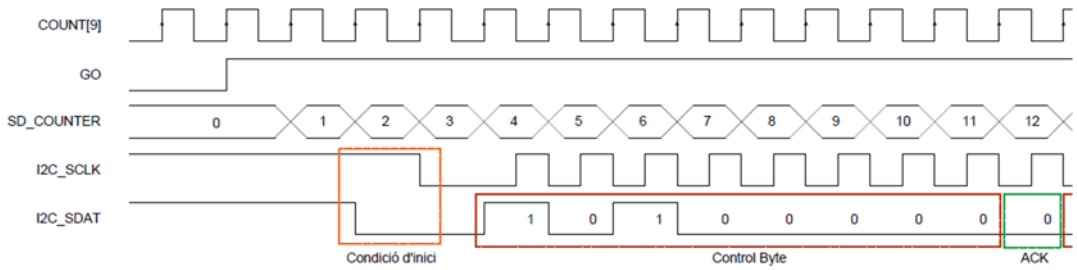


Figura 48. Primera part de la comunicació I2C

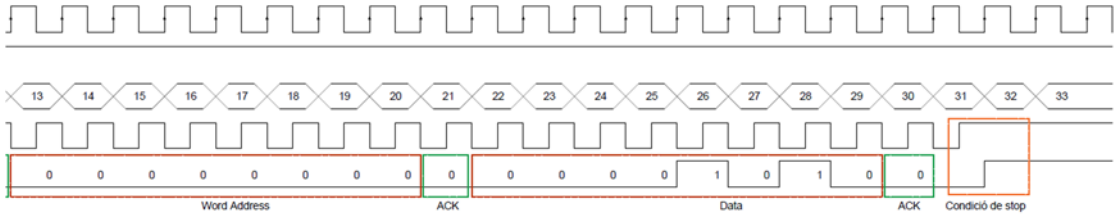


Figura 49. Segona part de la comunicació I2C

Com es pot observar a les figures anteriors les dades es transmeten quan la senyal I2C_SCLK es troba a nivell alt i es canvien de valor quan es troba a nivell baix.

Un cop acabada la seqüencia el LED0 queda il·luminat fins que no es reinicia el programa. A la Figura 50 es pot observar l'aplicació Control Panel amb el valor de l'adreça 0 que se li ha assignat el valor 1010 de l'exemple anterior. Com que es representa amb hexadecimal aquest valor és A. Cada adreça està formada per dos bytes i com a conseqüència el valor de rDATA és 000A.

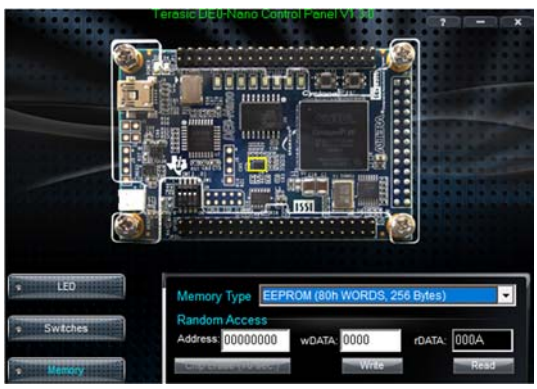


Figura 50. Control Panel

6.1 Modificació del programa DE0_NANO_EEPROM

A vegades la informació que s'envia a una memòria no sempre és formada per una única dada sinó que poden ser varies. La memòria disposa d'un mode de múltiple escriptura. La idea inicial era modificar el programa DE0_NANO_EEPROM per poder dur a terme aquest mode però, els canvis que s'han de realitzar són significatius i s'ha cregut convenient crear un nou programa amb el nom de DE0_NANO_EEPROM_V2 que permeti enviar diferent dades.

El funcionament d'aquest programa es basa en el DE0_NANO_EEPROM. El byte de control, la direcció i la primera dada es transmeten de la mateixa manera però en lloc de generar la condició de stop s'envia un nou byte de dades. Com a màxim es pot enviar fins a 7 bytes que es guarden temporalment a un *buffer* intern de la memòria. Un cop transmès l'últim byte es produeix la condició de stop i la memòria transfereix les dades emmagatzemades a l'adreça corresponent. A la Figura 51 s'observa la seqüencia que s'ha de seguir. En aquest programa s'enviarà 5 bytes.

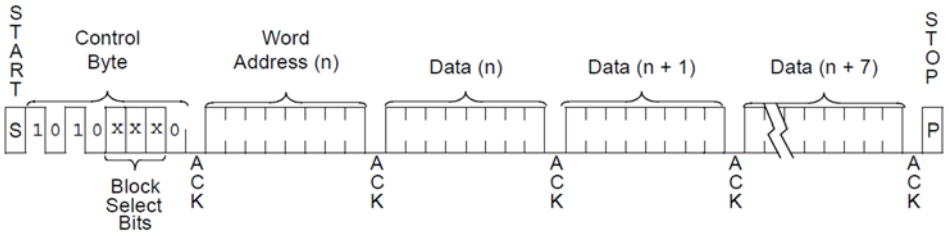


Figura 51. Seqüència de múltiple escriptura

Les modificacions que es realitzen són les següents: en lloc de utilitzar un comptador per fer el senyal de rellotge s'utilitza l'aplicació MegaWizard Plug-In Manager per crear el mòdul i2cpll el qual proporciona el senyal de rellotge, es canvien els llindars del comptador SD_COUNTER, s'afegeixen estats a l'estructura *case*, es treuen les sortides del mòdul principal COUNTER i SD_COUNTER i per últim, s'afegeix el mòdul EEPROM_CONTROL. A la Figura 52 es veu el mòdul DE0_NANO_EEPROM_V2.

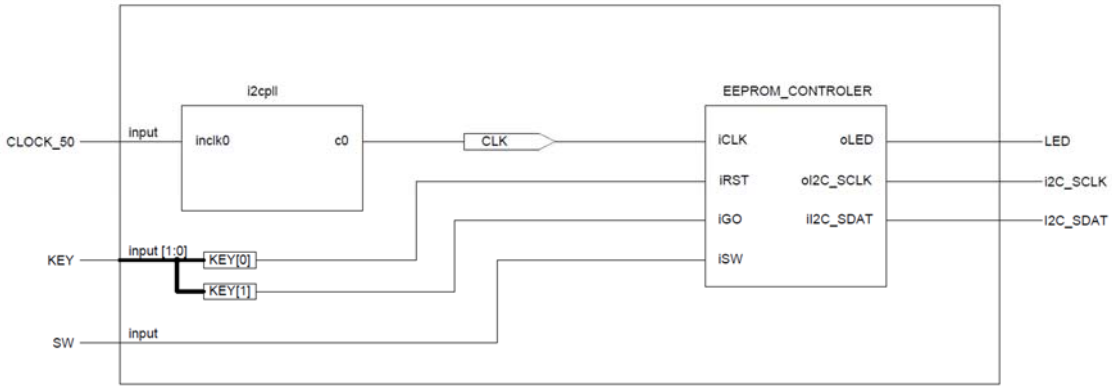


Figura 52. Estructura del mòdul DE0_NANO_EEPROM_V2

El mòdul i2cpll serveix com a senyal de rellotge, tant pel mòdul EEPROM_CONTROL com per a la comunicació I2C. Aquest substitueix la variable COUNT, motiu pel qual, el període del senyal de rellotge és de 10,24 µs. A la Figura 53 s'observa el mòdul i els paràmetres de configuració. El desfasament i el cicle de treball es mantenen respecte el senyal CLOCK_50.

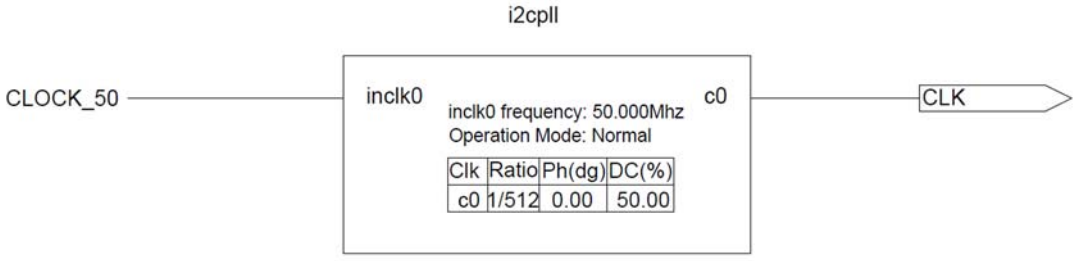


Figura 53. Estructura del mòdul i2cpll

El mòdul EEPROM_CONTROL conté tot el programa que hi ha en el DE0_NANO_EEPROM, amb les diferències que, la primera estructura *always* no hi és, el *case* de la quarta estructura té més estats i que es canvia el llinda de la variable SD_COUNTER i de la sortida oI2C_SCLK. El *case* té un total de 58 estats per tant, la condició d'habilitació de SD_COUNTER també canvia i augmenta el seu valor sempre que aquest sigui més petit de 60. També canvia la condició de la sortida de rellotge oI2C_CLK. Com que només es disposa de 4 interruptors aquests es combinen i es nega el seu valor per enviar diferents dades. A la Figura 54 s'observa l'estructura del mòdul.

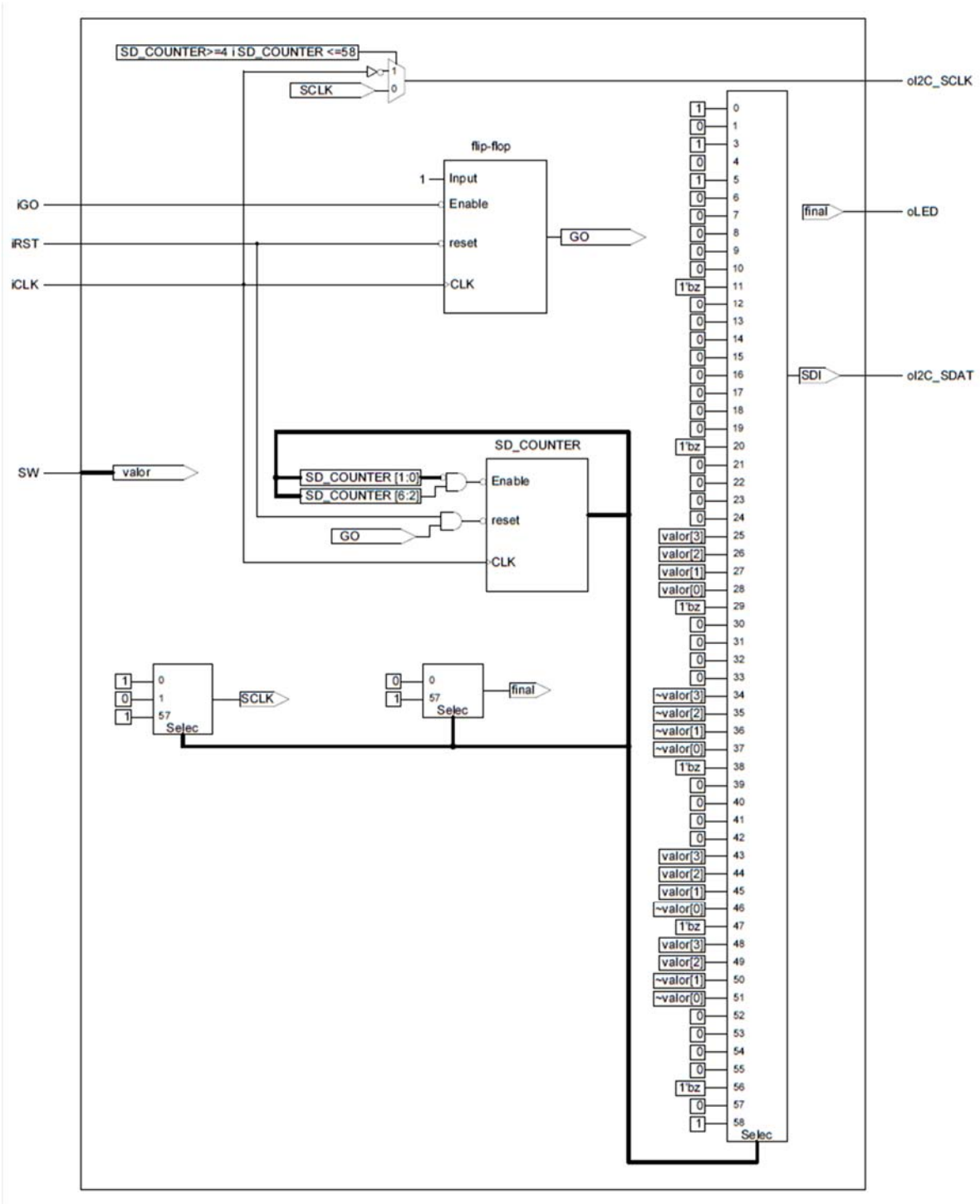


Figura 54. Estructura del mòdul EEPROM_CONTROLER

6.2 Proves, simulacions i resultats

En el programa DE0_NANO_EEPROM podem observar la seqüencia i visualitzar al mateix temps el resultat pel Control Panel utilitzant l'eina SingleTap II Logic Analizer. Les sortides COUNT i SD_COUNTER s'utilitzen en aquesta eina per visualitzar la seqüència. El problema

que he tingut és que al utilitzar una versió gratuïta del Quartus II aquesta dona error en el moment de compilar el programa. Com ha solució per poder comprovar que s'han escrit correctament les dades primer vaig transferir el programa a la placa DE0-Nano, seguidament vaig executar polsant el polsador KEY1, vaig obrir l'aplicació Control Panel, en l'apartat de memòries vaig seleccionar la EEPROM i finalment vaig preme "READ" per visualitzar el resultat.

Fet això va sortir el problema que un cop connectada la placa amb l'aplicació el programa només s'executava una vegada tot i què, es reinicies el programa. Per enviar un nou valor és necessari tancar l'aplicació, desconnectar la placa l'alimentació, tornar a carregar el programa, executar-lo de nou polsar el polsador KEY1 i finalment obrir l'aplicació i visualitzar el resultat.

Per no tenir que carregar el programa cada vegada s'ha convertit el fitxer DE0_NANO_EEPROM.sof a un fitxer tipus jic per poder-lo transferir a EPCS64, fent que cada cop que s'alimenti la placa es carregui el programa.

Per comprovar el funcionament del programa DE0_NANO_EEPROM_V2 també s'ha creat un arxiu jic amb el nom de multiple_write. S'ha transferit a la placa i s'ha obtingut els resultat esperat.

Amb els dos programes s'han provat diferents combinacions amb els interruptors i amb els canvis de valor d'adreça. Els resultats han estat els esperats, cada vegada s'han escrit correctament les dades a l'adreça escollida.

Una altre prova que s'ha realitzar es treure la limitació de la variable SD_COUNTER. D'aquesta manera el comptador sempre augmenta el seu valor. El programa funciona correctament ja que aquesta limitació només serveix perquè la variable SD_COUNTER no augmenti el seu valor un cop ha finalitzat la comunicació.

Quan s'envia el bit ACK el valor de SDI és 1'bz. Això significa que es deixa la línia I2C_SDAT lliure perquè la memòria enviï aquest bit com a resposta que ha rebut correctament les dades.

Per la sortida I2C_SCLK surt el valor negat del bit D9 de la variable COUNT i això fa que es realitzi correctament la comunicació I2C, canviant el valor de la sortida I2C_SDAT quan la senyal de rellotge és a nivell baix i enviant-lo a la memòria quan està a nivell alt.

Com a demostració del funcionament del programa DE0_NANO_EEPROM_V2 es posen tots els interruptors oberts fent que el valor de la variable SW sigui 1111 i es configura perquè es comenci escriure a partir de l'adreça 1. A la Taula 10 podem observar els valors que s'assignen a cada adreça.

Adreça	DATA2	DATA1
00000001	00000000	00001111
00000002	11000000	00001110

Taula 10. Valor de les adreces

Un cop executat el programa es comprova que s'ha escrit correctament amb l'aplicació Control Panel. A la Figura 55 es visualitza el resultat obtingut.

Figura 55. Visualització del resultat pel Control Panel

7 PROGRAMA DE0_NANO

Per a la implementació de la comunicació SPI de quatre senyals es fa servir el programa DE0_NANO. Aquest programa utilitza els polsadors KEY0 i KEY1, tres dels quatre interruptors, els vuit LEDs i el convertidor analògic a digital 128S022 de la placa. El programa permet escollir una de les entrades del convertidor mitjançant els interruptors i en funció de la tensió que s'apliqui a l'entrada, s'encendran els LEDs corresponents per representar aquest voltatge amb binari. Aquesta tensió l'aplica l'usuari a través del pin corresponent a l'entrada seleccionada del connector JP3. Perquè el programa entri en funcionament és imprescindible polsar el KEY1.

El programa es divideix en diferents mòduls que segueix una estructura jeràrquica. En el nivell més alt hi ha el fitxer DE0_NANO. Dintre aquest i trobem el mòdul ADC_CTRL on hi ha tot el programa i el mòdul spipll el qual és una megafunció que serveix per controlar el temps de tot el programa. La Figura 56 es pot observar l'estructura jeràrquica del programa.

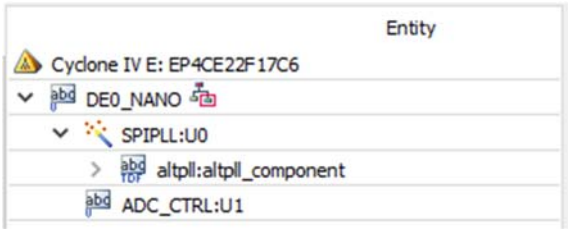


Figura 56. Estructura del programa DE0_NANO

7.1 Mòdul DE0_NANO

El mòdul DE0_NANO engloba els mòduls SPIPLL i ADC_CTRL. Les entrades d'aquest mòdul són la KEY, la CLOCK_50, la SW i la ADC_SDAT i les sortides són la LED, la ADC_CS_N, ADC_SADDR i la ADC_SDAT. Totes elles van a un pin de la Cyclone IV. A la Figura 57 es pot observar a quin pins estan assignades i si aquestes són una entrada o una sortida.

Node Name	Direction	Location
CLOCK_50	Unknown	PIN_R8
LED[0]	Unknown	PIN_A15
LED[1]	Unknown	PIN_A13
LED[2]	Unknown	PIN_B13
LED[3]	Unknown	PIN_A11
LED[4]	Unknown	PIN_D1
LED[5]	Unknown	PIN_F3
LED[6]	Unknown	PIN_B1
LED[7]	Unknown	PIN_L3
KEY[0]	Unknown	PIN_J15
KEY[1]	Unknown	PIN_E1
SW[0]	Unknown	PIN_M1
SW[1]	Unknown	PIN_T8
SW[2]	Unknown	PIN_B9
SW[3]	Unknown	PIN_M15
ADC_CS_N	Unknown	PIN_A10
ADC_SADDR	Unknown	PIN_B10
ADC_SCLK	Unknown	PIN_B14
ADC_SDAT	Unknown	PIN_A9

Figura 57. Assignació de les entrades i sortides als pins de la Cyclone IV

L'entrada KEY està formada per un *bus* de dos bits, on a cada un li correspon un polsador. El *bus* passa per l'etiqueta KEY0 i KEY1, atès que cada polsador té una funció diferent. El KEY0 va connectat a l'entrada iRST del mòdul ADC_CTRL per reiniciar-lo i el KEY1 a l'entrada iGO del mòdul ADC_CTRL perquè comenci a funcionar. L'entrada CLOCK_50 és la del senyal de rellotge proporcionada pel cristall oscil·lador de 50Mhz i va connectada a l'entrada inclk0 del mòdul SPIPLL. L'entrada SW està formada per un *bus* de tres bits, on a cada un li correspon un interruptor. Tot el *bus* va connectat a l'entrada iCH del mòdul ADC_CTRL i finalment l'entrada ADC_SAT va connectada a l'entrada iDOUT del mòdul ADC_CTRL, que per la qual es rebran les dades del convertidor.

La sortida oLED del mòdul ADC_CTRL va connectada a la sortida LED del mòdul DE_NANO i d'aquí va a als 8 LEDs de la placa. La sortida ADC_CS_N va connectada a l'entrada d'habilitació del convertidor 128S022. La sortida ADC_SADDR va connectada a la sortida oDIN del mòdul ADC_CTRL i finalment la sortida ADC_SCLK va connectada a la sortida oSCLK i a l'entrada SCLK del convertidor.

Les variables internes d'aquet mòdul són la wSPI_CLK i la wSPI_CLK_n. Ambdues, són de tipus *wire*. La wSPI_CLK connecta la sortida c0 del mòdul SPIPLL amb l'entrada iCLK del mòdul ADC_CTRL i la wSPI_CLK_n connecta la sortida c1 del mòdul SPIPLL amb l'entrada iCLK_n del mòdul ADC_CTRL.

A la Figura 58 es pot observar el mòdul DE0_NANO, amb les seves variables internes i externes i els dos mòduls que el forment.

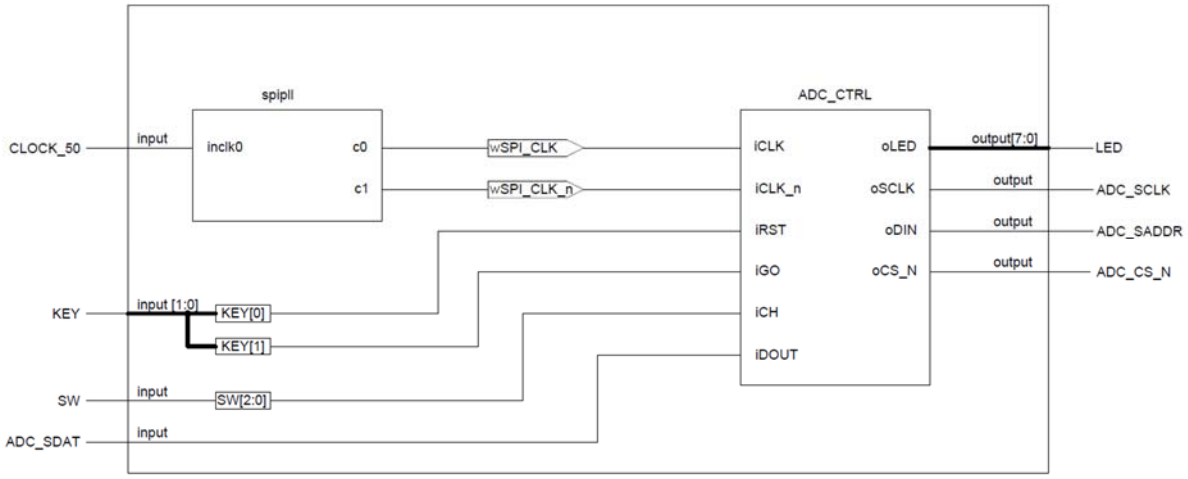


Figura 58. Estructura del mòdul DE0_NANO

7.2 Mòdul SPIPLL

El mòdul SPIPLL esta format per l'entrada inclk0 i les sortides C0 i C1. Aquest mòdul es crea utilitzant l'aplicació MegaWizard Plug-In Manager que incorpora el programa Quartus. De l'aplicació MegaWizard Plug-In Manager la megafunció PLL que s'utilitza és ALTPLL. Aquesta suporta cinc modes de treball: el normal, el síncron, el buffer de retard, el sense compensació i el de realimentació externa. Cada un d'aquest permet la multiplicació o divisió de la senyal de rellotge, desfasa la fase del rellotge i programa el cicle de treball. La configuració en aquest programa, tal com s'observa a la Figura 59, el mode de treball és normal, l'entrada inclk0 és la del rellotge de 50Mhz, la freqüència de les dues sortides és de 2 MHz , el cicle de treball de les dues sortides és del cinquanta per cent i el desfasament de les sortides és de 0 nanosegons per la sortida C0 i de 250 ns per la sortida C1.

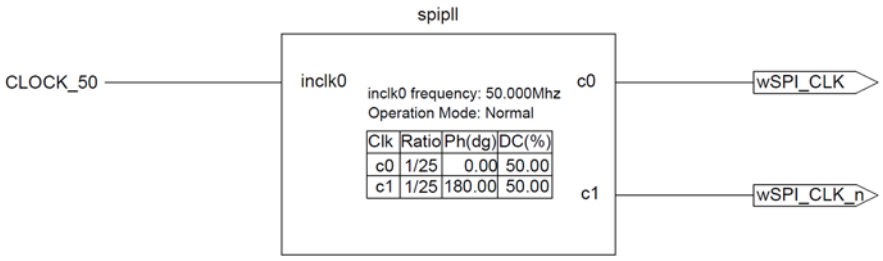


Figura 59. Mòdul spipll

La funció d'aquest mòdul es crear dues senyals de rellotge desfasades entre si. Aquestes dues senyals C0 i C1 fan funcionar el mòdul ADC_CTRL i el senyal de rellotge C1, a més a més serveix com a senyal SCLK entre el convertidor i la Cyclone IV. La raó que el senyal C1

vagi desfasada respecta C0 és per poder enviar les dades en els flancs de baixada, tal i com especifica el full de característiques del convertidor. A la Figura 60 podem observar les dues senyals de rellotge.

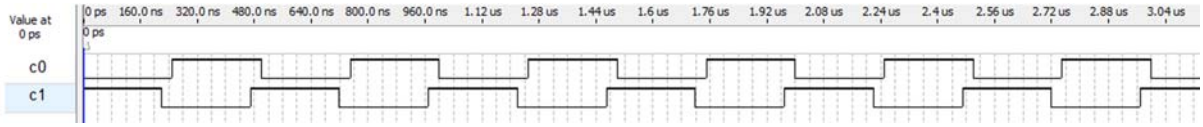


Figura 60. Sortides del mòdul SPIPLL

7.3 Mòdul ADC_CTRL

El mòdul està format per sis entrades, quatre sortides i vuit variables internes. Les entrades són la iRST, la iCLK, la iCLK_n, la iGO, la iCH i la iDOUT; i les sortides són la oLED, oDIN, oCS_n i la oSCLK. Les variables internes són la data, go_en, ch_sel, la sclk, la cont, la m_cont, la adc_data i la led. Totes les variables internes són de tipus *reg* excepte la ch_sel que és tipus *wire*.

La funció del mòdul és realitzar la comunicació SPI, seleccionar l'entrada que es vol llegir i il·luminar els LEDs corresponents. Aquí s'hi duen a terme cinc processos *always,* tots ells relacionats entre si. Sempre que es reinicia el mòdul totes les variables internes tenen un valor inicial 0.

La sortida oDIN correspon a la senyal MOSI d'una comunicació SPI de quatre senyals motiu pel qual se li assigna el valor de la variable *data* que és on s'especifica l'entrada que es vol seleccionar del convertidor. A la Figura 61 es pot observar l'estructura del mòdul ADC_CTRL.

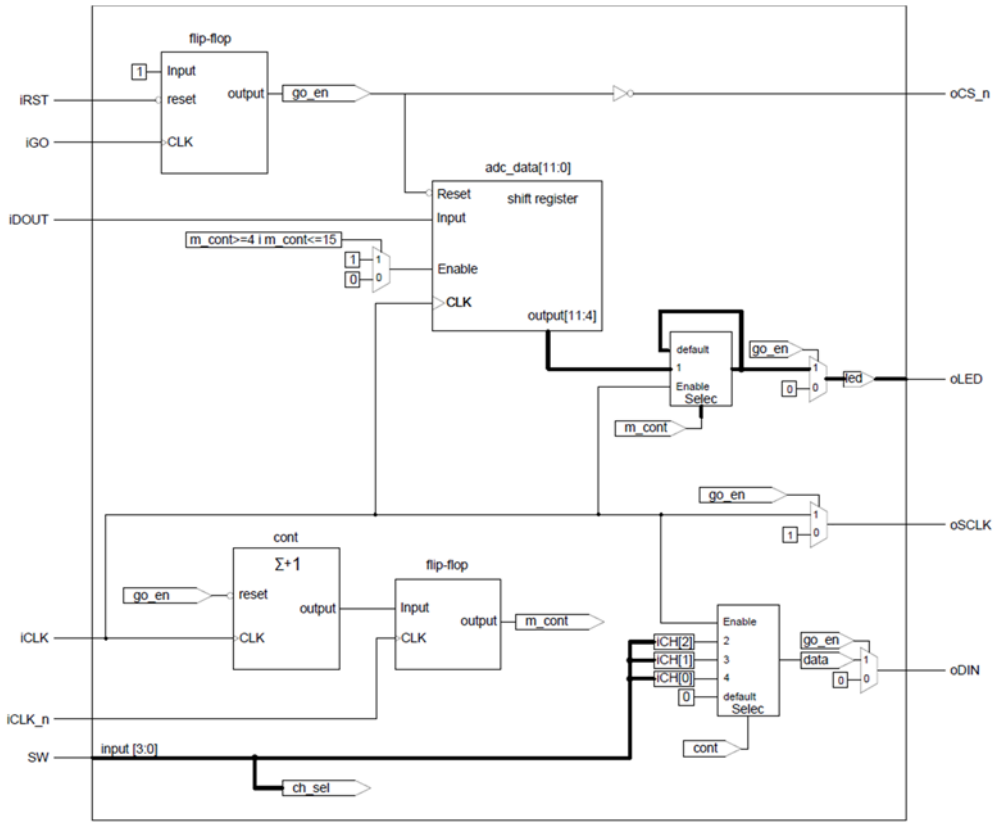


Figura 61. Estructura del mòdul ADC_CTRL

El primer procés *always* només serveix per assignar el valor 1 a la variable go_en quan la variable iGO val 1. Aquest fet passa quan es produeix un flanc de pujada provocat pel polsador KEY1. La representació d'aquest procés és un *flip flop* amb una entrada de valor 1. Aquest valor es transferirà a la variable go_en quan es produeixi un flanc ascendent a l'entrada de rellotge. En cas que s'activi l'entrada *reset* el valor de go_en és 0.

El segon procés *always* només s'utilitza per augmentar el valor de la variable cont en una unitat a cada flanc de rellotge iCLK sempre i quan, la variable go_en no tingui un valor 0. Aquest procés es representa com a comptador que té una entrada de *reset* activa a nivell baix, una entrada de rellotge i una sortida.

El tercer procés *always* assigna el valor de la variable cont a la variable m_cont, sempre que es produeix un flanc de pujada del rellotge iCLK_n. Aquest procés també es representa com un *flip flop* que sempre s'executa quan arriba un flanc de pujada de la variable iCLK_n.

El quart procés *always* té la funció de seleccionar l'entrada del convertidor que es vol llegir. Per a dur a terme aquesta tasca s'assigna el valor de iCH[2], iCH[1] i iCH[0] a la variable *data*

quan els valors de la variable *cont* valen 2,3 i 4 respectivament sempre i quan es produeixi un flanc ascendent del rellotge iCLK_n. Si es dóna el cas que la variable go_en val 0 o que la variable *cont* no val un dels valors comentats anteriorment el valor de la variable *data* és 0. Per representar aquest procés s'utilitza un multiplexor amb entrada d'habilitació.

El cinquè procés *always* assigna a la variable adc_data el valor de iDOUT. Aquesta variable conté el resultat de la conversió del convertidor i es representa en forma de registre de desplaçament. El senyal iDOUT correspon al senyal MISO d'una comunicació SPI de quatre senyals. Una vegada realitzada aquesta acció es transfereix el valor adc_data[11:4] a la variable *led* per tal de representar el valor que ha enviat el convertidor il·luminant els LEDs corresponents. De la variable adc_data només s'utilitzen els vuit bits més significatius perquè només es disposa de 8 LEDs.

Les sortides oCS_n, oSCLK, oDIN i oLED no es troben dins cap procés *always* sinó que el seu valor s'assigna mitjançant la funció *assign*.

Quan el valor de go_en sigui 0 el valor de la sortida oCS_n és 0 i per la oSCLK surt el senyal de rellotge iCLK. La funció de la sortida oCS_n és habilitar el convertidor 128S022 quan aquesta val 0.

El funcionament del mòdul és el següent: un cop es prem el polsador KEY1, s'envia el valor dels tres interruptors per la sortida oDIN. Aquest valor correspon a una entrada del convertidor analògic a digital. Seguidament es comença enviar el resultat de la conversió per l'entrada iDOUT i un cop enviat el resultat aquest s'assigna a la sortida oLED i s'il·luminen els LEDs corresponents. Finalitzat el procés es torna a començar de nou i així successivament fins que no es premi el polsador KEY0, que aquest apagarà tots els LEDs i el programa quedarà a l'espera fins que es torni a polsar el KEY1.

7.4 Proves, simulacions i resultats del programa DE0_NANO

El programa DE0_NANO que ens ofereix Altera disposa de moltes entrades i sortides que no s'utilitzen. Per aquest motiu s'han eliminat totes aquestes entrades i sortides perquè quedi clar com és el mòdul. En l'explicació del programa ja no s'han tingut en compte aquestes entrades i sortides.

També s'ha comprovat que al mòdul ADC_CTRL hi ha la variable *ch_sel*, la qual no realitza cap funció. Per aquest motiu s'ha eliminat i s'ha comprovat que el programa continua funcionant correctament. La funció que hauria de tenir la variable *ch_sel* és assignar a la variable *data* el valor de l'entrada iCH. Però com que és possible assignar el valor de iCH directament a *data*, a la pràctica, aquesta variable no fa servei.

Un vegada es polsa el polsador KEY1 s'inicia la comunicació SPI un cop finalitza torna a començar sense la necessitat de polsar KEY1. Per tant es pot canviar l'entrada del convertidor sense tenir que reiniciar el programa. Per comprovar que realment funciona s'han alimentat totes les entrades del convertidor i cada cert temps s'ha seleccionat una entrada diferent.

El motiu pel qual el valor de la variable iCH s'assigni a la variable *data* quan arriba un flanc de pujada de la iCLK_n és perquè aquest valor sorti per la sortida ADC_SADDR quan es produeixi un flanc de baixada del senyal de rellotge de la sortida ADC_SCLK.

El motiu pel qual les dades que envia el convertidor s'assignen a la variable adc_data quan es produeix un flanc de pujada de la iCLK és perquè aquestes tenen que ser llegides en el flanc de pujada de la variable ADC_SCLK.

Per un correcte funcionament del convertidor el rang de freqüències que permet l'entrada de rellotge SCLK és de 0,8 MHz a 3,2 MHz. EL senyal de rellotge en aquest programa és de 2 MHz per tant es troba dins els límits. Una prova que s'ha fet és reduir el temps a 1 MHz. Per portar-ho a terme només ha calgut modificar el valor *ratio* del mòdul SPIPLL. A la Figura 62 es mostra la seqüència de comunicació SPI de quatre senyals. Les dades que s'han d'escriure al convertidor s'envien en cada flanc de baixada i les que es llegeixen a la Cyclone IV s'envien en cada flanc de pujada. Els valors SW0, SW1 i SW2 contenen els valors dels interruptors.

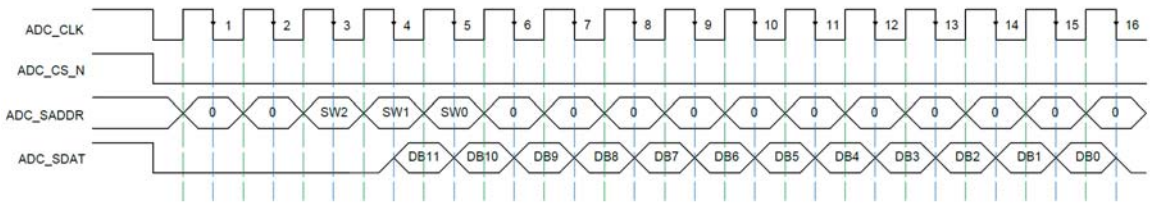


Figura 62. Comunicació SPI de quatre senyals

Un cop carregat el programa a la placa DE0-NANO se selecciona l'entrada 0 del convertidor i es polsa el polsador KEY1. Seguidament s'aplica un voltatge de 0 V a l'entrada, després un de 1,6 V i finalment un de 3,3 V. A la Figura 63 es mostren els resultats obtinguts.

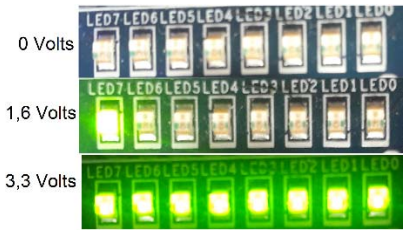


Figura 63. Resultats del programa DE0_NANO

8 PROGRAMA DE0_NANO_G_SENSOR

Per a la implementació de la comunicació SPI de tres senyals es fa servir el programa DE0_NANO_G_Sensor. Aquest programa utilitza els polsadors KEY0, els vuit LEDs i l'acceleròmetre ADXL345 de la placa. El programa permet que en funció de quin sigui el costat que s'inclini la placa respecte a l'eix X s'il·luminin els LEDs corresponents. Per reiniciar de nou el programa s'utilitza el polsador KEY0.

El programa es divideix en diferents mòduls que segueix una estructura jeràrquica. En el nivell més alt hi ha el fitxer DE0_NANO_G_Sensor. Dintre aquest i trobem el fitxer led_driver que controla els LEDs, el fitxer reset_delay que s'encarrega de reiniciar el programa, el fitxer spi_ee_config on hi ha la configuració de la comunicació SPI i el spipll que és una megafunció que controla el temps de tot el programa. Dins spi_ee_config hi ha el fitxer spi_controler que s'hi troba el comptador i el registre de desplaçament per a la comunicació SPI, entre d'altres. A la Figura 64 es pot observar l'estructura jeràrquica i del programa.

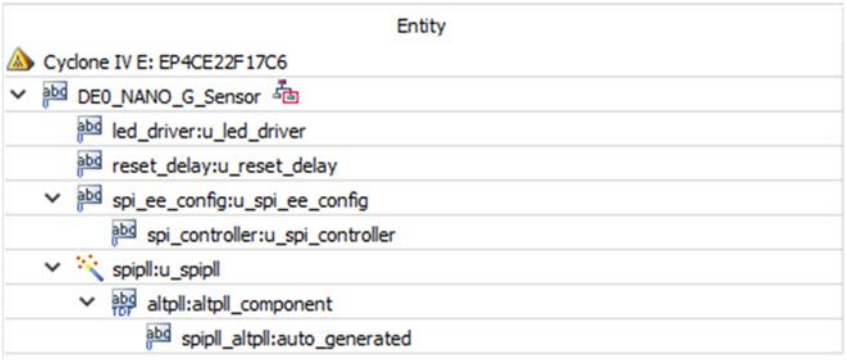


Figura 64. Estructura del programa i tots els fitxes que el formen

8.1 Configuració de l'acceleròmetre ADXL345

De les característiques i modalitats que l'acceleròmetre ofereix, només s'utilitzen les adreces que es troben en el fitxer spi_param.h. En aquest fitxer hi troben dos tipus d'adreces. Les que són configurables i les que són de lectura. En aquest apartat únicament s'anomenen les adreces que calen configurar perquè el programa funcioni correctament i de les adreces que només són de lectura es comentaran més endavant. A la Taula 11 es relaciona les adreces que es configuren amb el seu nom, el valor que tenen configurat de fàbrica i una descripció identificativa.

Adreça	Nom	Valor	Descripció
0x24	THRESH_ACT	00000000	Llindar d'activitat
0x25	TRESH_INACT	00000000	Llindar d'inactivitat
0x26	TIME_INACT	00000000	Temps d'inactivitat
0x27	ACTI_INACTI_CTL	00000000	Eixos habilitats per la detecció de activitat i inactivitat.
0x28	TRESH_FF	00000000	Llindar de caiguda lliure
0x29	TIME_FF	00000000	Temps de caiguda lliure
0x2C	BW_RATE	00001010	Velocitat de les dades i control del mode de potència
0x2D	POWER_CTL	00000000	Control de les funcions de l'estalvi energètic
0x2E	INT_ENABLE	00000000	Control d'habilitació de les interrupcions
0x2F	INT_MAP	00000000	Assignació de les interrupcions a una de les sortides
0x31	DATA_FORMAT	00000000	Format amb que s'envien les dades

Taula 11. Adreces de configuració de ADXL345

8.1.1 Configuració de l'adreça 0x24

L'adreça està formada per 8 bits, el valor dels quals determinen el valor llindar per a la detecció d'activitat. Quan es produeixi una interrupció d'activitat el valor d'aquesta interrupció es compara amb el valor llindar per determinar si hi ha o no activitat. El valor llindar d'activitat en aquest programa és de 2000 mg. Aquest valor es troba utilitzant l'Equació 1.

$$L_{activitat} = F \cdot X \qquad \text{(Eq. 1)}$$

On:

$L_{activitat}$ és el valor del llindar d'activitat.

F és el factor d'escala 62,5 mg/LSB.

X és el valor escollit per usuari.

8.1.2 Configuració de l'adreça 0x25

L'adreça està formada per 8 bits, el valor dels quals determinen el valor llindar per a la detecció d'inactivitat. Quan es produeixi una interrupció d'inactivitat el valor d'aquesta interrupció es compara amb el valor llindar per determinar si hi ha o no inactivitat. El valor llindar de inactivitat en aquest programa és de 187,5 mg. Aquest valor es troba utilitzant l'Equació 2.

$$L_{inactivitat} = F \cdot X \qquad \text{(Eq. 2)}$$

On:

$L_{inactivitat}$ és el valor del llindar de inactivitat.

F és el factor d'escala 62,5 mg/LSB.

X és el valor escollit per usuari.

8.1.3 Configuració de l'adreça 0x26

L'adreça està formada per 8 bits, el valor dels quals determinen el temps d'inactivitat el qual representa la quantitat de temps que l'acceleració és menor que el valor de l'adreça THRESH_INACT. A diferència d'altres funcions les dades són filtrades. El valor del temps d'inactivitat en aquest programa és d'1 segon. Aquest valor es troba utilitzant l'Equació 3.

$$T_{inactivitat} = F \cdot X \qquad \text{(Eq. 3)}$$

On:

$T_{inactivitat}$ és el valor del llindar del temps d'inactivitat.

F és el factor d'escala 1 segon/LSB.

X és el valor escollit per usuari.

8.1.4 Configuració de l'adreça 0x27

La adreça 0x27 serveix per habilitar els eixos amb que desitgem detectar activitat i si volem una configuració *dc-coupled* o *ac- coupled*. A la Figura 65 hi han els bits que formen aquesta adreça. El valor d'aquesta adreça en aquest programa és 01111111.

D7 ACT ac/dc	**D6** ACT_X enable	**D5** ACT_Y enable	**D4** ACT_Z enable
D3 INACT ac/dc	**D2** INACT_X enable	**D1** INACT_Y enable	**D0** INACT_Z enable

Figura 65. Adreça 0x27

Els bits D7 i D3 serveixen per escollir la configuració *dc-coupled* o *ac- coupled*. El bit D7 té un valor 0 i per tant queda seleccionada l'operació dc-acoblat, i el bit D3 té un valor 1 i per tant funciona amb *ac-acoblat*. En el funcionament dc-acoblat, la magnitud d'acceleració actual es compara directament amb THRESH_ACT i THRESH_INACT per determinar si es detecta activitat o inactivitat.

En el funcionament *ac-acoblat* per a la detecció de l'activitat, el valor d'acceleració en l'inici de la detecció d'activitat es pren com un valor de referència. Les noves mostres d'acceleració es comparen llavors amb aquest valor de referència, i si la magnitud de la diferència excedeix el valor THRESH_ACT, el dispositiu activa una interrupció de l'activitat.

De la mateixa manera, en el funcionament *ac-acoblat* per a la detecció d'inactivitat, un valor de referència s'utilitza per a la comparació i s'actualitza cada vegada que el dispositiu excedeix el llindar d'inactivitat. Després se selecciona el valor de referència, el dispositiu compara la magnitud de la diferència entre el valor de referència i l'acceleració actual amb THRESH_INACT. Si la diferència és menor que el valor en THRESH_INACT en el moment de TIME_INACT, el dispositiu es considera inactiu i la interrupció de la inactivitat s'activa.

Tots els altres bits serveixen per habilitar els eixos. Un valor 1 permet que l'eix participi en la detecció d'activitat o inactivitat i un valor 0 l'exclou. Amb aquest programa estan tots habilitats i per tant tenen un valor 1. Amb la detecció d'activitat tots els eixos es troben amb la funció lògica OR i amb la de inactivitat amb la funció lògica AND.

8.1.5 Configuració de l'adreça 0x28

En l'adreça 0x28 es defineix el valor llindar de la caiguda lliure. Les acceleracions de tots els eixos es comparen amb aquest valor per determinar si hi ha caiguda lliure o no. El valor del llindar de caiguda lliure en aquest programa és 562,5 mg. Aquest valor es troba utilitzant l'Equació 4.

$$A = F \cdot X \qquad \text{(Eq. 4)}$$

On:

A és el valor del llindar de caiguda lliure.

F és el factor d'escala 62,5 mg/LSB.

X és el valor escollit per usuari.

8.1.6 Configuració de l'adreça 0x29

L'adreça 0x29 serveix per determinar la quantitat de temps que els valors de les acceleracions de tots els eixos han de ser majors que el de l'adreça 0x28 per generar una caiguda lliure. El temps en aquest programa és de 0x46 en hexadecimal corresponent a 350 ms.

8.1.7 Configuració de l'adreça 0x2C

L'acceleròmetre canvia el seu consum d'energia en proporció a la velocitat de sortida de les dades de forma automàtica. Si a més a més es vol un estalvi d'energia addicional és possible activar el mode de baix consum. Amb aquest mode la freqüència interna de mostreig es redueix permeten un estalvi energètic en les freqüències compreses entre 12,5Hz i 400Hz, però comporta un augment del soroll. El registre 0x2C permet la configuració d'aquest paràmetres. Tal com mostra la Figura 66 aquest registre està format per 8 bits. Els 4 bits menys significatius serveixen per seleccionar l'ample de banda i la velocitat de sortida de les dades, amb el bit D4 podem escollir si volem o no el mode de baix consum i la resta de bits valen 0 perquè no s'utilitzen.

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	LOW_POWER	Rate			

Figura 66. Adreça 0x2C

El valor que se li assigna en aquest programa és de 00001010. Els quatre bits menys significatius estableixen un ample de banda de 25 Hz i una velocitat de sortida de 50 Hz i un consum de 90μA. Com que el bit D4 val 0 es treballa amb mode normal.

8.1.8 Configuració de l'adreça 0x2D

Addicionalment podem estalviar energia fent que l'acceleròmetre ADXL345 canviï automàticament a mode de repòs durant els períodes d'inactivitat. El dispositiu continuarà llegint dades però a una velocitat inferior. S'entrarà en el mode de repòs si la funció d'inactivitat està activada i si es detecta una acceleració per sota del valor THRESH_INACT durant els temps establert per TIME_INACT. Per tornar a funcionar amb normalitat i sortir del mode de repòs, la funció d'activitat ha d'estar habilitada i s'ha de detectar activitat. Si encara es vol reduir més el consum es pot utilitzar el mode d'espera, el qual, redueix el consum de corrent a 0,1 μA. Quan s'entra en aquest mode d'espera no es realitzen mesures. L'adreça 0x2D serveix per configurar el mode d'estalvi energètic. A la Figura 67 es pot observar que el registre està format per 8 bits. Els bits D0 i D1 serveixen per seleccionar la freqüència amb que es llegiran les dades en el mode de repòs. El bit D2, quan val 1, posa l'acceleròmetre en mode de repòs. El bit D3 activa el mode d'espera i el bit D4 activa el mode automàtic de repòs i per últim el bit D5, anomenat bit d'enllaç, quan el seu valor és 1, es redueixen el nombre

d'interrupcions d'activitat, fent que només es detecti activitat si abans hi ha hagut inactivitat i a la inversa. Quan aquest bit val 0 les funcions d'activitat i inactivitat són concurrents.

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

Figura 67. Adreça 0X2D

El valor que se li assigna en aquest programa és 00001000. Els bits D2 i D4 valen 0 per tant els mode de repòs i de repòs automàtic no estan activats, com a conseqüència els bits D1 i D0 que serveixen per escollir la velocitat de lectura en mode de respòs valen 0 perquè no s'utilitzen. El bit D3 val 1 per tant, el mode d'espera també esta desactivat i només treballa en mode de mesura. Per últim el bit D5 val 0 i les funcions d'activitat i inactivitat són concurrents.

8.1.9 Configuració de l'adreça 0x31

El format amb que s'envien les dades de cada un dels eixos que es troben en les adreces 0x32 a 0x37 es determina amb la configuració de l'adreça 0x31. Aquesta adreça està formada per 8 bits i el seu valor és 01000000, tal com es mostra a la Figura 68.

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

Figura 68. Format del registre 0x31

Els bits D0 i D1 serveixen per seleccionar el rang, valen 0 i per tant, estableixen un rang de mesura de +/-2g. El bit D2 determina com es troben posades les dades de les adreces 0X32 a 0x37 i el seu valor és 0, i en conseqüència el format és justificat a la dreta. El bit D3 determina la resolució. D'acord amb el programa aquest bit val 0 i per tant no hi ha resolució completa. El bit D4 no s'utilitza i el seu valor sempre és 0. El bit 5 serveix per el valor de les sortides d'interrupció INT0 i INT1. D'acord amb aquest programa el seu valor és 1, motiu pel qual, quan es produeixi una interrupció, el valor de sortida també serà 1. El bit D6 permet escollir el tipus de comunicació SPI que es vol utilitzar, el seu valor és 1 i per tant la configuració és SPI de tres senyals i per últim el bit D7 serveix per realitzar un auto test i en aquest cas com que el seu valor és 0 mai es realitzarà l'auto test.

8.1.10 Configuració de l'adreça 0x2E

L'ADXL345 proporciona els pins de sortida INT1 i INT2 per a les interrupcions. Ambdós pins són de baixa impedància. La configuració per defecte dels pins interrupció és activa a nivell alt. Es pot canviar la configuració a activa a nivell baix posant el bit INT_INVERT al registre DATA_FORMAT. Totes les funcions d'interrupció es poden utilitzar simultàniament, amb una única limitació, que és, que algunes de les funcions necessiten utilitzar els dos pins de sortida. Les interrupcions queden memoritzades i s'esborren quan es llegeixen les adreces 0x32 a 0x37 o bé, fins que la condició de interrupció deixi de ser valida. Les interrupcions s'habiliten a la adreça 0x2E, com es mostra a la Figura 69 aquesta adreça està formada per 8 bits els quals, a cada un correspon a una interrupció.

D7	D6	D5	D4	D3	D2	D1	D0
DATA_READY	SINGLE_TAP	DOUBLE_TAP	Activity	Inactivity	FREE_FALL	Watermark	Overrun

Figura 69. Adreça 0xE

El valor que se li dóna aquesta adreça és 00010000. L'única interrupció que actua és la d'activitat. Aquesta s'activa quan hi ha una acceleració major a la que hi ha en l'adreça THRESH_ACT en qualsevol dels eixos actius establerts en l'adreça 0x27.

8.1.11 Configuració de l'adreça 0x2F

Per assignar les interrupció a una de les sortides, INT1 o INT2, s'utilitza l'adreça 0x2F. Aquesta té el mateix format que l'adreça 0x2E tal com es mostra a la Figura 70. El valor que se li a assigna aquesta adreça és 00010000. Si el bit val 1 vol dir que s'assigna a la INT 2 i si val 0 a la INT1, com que la INT 1 a la placa DE0-Nano no va connectada en lloc només es pot fer servir la INT2.

D7	D6	D5	D4	D3	D2	D1	D0
DATA_READY	SINGLE_TAP	DOUBLE_TAP	Activity	Inactivity	FREE_FALL	Watermark	Overrun

Figura 70. Adreça 0x2F

8.2 Mòdul DE0_NANO_G_Sensor

El mòdul DE0_NANO_G_Sensor engloba tots els mòduls següents: el reset_dely, el spipll, el spi_ee_config i el led_driver. Les entrades d'aquest mòdul són: la KEY, la CLOCK_50, la

G_SENSOR_INT i la I2C_SDAT i les sortides són: la LED, la G_SENSOR_CS_N i la i2C_SCLK. Totes elles van a un pin de la Cyclone IV. A la Figura 71 es pot observar a quin pins estan assignades i si aquestes són una entrada o una sortida.

Node Name	Direction	Location	I/O Bank
CLOCK_50	Input	PIN_R8	3
G_SENSOR_CS_N	Output	PIN_G5	1
G_SENSOR_INT	Input	PIN_M2	2
I2C_SCLK	Output	PIN_F2	1
I2C_SDAT	Bidir	PIN_F1	1
KEY[1]	Input	PIN_E1	1
KEY[0]	Input	PIN_J15	5
LED[7]	Output	PIN_L3	2
LED[6]	Output	PIN_B1	1
LED[5]	Output	PIN_F3	1
LED[4]	Output	PIN_D1	1
LED[3]	Output	PIN_A11	7
LED[2]	Output	PIN_B13	7
LED[1]	Output	PIN_A13	7
LED[0]	Output	PIN_A15	7

Figura 71. Assignació de les entrades i sortides als pins de la Cyclone IV

El mòdul reset_dely serveix per reiniciar el programa, el mòdul spipll és l'encarregat de proporcionar dos senyals de rellotge per fer la comunicació SPI, el mòdul spi_ee_config és on hi han les configuracions de l'acceleròmetre i del protocol SPI i finalment el mòdul led_driver és el que controla els LEDs que s'han d'encendre o apagar.

L'entrada KEY està formada per un *bus* de dos bits, on a cada un li correspon un polsador. El *bus* passa per l'etiqueta KEY0, perquè només es fa servir el polsador 0, i va a l'entrada iRSTN del mòdul reset_delay. L'entrada CLOCK_50 és la del senyal de rellotge proporcionada pel cristall oscil·lador de 50Mhz i va connectada a les entrades iCLK dels mòduls reset_delay, spipll i led_driver. L'entrada G_SENSOR_INT prové del pin de interrupció de l'acceleròmetre i va connectada a les entrades iG_INT2 dels mòduls spi_ee_config i led_driver. Finalment l'entrada I2C_SDAT serveix com entrada i sortida depenent de si s'envien o es reben dades de l'acceleròmetre.

La sortida oDATA_L del mòdul led_driver va connectada a la sortida LED del mòdul DE0_NANO_G_Sensor i d'aquí va a als 8 LEDs de la placa.

La sortida G_SENSOR_CS_N va connectada a l'entrada d'habilitació de l'acceleròmetre i la sortida i2C_SCLK és per on surt el senyal de rellotge i va connectada a l'entrada SCL de l'acceleròmetre.

Les variables internes d'aquet mòdul són les que es mostren a la Taula 12. Totes són de tipus *wire*.

Nom	Descripció
dly_rst	Aquesta variable connecta la sortida oRST del mòdul rest_delay amb l'entrada areset del mòdul spipll i les entrades iRSTN dels mòduls spi_ee_config i led_driver.
spi_clk	Aquesta variable connecta la sortida c0 del mòdul spipll amb l'entrada iSPI_CLK del mòdul spi_ee_config.
spi_clk_out	Aquesta variable connecta la sortida c1 del mòdul spipll amb l'entrada iSPI_CLK_OUT del mòdul spi_ee_config.
data_x	Aquesta variable de 16 bits guarda el valor de les sortides oDATA_L i oDATA_H.

Taula 12. Variables internes del mòdul DE0_NANO_G_Sensor per connectar els diferents mòduls interns

La data_x és una variable de 16 bit. El seu valor depèn de dues sortides del mòdul spi_ee_config. Aquestes són la oDATA_L que omple els primer 8 bits i la oDATA_H que omple els 8 següents; a més a més, el valor dels bits D0 a D9 són enviats a l'entrada iDIG del mòdul led_driver. En aquests bits, com s'ha comentat anteriorment, hi ha el valor de l'eix X.

A la Figura 72 podem observar el mòdul DE0_NANO_G_Sensor, amb les seves variables internes i externes i els diferents mòduls que el formen.

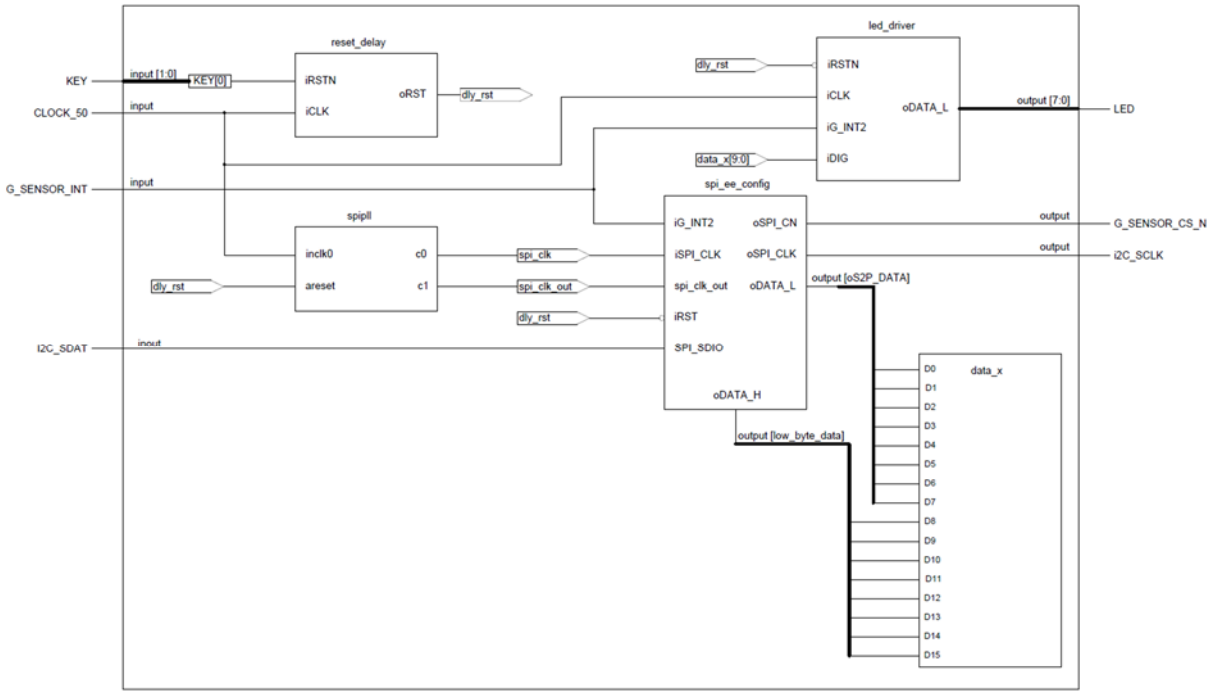


Figura 72. Estructura del mòdul DE0_NANO_G_Sensor

8.3 Mòdul reset_delay

El mòdul reset_delay tal com mostra la Figura 73 esta format per les variables externes iRSTN i iCLK que ambdues serveixen com entrades, per la sortida oRST de tipus *reg* i per una variable interna amb el nom cont, de tipus *reg* i formada per 21 bits. La funció de la variable interna és la d'un comptador ascendent, format per una entrada de senyal de rellotge, d'una entrada activa a nivell baix per reinicia el comptador i de les sortides D0 a D20 els quals corresponen a cada un dels bits del comptador.

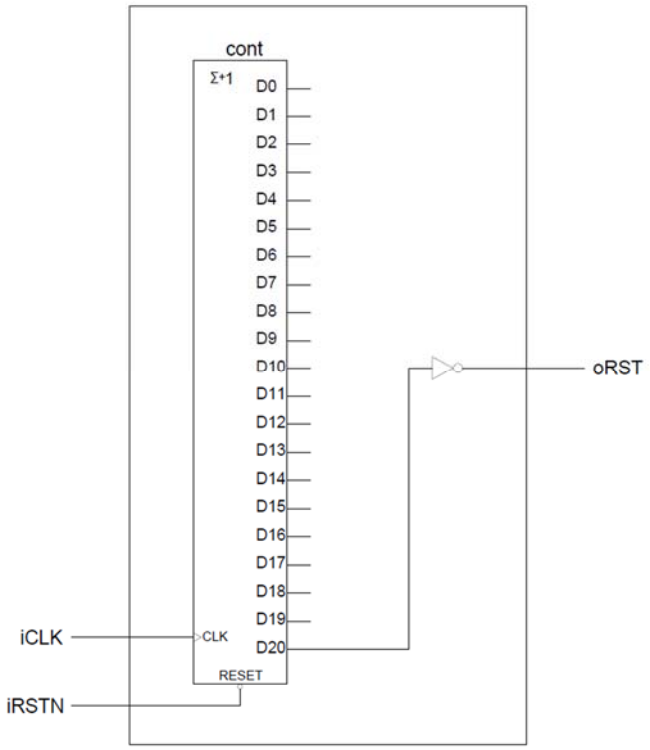


Figura 73. Estructura del mòdul reset_delay

La funció d'aquest mòdul es provocar un retard en el moment que comença el programa o sempre que es reinicií i serveix perquè el mòdul spill tingui temps de sincronitzar les seves sortides de rellotge abans d'enviar-les al mòdul spi_ee_config. En el mateix moment que se sincronitzen els rellotges del mòdul spill es reinicien els mòduls spi_ee_config i led_driver.

L'entrada iRSTN va connectada a l'entrada reset del comptador. El valor d'aquesta entrada depèn del polsador 0 el qual en estat de repòs és obert i el valor de l'entrada és 1, per tant el comptador augmentarà el seu valor en cada flanc de pujada de rellotge que arribi a l'entrada iCLK. Quan es prem el polsador es reinicia el comptador tornant al seu valor inicial. El valor

de la sortida oRST depèn del bit D20 del comptador, que quan el seu valor és 0 la sortida és 1 i a la inversa quan el valor sigui 1 la sortida és 0. Per altre banda, quan el bit D20 val 1 canvia, el valor del comptador és 10000000000000000000 en binari, que equival a 524.288 en decimal. Tal com es pot observar a la Figura 74 aquest valor sempre correspon a la meitat del període del temps de retard que es vol aconseguir.

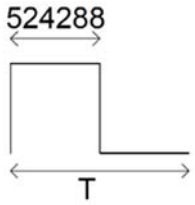


Figura 74. Període del temps de retard del mòdul reset_delay

Per quantificar el nombre de vegades de cicle del senyal de rellotge, utilitzem l'Equació 5 i obtenim un valor de 1.048.578.

$$C = \frac{N_{cicles}}{2} - 1 \qquad \text{(Eq. 5)}$$

On:

C és el valor de la meitat d'un cicle.

N_{cicles} és el número de vegades que s'ha de produir un cicle de rellotge.

Com que l'entrada de rellotge és de 50 Mhz el període d'un cicle de rellotge és de 20 ns. Aquest valor es troba utilitzant l'Equació 6.

$$T = \frac{1}{f} \qquad \text{(Eq. 6)}$$

On:

T és el període.

f és la freqüència.

Un cop trobats aquests dos valors determinem el temps de retard que provoca el mòdul utilitzant l'Equació 7, el qual s'obté un valor de 20,971 ms.

$$N_{cicles}=\frac{X}{T} \quad \text{(Eq. 7)}$$

On:

T és el període.

N_{cicles} és el nombre de vegades que s'ha de produir un cicle de rellotge.

X és el temps de retard.

8.4 Mòdul spipll

El mòdul spipll esta format per les entrades inclk0 i areset i les sortides C0 i C1. Aquest mòdul es crea utilitzant l'aplicació MegaWizard Plug-In Manager que incorpora el programa Quartus. De l'aplicació MegaWizard Plug-In Manager la megafunció PLL que s'utilitza és ALTPLL. Aquesta suporta cinc modes de treball: el normal, el síncron, el buffer de retard, el sense compensació i el de realimentació externa. Cada un d'aquest modes permet la multiplicació o divisió del senyal de rellotge, desfasa la fase del rellotge i programa el cicle de treball. La configuració en aquest programa, tal com s'observa a la Figura 75, és en mode de treball normal. L'entrada inclk0 és la del rellotge de 50Mhz, la freqüència de les dues sortides és de 2 MHz , el cicle de treball de les dues sortides és del cinquanta per cent i el desfasament de les sortides és de 200 ns a la sortida C0 i de 166,67 ns a la sortida C1.

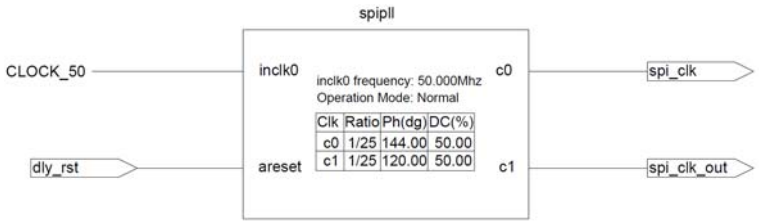


Figura 75. Mòdul spipll

La funció d'aquest mòdul es crear dues senyals de rellotge desfasades entre si. En els primers 20,97 ms del inici del programa tenen un valor 0 i un cop passat aquest temps s'activen els senyals de rellotge. El motiu d'això és que la l'entrada areset val 1 mentre el comptador del mòdul reset_dalay no arriba als 20,97 ms i canvia el valor a 0. El senyal de rellotge C0 fa funcionar tot el mòdul ee_spi_controler i el senyal de rellotge C1, tot i que entra al mòdul no l'afecta, únicament serveix com a senyal SCL entre l'acceleròmetre i la Cyclone IV. La raó que el senyal C0 vagi retardat respecta el C1 és perquè un cop es produeixi un flanc de pujada

s'envien dades entre l'acceleròmetre i la Cyclone IV i és convenient esperar un cert temps perquè aquestes s'estabilitzin abans de fer alguna acció amb elles.

Tot i que coneixem la configuració del mòdul s'ha cregut convenient comprovar el funcionament d'aquest per assegurar que es crea les senyals de rellotge que estableixen la comunicació SPI. Per aquest motiu es crea un nou projecte amb el nom DE0_Nano_PLL el qual només conté aquesta megafunció i complementat amb el University Program VWF es simula la megafunció per obtenir els resultats.

Quan el valor de l'entrada areset és 1 el valor de les sortides C0 i C1 és 0 tal com s'observa a la Figura 76 i si l'entrada areset val 0 a les sortides hi ha el corresponent senyal de rellotge, tal com s'observa a la Figura 77.

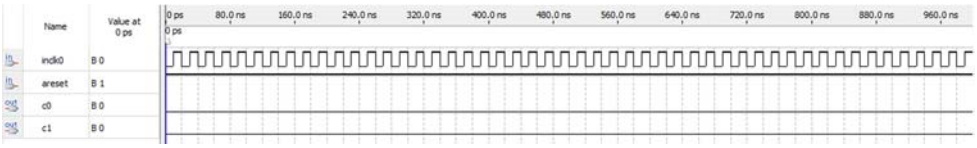


Figura 76. Resultat quan la senyal areset val 1

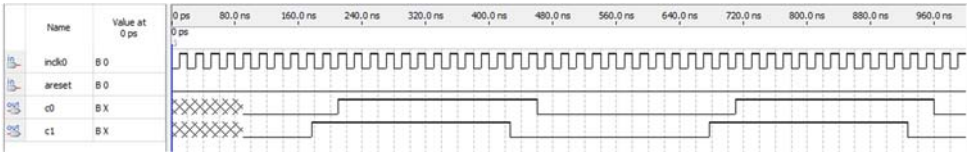


Figura 77. Resultat quan la senyal areset val 0

8.5 Mòdul spi_ee_config

El mòdul spi_ee_config està format per quatre entrades, quatre sortides, una entrada-sortida, quinze variables internes, un fitxer i un mòdul.

Les entrades són la iG_INT2, la iRSTN, la iSPI_CLK i la iSPI_CLK_OUT; les quatre sortides són la oSPI_CSN, la oDATAL, la oDATAH i la oSPI_CLK; i l'entrada-sortida és la SPI_SDIO.

Les variables internes del mòdul són la ini_index, la write_data, la p2s_data, la spi_go, la spi_end, la s2p_data, la low_byte_data, la spi_state, la read_back, la clear_status, la read_ready, la clear_status_d, la high_byte_d, la read_back_d i la read_idle_count.

El fitxer és el spi_para.h i el mòdul el spi_controller. A la Figura 78 s'observa la connexió d'entrades i sortides entre els dos mòduls i algunes variables internes.

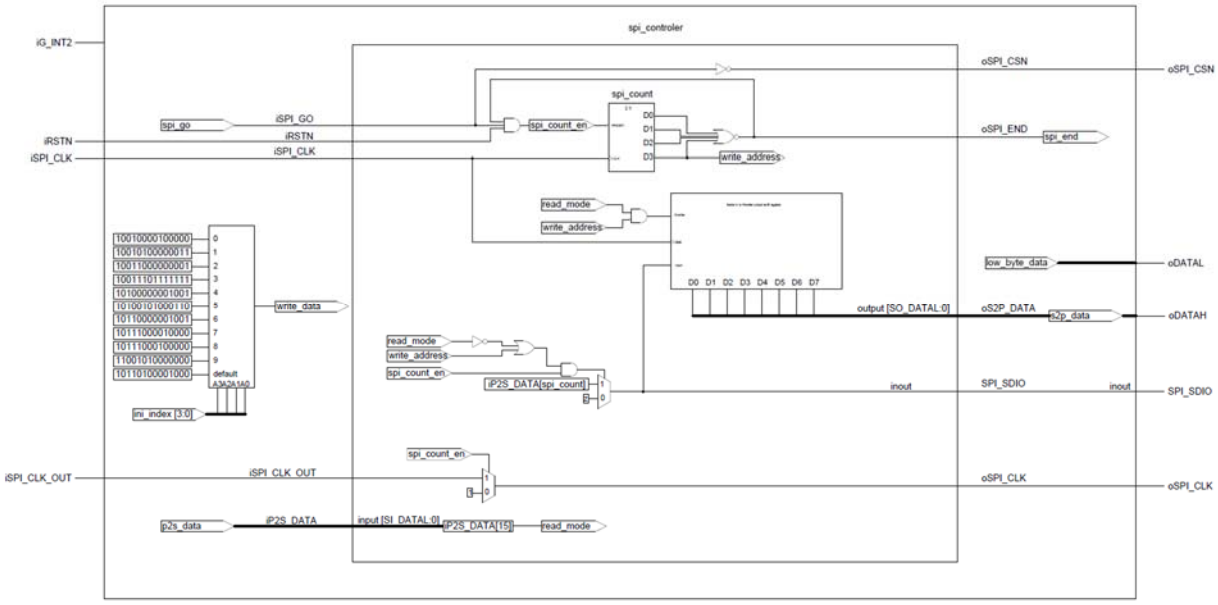


Figura 78. Estructura del mòdul spi_ee_config

Aquest mòdul té per funció realitzar la comunicació SPI, de configurar i llegir les dades de l'acceleròmetre i d'enviar les dades llegides al mòdul led_driver. Aquí s'hi duen a terme tres processos *always,* tots ells relacionats entre si, formant una màquina de *mealy*, on les accions passen en les transicions i no en els estats. Sempre que es reinicia el mòdul totes les variables internes tenen un valor inicial de 0 excepte, en la variable spi_state que val IDLE.

El primer procés *always* només conté una estructura *case* d'11 estats. Cada un d'ells uneix per concatenació una adreça de configuració de l'acceleròmetre del fitxer spi_param.h amb el valor que se li vol assignar i es transfereix a la variable write_data quan un dels estats és seleccionat. Per seleccionar cada un dels diferents casos s'utilitza la variable ini_index, aquesta és de tipus *reg* i està formada per 4 bits. Per representar aquesta estructura s'utilitza un multiplexor tal com s'observa a la Figura 79.

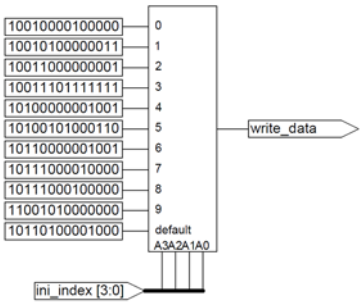


Figura 79. Representació del primer procés *always*

El segon procés *always* s'utilitza únicament per assignar valors a les variables high_byte_d, read_back_d i clear_status_d, en funció dels valors de high_byte, read_back i clear_status respectivament. Totes les variables són de tipus *reg* d'un bit, excepte la clear_status_d que està formada per 4 bits. A la variable celar_status_d se li assigna un bit a cada flanc de rellotge, i com que aquest és de quatre bits, el valor de la variable seran els 3 bits anteriors de més baix nivell i el nou bit. A la Figura 80 es pot observar la representació d'aquest procés.

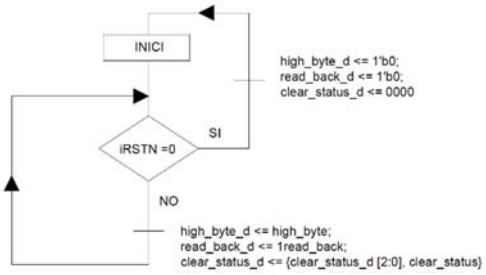
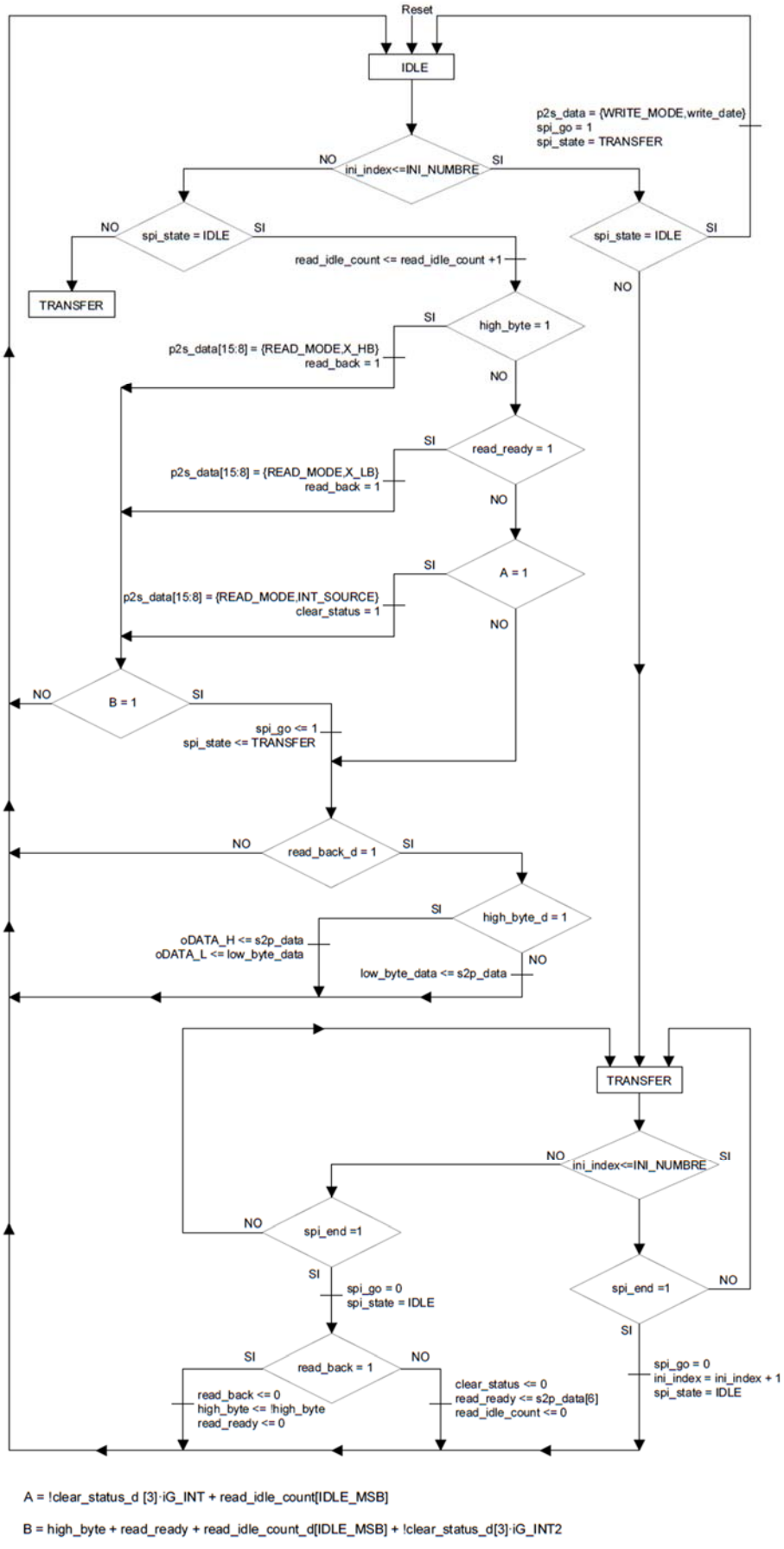


Figura 80. Diagrama de flux del segon proces always

És important destacar que quan el valor de les variables high_byte, read_back i clear_status canvien de valor com a conseqüència del tercer procés *always*, aquests valors no s'assignen immediatament a les variables high_byte_d, read_back_d i clear_status_d sinó, que serà en el següent flanc de rellotge. També cal dir que en el moment que es fa una assignació a una variable al mateix temps també s'avalua i el valor d'aquesta avaluació no és el que se li assigna sinó que és l'anterior ja que sempre hi ha un període de temps necessari perquè una variable canviï de valor.

Finalment, el tercer procés *always* està format per dos estats IDLE i TRANSFER. El primer serveix per gestionar les dades i el segon per transferir les dades. En funció del valor de les variables s'actuarà en un estat o en l'altre. El primer estat amb que ens trobem és l'estat IDLE, tal com s'observa a la Figura 81 sempre que es reinicïi el programa tornarem en aquest estat.



A = !clear_status_d [3]·iG_INT + read_idle_count[IDLE_MSB]

B = high_byte + read_ready + read_idle_count_d[IDLE_MSB] + !clear_status_d[3]·iG_INT2

Figura 81. Diagrama de flux del tercer procés *always*

El primer que fa el programa és comparar la variable ini_index amb la INI_NUMBER. Si ini_index és més petit que INI_NUMBER significa que encara falta enviar registres de configuració a l'acceleròmetre, motiu pel qual, acte seguit comprova si el valor de spi_state és IDLE. En cas afirmatiu s'assigna a la variable p2s_data el valor de la constant WRITE_MODE, seguit del valor de la variable write_data. El valor de la spi_go val 1, la spi_state és TRANSFER i es torna a iniciar el procés.

En aquest segon procés quan arriba a la spi_state al comparar que no és IDLE passa a l'estat TRANSFER.

Un cop a TRANSFER com que ini_index és més petit que INI_NUMBER es queda en aquest estat fins que el valor de la variable spi_end sigui 1 per tornar a l'estat IDLE. Perquè spi_end valgui 1 cal que el comptador del mòdul spi_controler haig arribat a 0 indicant que les dades han estat transferides. Quan passa això s'incrementa el valor de ini_index amb una unitat, spi_go se l'assigna el valor 0 i spi_state val IDLE permetent així el canvi d'estat.

De nou a l'estat IDEL és repeteix el procés, per poder enviat tots els registres de l'estructura case, fins que ini_index és més gran que INI_NUMBER.

Un cop ini_index és més gran que INI_NUMBER es deixa d'enviar configuracions a l'acceleròmetre i aquest queda en condicions d'enviar dades. Per a rebre dades primer cal enviar l'adreça que és vol llegir. Les tres adreces que s'utilitzen són 0x30, 0x32 i 0x33. L'adreça 0x30 amb el nom INT_SOURCE s'utilitza per saber si les dades estan preparades per ser llegides. Les adreces 0x32 i 0x33 amb el nom de X_LB i X_HB respectivament, contenen el valor de l'eix X.

Quan ini_index és més gran que INI_NUMBER es comprova si el valor de spi_state és IDLE. En cas afirmatiu s'augmenta el valor del comptador read_idle_count amb una unitat i seguidament es comprova si es compleix una de les tres condicions següents: primera, que el valor de la variable high_byte sigui 1; segona, que el valor de la read_ready sigui 1 i tercera, que el valor del quart bit de la variable clear_status_d sigui 0 i la entrada iG_INT2 valgui 1 o que el valor del 14 bit del comptador read_idle_count valgui 1.

El primer cop que es llegeixen dades és per que s'ha complert la tercera condició. El motiu és perquè el valor de les variables high_byte i read_ready no tindran mai el valor 1, fins que no es passi a l'estat de TRANSFER. Per altre banda el valor de clear_status_d és 0 i no canvia

per tant, només cal que esdevingui una interrupció perquè el valor iG_INT2 sigui 1 o que el bit 14 del comptador read_idle_count valgui 1. Aquest va augmentant cada cicle de rellotge. El comptador s'utilitza per assegurar que es llegiran les dades de l'acceleròmetre quan aquest no es troba en moviment.

Quan es produeix la tercera condició s'assigna a la variable p2s per concatenació el valor de la constant READ_MODE i de l'adreça INT_SOURCE i el valor de clear_status passa de 0 a 1. Un cop fetes aquestes assignacions si la read_mode, la high_byte, el bit 14 de read_idle_count o el resultat de multiplicar el bit 3 de clear_status_d amb la iG_INT2 és 1, llavors el valor de spi_go val 1 i el de spi_state és TRANSFER. A continuació com que el valor de read_back_d és 0 es torna a l'estat IDLE.

Ara a l'estat IDLE el valor de ini_index continua sent més gran que INI_NUMBER i el spi_state ja no val IDLE, per aquest motiu canvia a l'estat de TRANSFER. Ara de nou en aquest estat el procés continua a ini_index i com que aquest és més gran que INI_NUMBER, es passa a comprovar si el valor de la variable spi_end val 1. Si no es compleix aquest condició vol dir que el mòdul spi_controller encara no ha acabat de transmetre les dades a l'acceleròmetre i es retorna a l'estat de TRANSFER. Aquest procés es repeteix fins que spi_end valgui 1. Quan arribi aquest moment s'assignarà a la variable spi_go el valor 0, a la spi_state el de IDLE i es passa a comprovar el valor de read_back. Si aquest no val 1 s'assigna el valor 0 a la variable clear_status, el valor de s2p_data[6] a la read_redy, el valor 0 a la read_idle_count i aleshores es canvia a l'estat IDLE. És important remarcar que la variable read_ready indica que les dades de l'eix X estant a punt de ser llegides quan aquesta val 1. Per a cada flanc de rellotge el valor de s2p_data[6] canvia ja que depèn de les dades que s'envien en el registre de desplaçament del mòdul spi_config. Aquestes dades són de l'adreça INT_SOURE. A la Taula 13 es representen els 8 flanc on s'assigna el valor de la variable s2p_data en funció de les dades que rep.

Flanc	s2p_data [7]	s2p_data [6]	s2p_data [5]	s2p_data [4]	s2p_data [3]	s2p_data [2]	s2p_data [1]	s2p_data [0]
0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	0
5	0	0	1	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0

Taula 13. Valor de la variable s2p_data en cada flanc de pujada

El bit D7 de l'adreça INT_SOURCE és el que ens interessa perquè indica que les dades de l'eix X estan a punt de ser llegides. Per tal d'assignar aquest valor a la variable read_ready és imprescindible fer-ho quan el bit D7 es troba a la posició s2p_data[6], el motiu és perquè el desfasament entre els senyals de rellotge provoca que es compleixi la condició que spi_end valgui 1 abans que el bit D7 arribi a la posició s2p_data[7]. Quan això succeeix la variable spi_go val 0, la spi_state val IDLE i es passa a comprovar el valor de read_back. Com que el seu valor és 0 s'assigna a la variable clear_status el valor 0, a la read_ready el valor s2p_data[6], a la read_idle_count el valor 0 i es retorna a l'estat IDLE.

De nou a l'estat IDLE el valor de ini_index continua sent més gran que INI_NUMBER i el spi_state torna a tenir el valor de IDLE. Se suma una unitat al comptador i aquest cop a diferència d'abans es compleix la segona condició que read_ready és igual a 1. Al complir-se aquesta condició s'assigna a la variable p2s_data per concatenació el valor de la constant READ_MODE i el de l'adreça X_LB i també s'assigna a la variable read_back el valor 1.

I una vegada més es repeteix que si la read_mode, la high_byte, el bit 14 de read_idle_count o el resultat de multiplicar el bit 3 de clear_status_d amb la iG_INT2 és 1, llavors el valor de spi_go val 1 i el de spi_state és TRANSFER. A continuació com que el valor de read_back_d és 0 es torna a l'estat IDLE.

Com ha passat anteriorment el valor de ini_index continua sent més gran que INI_NUMBER i el spi_state ja no val IDLE, per aquest motiu canvia a l'estat de TRANSFER. Ara de nou en aquest estat el procés continua a ini_index i com que aquest és més gran que INI_NUMBER, es passa a comprovar si el valor de la variable spi_end val 1. Si no es compleix aquesta condició vol dir que el mòdul spi_controller encara no ha acabat de transmetre les dades a l'acceleròmetre. En el moment que spi_end val 1 el valor de spi_go passa a ser 0, el spi_state passa a ser IDLE. i es passa a comprovar el valor de read_back. Ara aquest valor és 1 i per tant el valor de la variable high_byte és 1, el de read_ready val 0 i el de read_back val 0. Es torna a l'estat inicial IDLE.

Un vegada a l'estat IDLE el valor de ini_index continua sent més gran que INI_NUMBER i el spi_state torna a tenir el valor de IDLE però ara a diferència de les altres vegades es compleix la primera condició, high_byte val 1. Per aquest motiu a p2s_data s'assigna per concatenació el valors de READ_MODE i de X_HB i a read_back el valor 1. I per tercera vegada es repeteix que si la read_mode, la high_byte, el bit 14 de read_idle_count o el resultat de multiplicar el bit 3 de clear_status_d amb la iG_INT2 és 1, llavors el valor de spi_go val 1 i el de spi_state

és TRANSFER. Després d'assignar aquests valors es compara el valor de read_back_d. Com que el seu valor és 1 es passa a comprovar el de la high_byte_d i com que aquest val 0 s'assigna a la variable low_byte_data el valor s2p_data.

De la mateixa manera que ha passat anteriorment quan spi_state val TRANSFER es va a aquest estat. Ara de nou el procés torna a continuar cap a la ini_index i com que aquest és més gran que INI_NUMBER, es passa a comprovar si el valor de la variable spi_end val 1. Si no es compleix aquest condició vol dir que el mòdul spi_controller encara no ha acabat de transmetre les dades a l'acceleròmetre. En el moment que spi_end val 1 el valor de spi_go passa a ser 0, el spi_state passa a ser IDLE i a continuació es comprova el valor de read_back que com ha passat anteriorment, el seu valor és 1. Com a conseqüència d'això el valor de la read_ready i el de la read_back valen 0 i el valor de la variable high_byte a diferència de l'altra vegada també és 0. Es torna a l'estat inicial IDLE.

Finalment a l'estat IDLE el valor de ini_index continua sent més gran que INI_NUMBER i el de spi_state és IDLE per això se suma una unitat al comptador read_idle_count. Continuant el procés com que no es compleix cap de les tres condicions exposades anteriorment, es va directament a comprovar si el valor de la read_back_d és 1. Si és així es comprova si el valor de high_byte és 1. Com que també val 1 s'assigna a oDATA_H el valor de s2p_data i a oDATA_L el valor de low_byte_data i es retorna a l'estat IDLE finalitzant així una transmissió de dades entre l'acceleròmetre i la Cyclone IV.

8.5.1 Mòdul spi_controler

El spi_controler és un mòdul format per cinc entrades, quatre sortides, una entrada-sortida, quatre variables internes i un fitxer.

Les entrades són les següents: la iSPI_GO, la iRSTN, la iSPI_CLK, la iSPI_CLK_OUT i la iP2S_DATA. Les quatre sortides són: la oSPI_CSN, la oSPI_END, la oS2P_DATA i la oSPI_CLK. L'entrada-sortida és la SPI_SDIO.

Les variables internes del mòdul són la read_mode i la write_address de tipus *wire*, la variable spi_count_en que és tipus *reg* i la spi_count que és tipus *reg* de quatre bits. La Figura 82 representa el mòdul spi_controler.

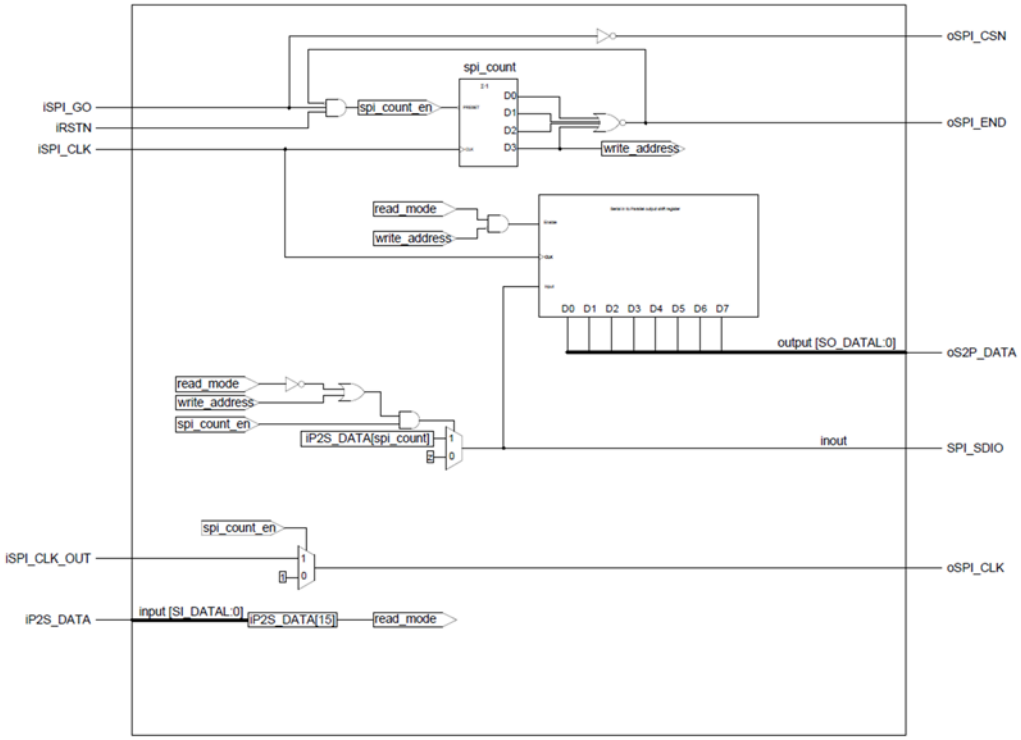


Figura 82. Estructura del mòdul spi_controler

L'entrada iSPI_GO s'utilitza per indicar que es vol començar una comunicació amb l'acceleròmetre. Per aquest motiu, el valor de l'entrada influeix en la variable spi_count, explicada més endavant, i el seu valor negat influeix a la sortida oSPI_CSN. Aquesta sortida serveix per habilitar l'acceleròmetre.

La iSPI_CLK és l'entrada de rellotge del mòdul i només influeix directament a les dues variables spi_count i oS2P_DATA perquè el seu valor canvia en cada flanc de pujada del rellotge. El valor de les variables read_mode, write_address, oSPI_END, oSPI_CSN, oSPI_CLK i SPI_SDIO és assignat mitjançant la funció *assign,* per tant no depenen del senyal de rellotge sinó que s'evaluen contínuament.

La variable spi_count és una variable interna tipus *reg* i recorda el seu últim valor. Aquesta fa la funció de comptador descendent, motiu pel qual, no es representa en forma d'etiqueta sinó en forma de bloc. Tal com s'observa a la Figura 83 té una entrada de senyal de rellotge, una entrada *preset* activa a nivell baix per reiniciar el comptador i quatre sortides, D0, D1, D2 i D3, corresponent als quatre bits del comptador.

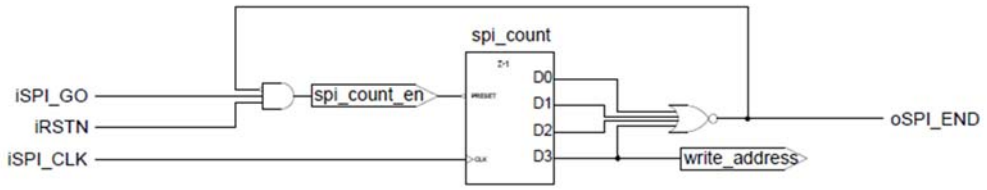


Figura 83. Esquema de la variable spi_count

La funció del comptador és transmetre els 16 bits que formen les trames de lectura i de escriptura. Els valors de les quatre sortides que té el comptador se sumem utilitzant una porta lògica NOR i el resultat va a la variable de sortida oSPI_END. Quan aquesta tingui un valor 1, indica que s'han transmès els 16 bits. El valor de la sortida D3 a més d'anar a la porta lògica NOR també va a la variable interna write_addres.

La variable spi_count_en va a l'entrada preset del comptador. Per aquest motiu és de tipus *reg* ja que ha de recordar el seu valor i reiniciar només en el cas que el comptador arribi a 0. El valor de spi_count_en depèn del resultat de multiplicar tres variables: la iSPI_GO, la iRSTN i la oSPI_END negada. La iSPI_GO depèn del mòdul spi_ee_config i la iRSTN del mòdul reset_delay que són mòduls externs. La oSPI_END negada depèn del valor del comptador que hi ha en aquest mòdul que sempre té un valor 1 fins que el comptador no arriba a 0.

Inicialment el comptador té un valor 16 i segueix en aquest valor sempre que una de les dues variables iSPI_GO o iRSTN tenen un valor 0, atès que el valor que se li assigna a la spi_count_en és també 0 i conseqüentment fa un preset del comptador amb independència del valor que pugui tenir la variable oSPI_END negada.

Si la iSPI_GO i la iRSTN valen 1 la spi_count_en també val 1 fent que quan arribi un flanc de pujada a l'entrada CLK el comptador descendeix una unitat i així successivament fins que les sortides D0 a D3 assoleixen un valor 0. Quan això passa el valor de la sortida oSPI_END val 1 indican que s'ha acabat de transmetre la trama de 16 bits i la variable spi_count_en té un valor 0 perquè la oSPI_END negada val 0.

Una altre funció de la variable spi_count_en és deixar passar el senyal de rellotge que ha d'anar a l'acceleròmetre. Aquesta senyal entra per l'entrada iSPI_CLK_OUT i surt per la sortida oSPI_CLK. Tal com s'observa a la Figura 84 el valor de la sortida oSPI_CLK depen de la variable spi_cout_en que actua com a selector d'un multiplexor i que només deixa passar el senyal de rellotge provinent de l'entrada iSPI_CLK_OUT quan el valor de la variable

spi_cout_en valgui 1. En cas contrari el valor de oSPI_CLK és 1 impedint que es pugui transmetre cap dada. Per tant es deixa passar el senyal de rellotge sempre que el comptador estigui funcionant, atès que és quan s'envien o es reben dades.

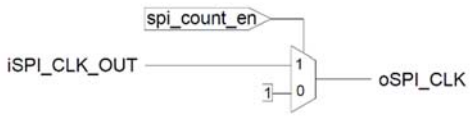


Figura 84. Multiplexor per seleccionar el valor de la sortida oSPI_CLK

Per últim la spi_count_en influeix alhora d'enviar i rebre dades. Tal com s'observa la Figura 85 si el seu valor és 0 la entrada-sortida SPI_SDIO val alta impedància, cosa que significa que encara que hi hagi senyal de rellotge no es transmet cap dada i deixa la línia lliure. La SPI_SDO s'utilitza com entrada quan es reben les dades de l'acceleròmetre i com a sortida quan es vol configurar l'acceleròmetre. El fet que la SPI_SDO actuï d'una manera o d'una altra depèn de les variables internes red_mode i write_addres. Quan la Cyclone IV rep les dades de l'acceleròmetre aquestes s'emmagatzemen a un registre de desplaçament de 8 bits, que té una entrada d'habilitació, una entrada de senyal de rellotge, una entrada per a les dades i vuit sortides que corresponent als 8 bits.

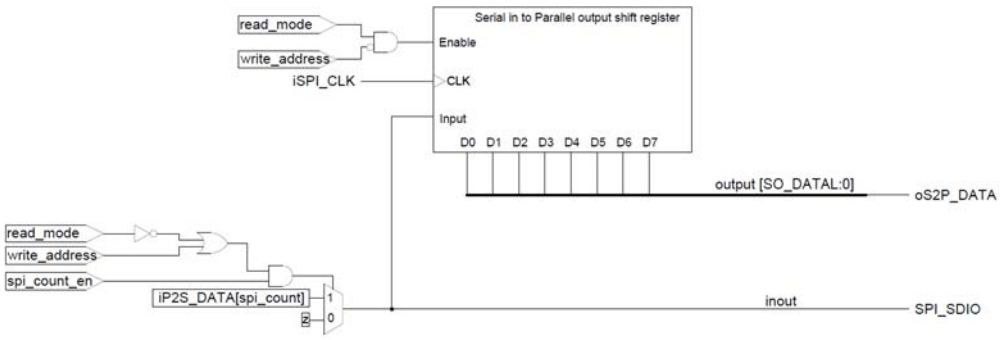


Figura 85. Estructura per enviar i rebre dades entre l'acceleròmetre i la Cyclone IV

El valor de la variable red_mode depèn del bit 15 del *bus* de dades el qual entra per l'entrada iP2S_DATA, tal com es pot observar a la Figura 86. Aquest bit correspon al R/W de la trama de bits que s'envien. Quan la red_mode val 1 significa que es volen rebre dades de l'acceleròmetre i quan val 0 que es volen enviar. El valor de la variable write_address depèn del bit D3 del comptador. En els 8 primers flancs de rellotge el bit D3 val 1 i en els 8 restants val 0.

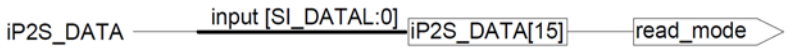


Figura 86. Assignació del valor a la variable read_mode

Aquest mòdul pot funcionar de dues maneres, una quan es vol configurar l'acceleròmetre i l'altra quan es vol llegir dades l'acceleròmetre.

Quan es configura l'acceleròmetre el comptador entra en funcionament al moment que la iSPI_GO val 1, al mateix temps a la variable read_mode se li assigna el valor iP2S_DATA que aquest és 0. Com a conseqüència, el registre de desplaçament es troba inhabilitat, la sortida oSPI_CSN val 0 habilitant l'acceleròmetre, a la sortida oSPI_CLK surt el senyal de rellotge i per la sortida SPI_SDIO surt bit a bit el valor de la variable iP2S_DATA en cada flanc de rellotge de l'entrada iPSI_CLK. En el moment que el comptador arriba a 0 les sortides oSPI_CSN i oSPI_CLK valen 1, la sortida SPI_SDIO val alta impedància i la oS2P_DATA val 0.

Quan es vol rebre dades de l'acceleròmetre el comptador entra en funcionament en el moment que la iSPI_GO val 1, al mateix temps a la variable read_mode se li assigna el valor iP2S_DATA que aquest és 1. El procés es divideix en 16 flancs de rellotge. En els 8 primers el registre de desplaçament es troba inhabilitat, a la sortida oSPI_CLK surt el senyal de rellotge, la sortida oSPI_CSN val 0 i per la entrada-sortida SPI_SDIO surt bit a bit el valor de la variable iP2S_DATA en cada flanc de rellotge. Després d'aquest 8 flancs el registre de desplaçament està habilitat i guarda les dades que entren per l'entrada-sortida SPI_SDIO i la resta de sortides no canvien. Un cop el comptador arriba a 0 les sortides oSPI_CSN i oSPI_CLK valen 1, la sortida SPI_SDIO val alta impedància i la oS2P_DATA val el valor que ha enviat l'acceleròmetre.

8.6 Mòdul led_driver

La funció d'aquest mòdul és il·luminar els LEDs de la placa DE0_Nano i això serà així en concordança amb les dades que rep del mòdul spi_ee_config. Aquest mòdul tal com es mostra a la Figura 87 està format per les quatre entrades iDIG, iG_INT2, iCLK i iRSTN, la sortida LED i les sis variables internes select_data, signed_bit, abs_select_high, int2_d, int_count, int2_count_en.

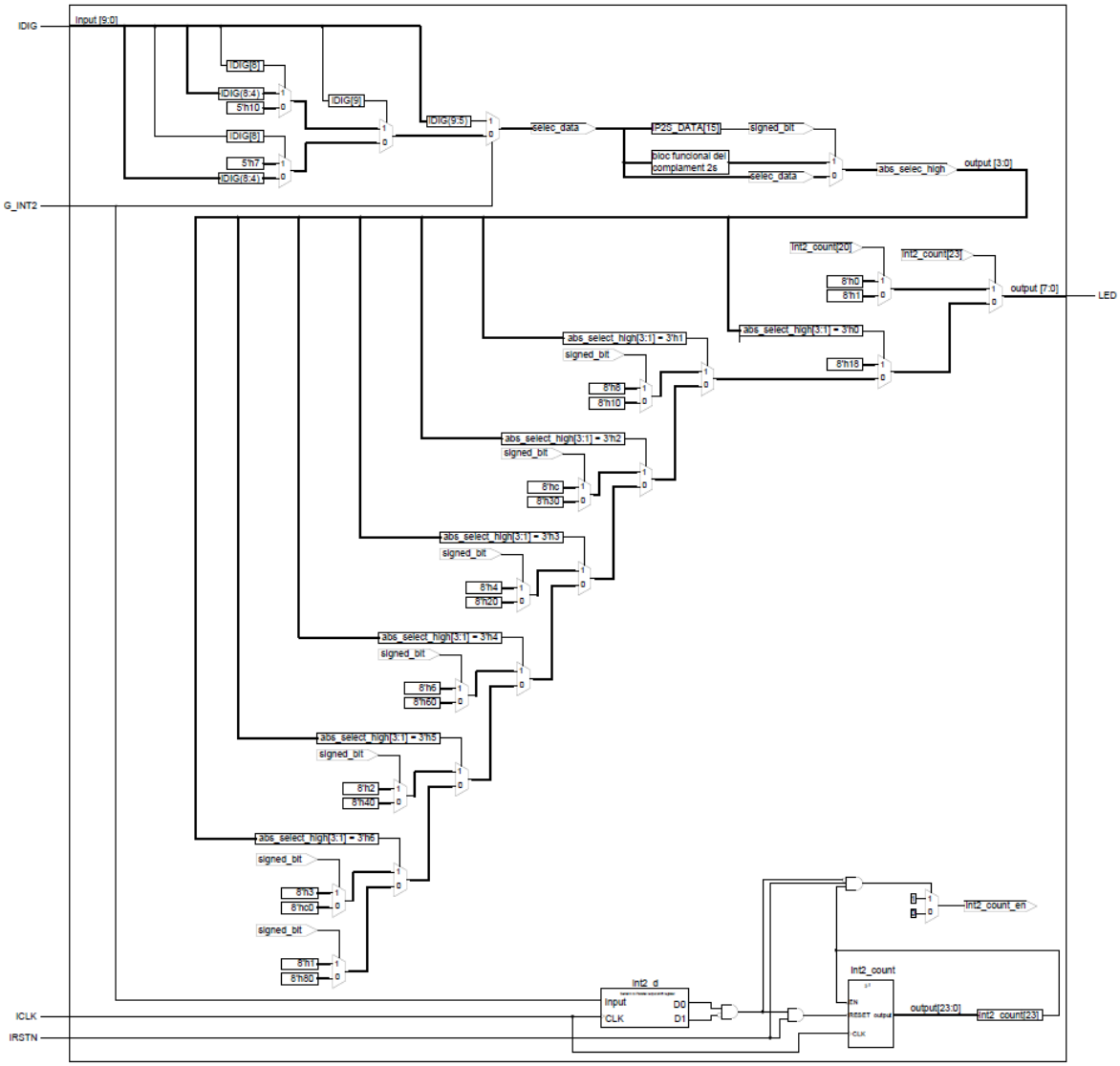


Figura 87. Estructura del mòdul led_driver

Les dades entren per la iDIG i s'assignen a la variable select_data en funció del valor de iG_INT2 .Tal com s'observa a la Figura 88 si hi ha una interrupció el valor de iG_INT2 val 1 per tant, el valor de select_data correspon al 5 bits més alts de l'entrada iDIG. Si no hi ha interrupció el valor de iG__INT2 val 0 i com a conseqüència el valor de select_data depèn dels bits iDIG[8] i iDIG [9]. Si iDIG [9] i iDIG[8] valen tots dos 0 o 1 s'assigna el valor iDIG[8:4]. Si iDIG [9] val 0 i iDIG [8] val 1 s'assigna el valor de 5'h7 i si iDIG[9] val 1 i iDIG[8] val 0 s'assigna el valor de 5'h10.

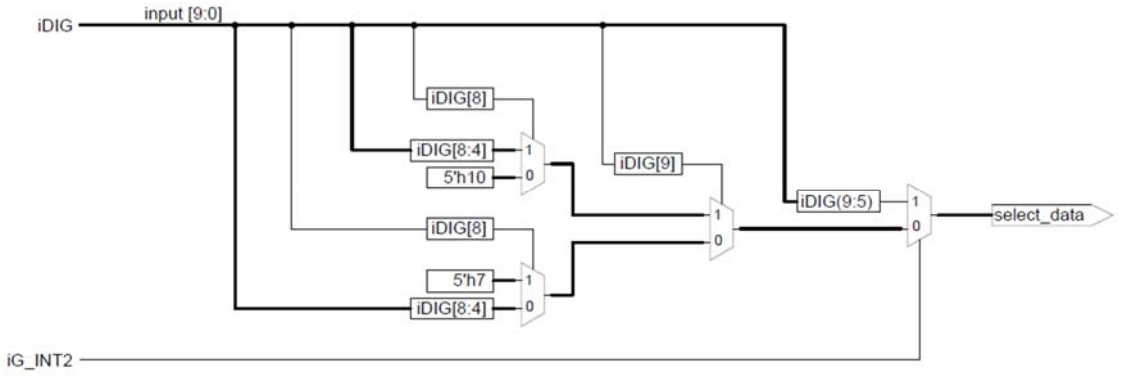


Figura 88. Assignació del valor selec_data

Un cop assignat el valor de select_data d'aquest s'agafa el bit de més pes i aquest s'assigna a la variable signed_bit. Això es fa per poder determinar si el número és positiu o negatiu. En el cas que sigui positiu la variable signed_bit val 1 i si és negatiu val 0. Si el signed_bit és positiu a la variable abs_select_high se li assigna el valor de select_data directament, en cas contrari, es realitza el negat de tots el bits de select_data abans d'assignar el valor a la variable abs_selct_high. A la Figura 89 es pot observar aquest procés.

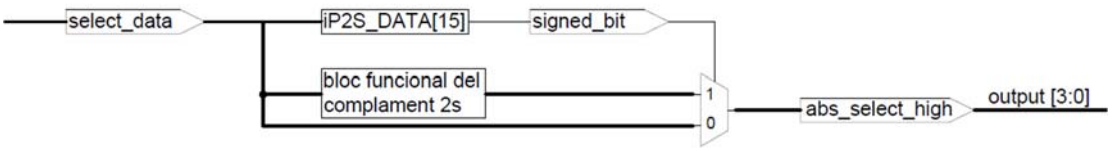


Figura 89. Assignació del valor de abs_select_high

El valor de la sortida oLED depèn del valor del bit 23 de la variable int2_count. Aquest bit té el valor de 0 o 1. Sempre que s'inicia o reinicia el programa el valor de la variable int2_count és 2'h800000 assignant al bit 23 el valor 1 i sempre tindrà aquest valor a no ser què el valor de la variable int2_d valgui 2'b01. Això es produeix quan en un període de rellotge no hi ha interrupció i en el següent si.

En el cas que el bit 23 de la variable int2_count val 0, es compara el bit 20 de la mateixa variable amb 0 i 1. Si val 0 s'il·luminen tots els LEDs en cas contrari s'apaguen tots.

En el cas que el bit 23 valgui 1 es compara el valor de abs_select_high amb diferents constants definides en el programa per determinar quins són els LEDs que s'han de il·luminar. Tal com es pot veure a la Taula 14 hi han 15 opcions possibles. La número 1 es dóna quan la placa

esta totalment plana. De la número 2 a la 8 és quan hi ha una inclinació negativa respecta l'eix X, a inclinació màxima s'il·lumina el LED0. De la número 9 a la 15 és quan hi ha una inclinació positiva respecta l'eix X, a inclinació màxima s'il·lumina el LED7.

Número	LED 7	LED 6	LED 5	LED 4	LED 3	LED 2	LED 1	LED 0
1	0	0	0	1	1	0	0	0
2	0	0	0	0	1	0	0	0
3	0	0	0	0	1	1	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	1	0
6	0	0	0	0	0	0	1	0
7	0	0	0	0	0	0	1	1
8	0	0	0	0	0	0	0	1
9	0	0	0	1	0	0	0	0
10	0	0	1	1	0	0	0	0
11	0	0	1	0	0	0	0	0
12	0	1	1	0	0	0	0	0
13	0	1	0	0	0	0	0	0
14	1	1	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0

Taula 14. Il·luminació dels LEDs en funció de la inclinació

8.7 Proves, simulacions i resultats del programa DE0_NANO_G_Sensor

Un cop carregat el programa a la placa DE0-Nano s'ha comprovat el seu funcionament. A la Figura 90, 91 i 92 es veu el cas número 1, el 5 i el 14 respectivament de la Taula comentada anteriorment.

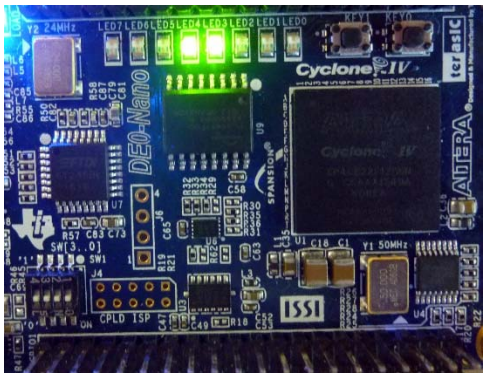


Figura 90. Cas número 1

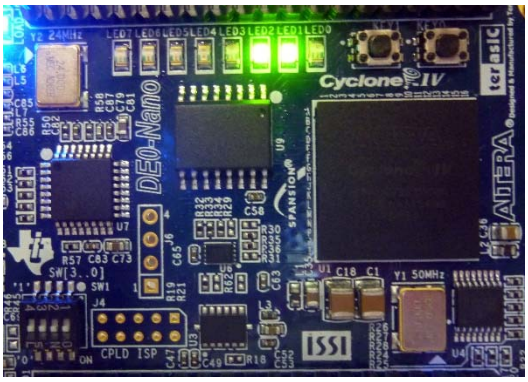


Figura 91. Cas número 5

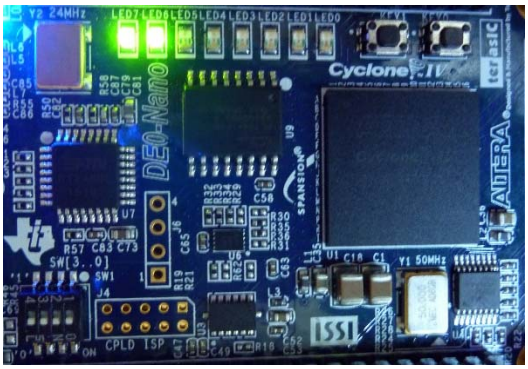


Figura 92. Cas número 14

Per comprovar el funcionament del programa respecte l'eix Y i l'eix Z ha estat necessari canviar la part del codi que envia l'adreça de lectura dels eixos. Els resultats respecta l'eix Y, són que quan la placa està totalment anivellada horitzontalment s'il·luminen els dos LEDs del mig. Si s'inclina cap a un costat o cap a l'altre respecta aquest eix, s'il·luminen els LEDs corresponents tal com passava quan es feia respecta l'eix X. Els resultats amb l'eix Z són que amb la placa totalment anivellada horitzontalment s'il·lumina el LED7, perquè s'il·luminin els LEDs centrals cal posar la placa en posició vertical i finalment per que s'il·lumini el LED0 cal girar la placa totalment de cap per avall.

Com que les sortides oDATA_L i oDATA_H són tipus reg, llavors l'entrada iDIG del mòdul led_dirver sempre té l'ultim valor que s'ha enviat i el manté fins que arriba un nou valor. Com a conseqüència, una vegada s'ha iniciat el program si es manté polsat el posador KEY0 es mantindran il·luminats sempre els mateixos LEDs, encara que s'inclini la placa en un dels dos costats.

La sortida oLED del mòdul led_driver té un valor que fa que tots els LEDs estiguin il·luminats. Per arribar en aquesta situació cal que els bits 23 i 20 de la variable int2_count tinguin un valor 0. S'ha comprovat que possant la placa de diferents posicions no s'ha esdevingut mai aquesta situació. El motiu és que el bit 23 de la variable int2_count sempre val 1 encara que es faci un reset del programa.

9 RESUM DEL PRESSUPOST

El pressupost d'aquest projecte descriu els materials necessaris per programar i utilitzar la placa DE0-Nano. A més a més inclou la instal·lació del programa Quartus II, de les aplicacions Control Panel i System Builder i del programes. Tot ells es comprovaran que funcionin correctament. Els components escollits per poder utilitzar els programes i les aplicacions de la placa DE0-Nano, s'ha tingut en compte que siguin econòmics, sense deixar de banda la qualitat dels mateixos.

El cost total per portar a terme aquest projecte és de nou cents quaranta-tres euros amb trenta-cinc cèntims, sense IVA.

10 CONCLUSIONS

Finalitzat aquest projecte es pot concloure que s'han assolit els objectius proposats inicialment. S'han estudiat i implementat les comunicacions sèrie I2C, SPI de tres senyals i SPI de quatre senyals utilitzant el llenguatge Verilog i la placa DE0-Nano. Tot això s'ha fet mitjançant els programes que s'han desenvolupat en aquest projecte, els quals, s'han realitzat esquemes i blocs per representar-los i compendre millor el seu funcionament. També s'ha realitzat una introducció al llenguatge Verilog que serveix com a guia d'aprenentatge d'aquest. En aquesta introducció s'han fet exemples amb aquest llenguatge mitjançant el programa Quartus II que justifiquen que funciona correctament. Finalment s'han explicat dues maneres diferents de carregar un programa a la Cyclone IV.

Amb la realització d'aquest projecte he ampliat els meus coneixements en el funcionament del programa Quartus II i de les comunicacions sèrie. A més a més he aprés un nou llenguatge de programació, el Verilog. La meva opinió en vers aquest llenguatge és que la seva utilització és fàcil i intuïtiva.

Comentar que el kit facilitat per l'Escola Politècnica Superior de la Universitat de Girona inclou l'aplicació Control Panel versió 1.2.0 però aquesta versió no és operativa. Per aquest motiu ha estat necessari descarregar la versió 1.3.0 de la pàgina web oficial de Terasic. Aquest kit també inclou la versió 13.0 gratuïta del programa Quartus II fet que limita la seva aplicació.

En el programa DE0_NANO_EEPROM es volia utilitza l'eina Single Tap II Logic Analizer per comprovar que la comunicació sèrie I2C entre la memòria EEPROM i la Cyclone IV es realitzava correctament. Per fer-ho s'havia d'executar el programa amb aquesta eina i visualitzar el valor assignat a la adreça amb l'aplicació Control Panel per veure si s'havia executat correctament. Malgrat tot, no va ser possible perquè aquesta no està disponible en la versió gratuïta del Quartus II. Per poder visualitzar el valor assignat a l'adreça a través del Control Panel es va carregar el programa al EPCS64.

Un altre aspecte que el diferencia del altres programes és que per crear el senyal de rellotge no s'utilitza la megafunció ALTPLL sinó que utilitza un comptador i només esta format per un sol mòdul.

Es pot afirmar que el programa DE0_NANO_EEPROM_V2 és millor que el programa original ja que el llenguatge Verilog es basa en mòduls i tal com s'ha vist amb els altres programes la

generació del senyal de rellotge amb la megafunció ALTPLL és més recomanable que no pas fer-ho amb una variable.

Una dificultat que va aparèixer quan feia l'estudi del programa DE0_NANO_G_Sensor és que el programa utilitza els noms i2C_SCLK i I2C_SDAT, fet que comporta a pensar si s'està fent una comunicació I2C o bé una SPI de tres senyals. Com que en cap moment s'envia l'adreça corresponent a l'acceleròmetre es va descartar que es tractés d'una comunicació I2C. També cal fer constar que de totes les configuracions i modes de funcionament de l'acceleròmetre només se'n utilitzen unes quantes.

Les modificacions que s'han realitzat per comprovar el funcionament del programa respecte l'eix Y i l'eix Z s'ha pogut comprovar que quan es llegeixen les dades en relació a l'eix Y els resultats són els mateixos que en l'eix X. En canvi amb l'eix Z al ser un eix amb posició perpendicular als altres dos eixos, perquè si il·luminin alguns LEDs és necessari posar la placa de cap per avall.

El fet que la sortida INT2 de l'acceleròmetre no estigui connectada a cap pin de la FPGA condiciona que totes les interrupcions tenen que passar per la sortida INT1 i en conseqüència limita l'ús de les interrupcions ja que si s'utilitzen totes, aquestes no poden sortir a la vegada per la sortida INT1. Per tant podria ser que una interrupció estigui esperant per ser llegida i deixi de ser vàlida i no sigui enviada.

Després d'estudiar les tres comunicacions es pot afirmar que les comunicacions sèrie que s'utilitzen en el convertidor analògic a digital i la memòria EEPROM són més fàcils de implementar que la de l'acceleròmetre. El motiu és que a diferència d'aquest el convertidor i la memòria no els hi fa falta configurar el seu mode de funcionament, ja que en el convertidor només cal definir l'entrada que es vol llegir i a la memòria l'adreça que es vol utilitzar, ja sigui per llegir o per a escriure.

Amb tot el que s'ha estudiat de les comunicacions sèrie I2C, SPI de tres senyals i SPI de quatre senyals es pot concloure que el temps en que s'executen les accions és molt important. En els dos tipus de comunicació SPI s'observa que el temps en que s'executa el programa i el que s'utilitza per realitzar les comunicacions està clarament diferenciat.

Finalment es pot concloure que els tres programes estudiats es poden utilitzar per a futures aplicacions. Per exemple el programa DE0_NANO_G_Sensor es pot modificar perquè en lloc

de encendre els LEDs de la placa faci girar un ventilador cap a la dreta o cap a l'esquerra i més o menys ràpid en funció de la inclinació. Això es pot portar a terme si es conserven els mòduls reset_delay, spipll, spi_ee_config i spi_controller i es modifica el mòdul led_driver perquè les sortides siguin per un pin del connector JP2 o JP3, a més a més, dels components externs a la placa.

El programa DE0_NANO també pot servir com a base per a diferents aplicacions ja que avui en dia s'utilizen diferents sensor que capten dades de l'entorn que ens envolta, com poden ser de temperatura, de pressió, d'intensistat de llum, de velocitat, etc. Aquests sensors converteixen les varibles físiques en elèctriques. Un exemple d'aplicacio del programa podria ser que en funció de la llum solar s'il·luminessin amb més o menys intensitat els llums a l'entrada i a la sortida d'un túnel de carretera. Tanmateix es pot utilitzar el programa DE0_NANO_EEPROM con a base per fer un programa que guardi les dades captades i que serveixin com a registre de les hores de sol.

En resum i com avaluació general de la placa es pot dir que permet fer una gran quantitat d'aplicacions. Tal com es subministra de fàbrica no disposa dels connectors J5, J1, J2 i J6, si l'usurari ho desitja els pots posar ampliant les sortides de voltatge.

Naïm Baulenas Sanglas

Graduat en Enginyeria Electrònica Industrial i Automàtica

Manlleu, 4 de setembre de 2017

11 RELACIÓ DE DOCUMENTS

Els documents que formen aquest projecte són els següents: memòria, plec de condicions, estat d'amidament i pressupost.

12 BIBLIOGRAFIA

Altera, ALTPLL IP Core User Guide. (https://www.altera.com/en_US/pdfs/literature/ug/ug_altpll.pdf, 15 de juny de 2017).

Altera, Cyclone IV Development & Education Board,DE0-Nano. (https://wiki.ntb.ch/infoportal/_media/fpga/boards/ de0_nano/de0-nano-schematic.pdf, 3 de juliol de 2017).

Altera, Cyclone IV FPGA. (https://www.altera.com/en_US/pdfs/literature/hb/cyclone-iv/cyiv-51001.pdf, 20 de maig de 2017).

Altera, manual DE0-Nano. (https://www.altera.com/en_US/pdfs/literature/ug/DE0_Nano_User_Manual_v1.9.pdf, 22 de juny de 2017).

Altera, Quartus II. (https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/intro_to_quartus2.pdf, 15 de juny de 2017).

Analog devices, ADXL345. (http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf, 20 de maig de 2017).

Evaluix. (http://www.eveliux.com/mx/Transmision-Serial-y-Paralelo.html, 20 de juny de 2017).

I2C, I2C bus. (http://www.i2c-bus.org/clock-stretching/, 20 de maig de 2017).

Microchip, basic serial EEPROM operation. (http://ww1.microchip.com/downloads/en/AppNotes/00536.pdf, 20 de maig de 2017).

Microchip. memòria 24LC02B. (http://www.microchip.com/downloads/en/DeviceDoc/21709c.pdf, 14 de juny de 2017).

Sparkfun, I2C. (https://learn.sparkfun.com/tutorials/i2c, 15 de juny de 2017).

Sparkfun, SPI. (https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi, 15 de juny de 2017).

Terasic, DE0-Nano. (http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=593, 15 de juny de 2017).

Texas instruments, ADC128S022. (http://www.ti.com/lit/ds/symlink/adc128s022.pdf, 21 de juny de 2017).

Verilog, Verilog.renerta. (http://verilog.renerta.com/source/vrg00019.htm, 21 de juny de 2017).

Wikipedia, comunicació sèrie. (https://en.wikipedia.org/wiki/Serial_communication, 21 de juny de 2017).

13 GLOSSARI

ACK: Acknowledged

ASCI: Circuits integrats per aplicacions especifiques

ASCII: American Standard Code for Information Interchange

CAN: Controller Area Network

CPHA: Clock Phase

CPLD: Complex Programmable Logic Device

CPOL: Clock Polarity

CS: Chip Select

DE: Development and Education

DIN: Digital Input

DOUT: Digital Output

DSP: Digital Signal Processor

EEPROM: Electrically Erasable Programmable Read-Only Memory

FIFO: First In, First Out

FPGA: Filed Programmable Gate Array

GPIO: General Purpose Input Output

HDL: Hardware Design Language

LED: Light Emitting Diode

MISO: Master In Slave Out

MOSI: Master Out Slave In

NES: Nintendo Entertainment System

QSPI: Queued Serial Peripheral Interface

RAM: Random Access Memory

ROM: Read-Only Memory

RS232: Recommended Standard 232

RTL: Register Transfer Level

SCI: Serial Communications Interface

SCL: Serial Clock

SCLK: Serial Clock

SDA: Slave Data

SDIO: Slave Data Input Output

SPI: Serial Peripheral Interface

SS: Slave Select

UART: Universal Asynchronous Receiver Transmitter

USB: Universal Serial Bus

VCC: Voltatge de Corrent Continua

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuit

A PROGRAMES

En aquest apartat es recullen tots els programes que s'han utilitzat en aquest projecte.

A.1 DE0_NANO_EEPROM

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module DE0_NANO_EEPROM(

      ///////////// CLOCK /////////
      CLOCK_50,

      ///////////// LED /////////
      LED,

      ///////////// KEY /////////
      KEY,

      ///////////// SW /////////
      SW,

      ///////////// EEPROM /////////

      I2C_SCLK,
      I2C_SDAT,

      ///////////// TEST /////////
      COUNT,
      SD_COUNTER
);

//=======================================================
//  Variables d'entrada i sortida
//=======================================================
///////////// CLOCK /////////
input                           CLOCK_50;
///////////// LED /////////
output                [7:0]     LED;
///////////// KEY /////////
input                 [1:0]     KEY;
///////////// SW /////////
input                 [3:0]     SW;
////////EPROM /////////
output                          I2C_SCLK;
inout                           I2C_SDAT;

///////////// TEST /////////
output                      [5:0] SD_COUNTER;
```

```
output                        [9:0] COUNT;

//=======================================================
//  Variables internes REG/WIRE
//=======================================================
wire              reset_n;
reg               final;
wire  [3:0] valor;

reg         I2C_READ_END;
reg         GO;
reg   [6:0] SD_COUNTER;
reg         SDI;
reg         SCLK;
reg      [9:0] COUNT;

//=======================================================
//  Structural coding
//=======================================================
assign reset_n = KEY[0]; //Per reiniciar el programe
assign I2C_SCLK = ((SD_COUNTER >= 4)&(SD_COUNTER <=31))? ~COUNT[9]: SCLK;
//Sortida del senyal SCLK
assign I2C_SDAT = SDI;//Entrada-Sortida de les dades.
assign LED[0] =final; // Indica que s'han enviar les dades a la memoria
assign valor=SW [3:0]; // Per seleccionar el valor que es vol enviar.

//Primera estructura always
always@(posedge CLOCK_50)

COUNT<= COUNT +1;
//Segona estructura always
always@ (posedge COUNT[9] or negedge reset_n)
begin
   if (!reset_n)
      GO <= 0;

      else
      if(!KEY[1])
          GO <= 1;

end


//Tercera estructura always, I2C COUNTER
always@(posedge COUNT[9] or negedge reset_n)
begin
   if (!reset_n)
          SD_COUNTER <= 6'b0;
   else
      begin
         if (!GO)
               SD_COUNTER <= 0;
           else
           if (SD_COUNTER<33)
```

```
                SD_COUNTER <= SD_COUNTER+1;
   end
end

//Quarta estructura always, I2C Operation

always @(posedge COUNT[9] or negedge reset_n)
begin
    if (!reset_n)
       begin
        SCLK <=1;
        SDI <=1;

        end
        else

 case (SD_COUNTER)

           6'd0 : begin SDI <=1; SCLK<=1; final=1'b0; end

            //Condició d'inici
            6'd1 : SDI <=0;
            6'd2 : SCLK <=0;

            //Adreça de l'esclau
            6'd3  : SDI <=1;
            6'd4  : SDI <=0;
            6'd5  : SDI <=1;
            6'd6  : SDI <=0;
            6'd7  : SDI <=0;
            6'd8  : SDI <=0;
            6'd9  : SDI <=0;
            6'd10 : SDI <=0;
            6'd11 : SDI <=1'bz; //Slave ACK

            //Adreça
            6'd12  : SDI <=0;
            6'd13  : SDI <=0;
            6'd14  : SDI <=0;
            6'd15  : SDI <=0;
            6'd16  : SDI <=0;
            6'd17  : SDI <=0;
            6'd18  : SDI <=0;
            6'd19  : SDI <=0;
            6'd20  : SDI <=1'bz; //Slave ACK


            //DATA
       6'd21  : SDI <=0;
            6'd22  : SDI <=0;
         6'd23  : SDI <=0;
         6'd24  : SDI <=0;
         6'd25  : SDI <=valor[3];
         6'd26  : SDI <=valor[2];
```

```
          6'd27  : SDI <=valor[1];
          6'd28  : SDI <=valor[0];
          6'd29  : SDI <=1'b1;

             //Condició de stop
       6'd30 : begin SDI <= 1'b0; SCLK <= 1'b1;final=1'b1;end
             6'd31 : SDI <= 1'b1;
      endcase
end
endmodule
```

A.2 DE0_NANO_EEPROM_V2

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================
module DE0_NANO_EEPROM_V2(
      ///////////// CLOCK /////////
      input                               CLOCK_50,
      ///////////// LED /////////
      output                [7:0]         LED,
      ///////////// KEY /////////
      input                 [1:0]         KEY,
      ///////////// SW /////////
      input                 [3:0]         SW,
      ///////////// EEPROM /////////
      output                              I2C_SCLK,
      inout                               I2C_SDAT
);

//=======================================================
//  Variables internes tipus REG/WIRE
//=======================================================
wire  CLK;
//=======================================================
// Definició dels submoduls
//=======================================================
i2cpll   U0 ( .inclk0 (CLOCK_50),.c0(CLK));

EEPROM_CONTROL U1 (
                .iRST (KEY[0]),
                              .iCLK(CLK),
                              .oLED(LED),
                              .iGO (KEY[1]),
                              .iSW(SW),

                              .oI2C_SCLK(I2C_SCLK),
                              .iI2C_SDAT(I2C_SDAT)

);
endmodule
```

A.2.1 I2cpll

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll
// ============================================================
// File Name: i2cpll.v
// Megafunction Name(s):
//               altpll
// Simulation Library Files(s):
//               altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// ************************************************************
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module i2cpll (
      inclk0,
      c0);

      input   inclk0;
      output        c0;

      wire [4:0] sub_wire0;
      wire [0:0] sub_wire4 = 1'h0;
      wire [0:0] sub_wire1 = sub_wire0[0:0];
      wire  c0 = sub_wire1;
      wire  sub_wire2 = inclk0;
      wire [1:0] sub_wire3 = {sub_wire4, sub_wire2};

      altpll      altpll_component (
                        .inclk (sub_wire3),
                        .clk (sub_wire0),
                        .activeclock (),
                        .areset (1'b0),
                        .clkbad (),
                        .clkena ({6{1'b1}}),
                        .clkloss (),
                        .clkswitch (1'b0),
                        .configupdate (1'b0),
                        .enable0 (),
                        .enable1 (),
                        .extclk (),
                        .extclkena ({4{1'b1}}),
                        .fbin (1'b1),
                        .fbmimicbidir (),
                        .fbout (),
                        .fref (),
                        .icdrclk (),
```

```
                        .locked (),
                        .pfdena (1'b1),
                        .phasecounterselect ({4{1'b1}}),
                        .phasedone (),
                        .phasestep (1'b1),
                        .phaseupdown (1'b1),
                        .pllena (1'b1),
                        .scanaclr (1'b0),
                        .scanclk (1'b0),
                        .scanclkena (1'b1),
                        .scandata (1'b0),
                        .scandataout (),
                        .scandone (),
                        .scanread (1'b0),
                        .scanwrite (1'b0),
                        .sclkout0 (),
                        .sclkout1 (),
                        .vcooverrange (),
                        .vcounderrange ());
defparam
      altpll_component.bandwidth_type = "AUTO",
      altpll_component.clk0_divide_by = 512,
      altpll_component.clk0_duty_cycle = 50,
      altpll_component.clk0_multiply_by = 1,
      altpll_component.clk0_phase_shift = "0",
      altpll_component.compensate_clock = "CLK0",
      altpll_component.inclk0_input_frequency = 20000,
      altpll_component.intended_device_family = "Cyclone IV E",
      altpll_component.lpm_hint = "CBX_MODULE_PREFIX=i2cpll",
      altpll_component.lpm_type = "altpll",
      altpll_component.operation_mode = "NORMAL",
      altpll_component.pll_type = "AUTO",
      altpll_component.port_activeclock = "PORT_UNUSED",
      altpll_component.port_areset = "PORT_UNUSED",
      altpll_component.port_clkbad0 = "PORT_UNUSED",
      altpll_component.port_clkbad1 = "PORT_UNUSED",
      altpll_component.port_clkloss = "PORT_UNUSED",
      altpll_component.port_clkswitch = "PORT_UNUSED",
      altpll_component.port_configupdate = "PORT_UNUSED",
      altpll_component.port_fbin = "PORT_UNUSED",
      altpll_component.port_inclk0 = "PORT_USED",
      altpll_component.port_inclk1 = "PORT_UNUSED",
      altpll_component.port_locked = "PORT_UNUSED",
      altpll_component.port_pfdena = "PORT_UNUSED",
      altpll_component.port_phasecounterselect = "PORT_UNUSED",
      altpll_component.port_phasedone = "PORT_UNUSED",
      altpll_component.port_phasestep = "PORT_UNUSED",
      altpll_component.port_phaseupdown = "PORT_UNUSED",
      altpll_component.port_pllena = "PORT_UNUSED",
      altpll_component.port_scanaclr = "PORT_UNUSED",
      altpll_component.port_scanclk = "PORT_UNUSED",
      altpll_component.port_scanclkena = "PORT_UNUSED",
      altpll_component.port_scandata = "PORT_UNUSED",
      altpll_component.port_scandataout = "PORT_UNUSED",
```

```
            altpll_component.port_scandone = "PORT_UNUSED",
            altpll_component.port_scanread = "PORT_UNUSED",
            altpll_component.port_scanwrite = "PORT_UNUSED",
            altpll_component.port_clk0 = "PORT_USED",
            altpll_component.port_clk1 = "PORT_UNUSED",
            altpll_component.port_clk2 = "PORT_UNUSED",
            altpll_component.port_clk3 = "PORT_UNUSED",
            altpll_component.port_clk4 = "PORT_UNUSED",
            altpll_component.port_clk5 = "PORT_UNUSED",
            altpll_component.port_clkena0 = "PORT_UNUSED",
            altpll_component.port_clkena1 = "PORT_UNUSED",
            altpll_component.port_clkena2 = "PORT_UNUSED",
            altpll_component.port_clkena3 = "PORT_UNUSED",
            altpll_component.port_clkena4 = "PORT_UNUSED",
            altpll_component.port_clkena5 = "PORT_UNUSED",
            altpll_component.port_extclk0 = "PORT_UNUSED",
            altpll_component.port_extclk1 = "PORT_UNUSED",
            altpll_component.port_extclk2 = "PORT_UNUSED",
            altpll_component.port_extclk3 = "PORT_UNUSED",
            altpll_component.width_clock = 5;
endmodule
```

A.2.2 EEPROM_CONTROL

```
module EEPROM_CONTROL (iRST , iCLK, oLED, oI2C_SCLK, iI2C_SDAT, iGO, iSW);

input iRST;
input iCLK;
input iGO;
output [7:0]oLED;
output oI2C_SCLK;
inout iI2C_SDAT;
input [3:0]iSW;

///Variables internes////
reg       go;
reg [6:0] SD_COUNTER;
reg       SDI;
reg       SCLK;
reg       final;
wire [3:0] valor;

///Assignacions///////////////

assign oI2C_SCLK = ((SD_COUNTER >= 4)&(SD_COUNTER <=58))? ~iCLK: SCLK;
//Sortida del senyal SCLK
assign iI2C_SDAT = SDI;
assign oLED[0]= final;

assign valor=iSW[3:0];

//Primera estructura always
//Condició perque comenci la comunicacio I2C
```

```
always@ (posedge iCLK or negedge iRST)
begin
   if (!iRST)
      go <= 0;

      else
      if(!iGO)
          go <= 1;

end

//Segona estructura always, I2C COUNTER
always@ (posedge iCLK or negedge iRST)
begin
   if (!iRST)
         SD_COUNTER <= 6'b0;
   else
      begin
         if (!go)
               SD_COUNTER <= 0;
            else
               if (SD_COUNTER < 60)
                     SD_COUNTER <= SD_COUNTER+1;
   end
end

//Tercera estructura always, I2C Operation

always@ (posedge iCLK or negedge iRST)
begin
     if (!iRST)
       begin
        SCLK <=1;
        SDI <=1;

        end
        else

  case (SD_COUNTER)

            6'd0 : begin SDI <=1; SCLK<=1; final<=1'b0;end

            //Condició d'inici
            6'd1 : SDI <=0;
            6'd2 : SCLK <=0;

            //Adreça de l'escalu
            6'd3  : SDI <=1;
            6'd4  : SDI <=0;
            6'd5  : SDI <=1;
            6'd6  : SDI <=0;
            6'd7  : SDI <=0;
            6'd8  : SDI <=0;
            6'd9  : SDI <=0;
```

```
      6'd10 : SDI <=0; // 0 per escriure i 1 per llegir
      6'd11 : SDI <=1'bz; //Slave ACK
      //SUB ADDR (n)
      6'd12  : SDI <=0;
      6'd13  : SDI <=0;
      6'd14  : SDI <=0;
      6'd15  : SDI <=0;
      6'd16  : SDI <=0;
      6'd17  : SDI <=0;
      6'd18  : SDI <=1;
      6'd19  : SDI <=0;
      6'd20  : SDI <=1'bz; //Slave ACK

//DATA0 (n+1)
6'd21  : SDI <=0;
      6'd22  : SDI <=0;
   6'd23  : SDI <=0;
   6'd24  : SDI <=0;
   6'd25  : SDI <=valor[3];
   6'd26  : SDI <=valor[2];
   6'd27  : SDI <=valor[1];
   6'd28  : SDI <=valor[0];
   6'd29  : SDI <=1'bz;

      //DATA0 (n+2)
6'd30  : SDI <=0;
      6'd31  : SDI <=0;
   6'd32  : SDI <=0;
   6'd33  : SDI <=0;
   6'd34  : SDI <=~valor[3];
   6'd35  : SDI <=~valor[2];
   6'd36  : SDI <=~valor[1];
   6'd37  : SDI <=~valor[0];
   6'd38  : SDI <=1'bz;

      //DATA1 (n+3)
6'd39  : SDI <=0;
      6'd40  : SDI <=0;
   6'd41  : SDI <=0;
   6'd42  : SDI <=0;
   6'd43  : SDI <=valor[3];
   6'd44  : SDI <=valor[2];
   6'd45  : SDI <=valor[1];
   6'd46  : SDI <=~valor[0];
   6'd47  : SDI <=1'bz;

      //DATA1 (n+4)
6'd48  : SDI <=valor[3];
      6'd49  : SDI <=valor[2];
   6'd50  : SDI <=~valor[1];
   6'd51  : SDI <=~valor[0];
   6'd52  : SDI <=0;
   6'd53  : SDI <=0;
   6'd54  : SDI <=0;
```

```
          6'd55  : SDI <=0;
          6'd56  : SDI <=1'bz;

          //Condició de STOP
       6'd57 : begin SDI <= 1'b0; SCLK <= 1'b1;final=1'b1;end
             6'd58 : SDI <= 1'b1;
      endcase

end

endmodule
```

A.3 DE0_NANO

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================
module DE0_NANO(

      ////////////// CLOCK //////////
      CLOCK_50,
      ////////////// LED //////////
      LED,
      ////////////// KEY //////////
      KEY,
      ////////////// SW //////////
      SW,

      ////////////// ADC //////////
      ADC_CS_N,
      ADC_SADDR,
      ADC_SCLK,
      ADC_SDAT
            );

//=======================================================
//  PORT declarations
//=======================================================

////////////// CLOCK //////////
input                           CLOCK_50;

////////////// LED //////////
output                [7:0]     LED;

////////////// KEY //////////
input                 [1:0]     KEY;

////////////// SW //////////
input                 [3:0]     SW;
```

```
////////////// ADC //////////
output                              ADC_CS_N;
output                              ADC_SADDR;
output                              ADC_SCLK;
input                               ADC_SDAT;

//=======================================================
//  REG/WIRE declarations
//=======================================================
wire                                wSPI_CLK;
wire                                wSPI_CLK_n;

//=======================================================
//  Structural coding
//=======================================================
SPIPLL            U0    (
                                    .inclk0(CLOCK_50),
                                    .c0(wSPI_CLK),
                                    .c1(wSPI_CLK_n)
                              );

ADC_CTRL          U1    (
                                    .iRST(KEY[0]),
                                    .iCLK(wSPI_CLK),
                                    .iCLK_n(wSPI_CLK_n),
                                    .iGO(KEY[1]),
                                    .iCH(SW[2:0]),
                                    .oLED(LED),

                                    .oDIN(ADC_SADDR),
                                    .oCS_n(ADC_CS_N),
                                    .oSCLK(ADC_SCLK),
                                    .iDOUT(ADC_SDAT)
                              );
endmodule
```

A.3.1 SPIPLL

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll

// ============================================================
// File Name: SPIPLL.v
// Megafunction Name(s):
//               altpll
//
// Simulation Library Files(s):
//               altera_mf
// ============================================================
// ************************************************************
```

```
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module SPIPLL (
      inclk0,
      c0,
      c1);
      input   inclk0;
      output        c0;
      output        c1;
      wire [4:0] sub_wire0;
      wire [0:0] sub_wire5 = 1'h0;
      wire [1:1] sub_wire2 = sub_wire0[1:1];
      wire [0:0] sub_wire1 = sub_wire0[0:0];
      wire  c0 = sub_wire1;
      wire  c1 = sub_wire2;
      wire  sub_wire3 = inclk0;
      wire [1:0] sub_wire4 = {sub_wire5, sub_wire3};
      altpll      altpll_component (
                  .inclk (sub_wire4),
                  .clk (sub_wire0),
                  .activeclock (),
                  .areset (1'b0),
                  .clkbad (),
                  .clkena ({6{1'b1}}),
                  .clkloss (),
                  .clkswitch (1'b0),
                  .configupdate (1'b0),
                  .enable0 (),
                  .enable1 (),
                  .extclk (),
                  .extclkena ({4{1'b1}}),
                  .fbin (1'b1),
                  .fbmimicbidir (),
                  .fbout (),
                  .fref (),
                  .icdrclk (),
                  .locked (),
                  .pfdena (1'b1),
                  .phasecounterselect ({4{1'b1}}),
                  .phasedone (),
                  .phasestep (1'b1),
                  .phaseupdown (1'b1),
                  .pllena (1'b1),
                  .scanaclr (1'b0),
                  .scanclk (1'b0),
                  .scanclkena (1'b1),
                  .scandata (1'b0),
                  .scandataout (),
                  .scandone (),
                  .scanread (1'b0),
                  .scanwrite (1'b0),
                  .sclkout0 (),
```

```
                    .sclkout1 (),
                    .vcooverrange (),
                    .vcounderrange ());
    defparam
          altpll_component.bandwidth_type = "AUTO",
          altpll_component.clk0_divide_by = 50,
          altpll_component.clk0_duty_cycle = 50,
          altpll_component.clk0_multiply_by = 1,
          altpll_component.clk0_phase_shift = "0",
          altpll_component.clk1_divide_by = 50,
          altpll_component.clk1_duty_cycle = 50,
          altpll_component.clk1_multiply_by = 1,
          altpll_component.clk1_phase_shift = "500000",
          altpll_component.compensate_clock = "CLK0",
          altpll_component.inclk0_input_frequency = 20000,
          altpll_component.intended_device_family = "Cyclone III",
          altpll_component.lpm_hint = "CBX_MODULE_PREFIX=SPIPLL",
          altpll_component.lpm_type = "altpll",
          altpll_component.operation_mode = "NORMAL",
          altpll_component.pll_type = "AUTO",
          altpll_component.port_activeclock = "PORT_UNUSED",
          altpll_component.port_areset = "PORT_UNUSED",
          altpll_component.port_clkbad0 = "PORT_UNUSED",
          altpll_component.port_clkbad1 = "PORT_UNUSED",
          altpll_component.port_clkloss = "PORT_UNUSED",
          altpll_component.port_clkswitch = "PORT_UNUSED",
          altpll_component.port_configupdate = "PORT_UNUSED",
          altpll_component.port_fbin = "PORT_UNUSED",
          altpll_component.port_inclk0 = "PORT_USED",
          altpll_component.port_inclk1 = "PORT_UNUSED",
          altpll_component.port_locked = "PORT_UNUSED",
          altpll_component.port_pfdena = "PORT_UNUSED",
          altpll_component.port_phasecounterselect = "PORT_UNUSED",
          altpll_component.port_phasedone = "PORT_UNUSED",
          altpll_component.port_phasestep = "PORT_UNUSED",
          altpll_component.port_phaseupdown = "PORT_UNUSED",
          altpll_component.port_pllena = "PORT_UNUSED",
          altpll_component.port_scanaclr = "PORT_UNUSED",
          altpll_component.port_scanclk = "PORT_UNUSED",
          altpll_component.port_scanclkena = "PORT_UNUSED",
          altpll_component.port_scandata = "PORT_UNUSED",
          altpll_component.port_scandataout = "PORT_UNUSED",
          altpll_component.port_scandone = "PORT_UNUSED",
          altpll_component.port_scanread = "PORT_UNUSED",
          altpll_component.port_scanwrite = "PORT_UNUSED",
          altpll_component.port_clk0 = "PORT_USED",
          altpll_component.port_clk1 = "PORT_USED",
          altpll_component.port_clk2 = "PORT_UNUSED",
          altpll_component.port_clk3 = "PORT_UNUSED",
          altpll_component.port_clk4 = "PORT_UNUSED",
          altpll_component.port_clk5 = "PORT_UNUSED",
          altpll_component.port_clkena0 = "PORT_UNUSED",
          altpll_component.port_clkena1 = "PORT_UNUSED",
          altpll_component.port_clkena2 = "PORT_UNUSED",
```

```
            altpll_component.port_clkena3 = "PORT_UNUSED",
            altpll_component.port_clkena4 = "PORT_UNUSED",
            altpll_component.port_clkena5 = "PORT_UNUSED",
            altpll_component.port_extclk0 = "PORT_UNUSED",
            altpll_component.port_extclk1 = "PORT_UNUSED",
            altpll_component.port_extclk2 = "PORT_UNUSED",
            altpll_component.port_extclk3 = "PORT_UNUSED",
            altpll_component.width_clock = 5;


endmodule
```

A.3.2 ADC_CTRL

```
module ADC_CTRL   (iRST, iCLK, iCLK_n, iGO, iCH, oLED, oDIN, oCS_n, oSCLK,
iDOUT);

input                   iRST;
input                   iCLK;
input                   iCLK_n;
input                   iGO;
input [2:0]       iCH;
output      [7:0]       oLED;

output                        oDIN;
output                        oCS_n;
output                        oSCLK;
input                   iDOUT;
//Variables internes//
reg                           data;
reg                           go_en;
wire  [2:0]       ch_sel;
reg                           sclk;
reg         [3:0]       cont;
reg         [3:0]       m_cont;
reg         [11:0]            adc_data;
reg         [7:0]       led;
/////
assign      oCS_n       =     ~go_en;
assign      oSCLK       =     (go_en)? iCLK:1;
assign      oDIN        =     data;
assign      ch_sel            =     iCH;
assign      oLED        =     led;
//Primera estructura always
always@(posedge iGO or negedge iRST)
begin
      if(!iRST)
            go_en <=    0;
      else
      begin
            if(iGO)
                  go_en <=    1;
```

```
	end
end
//Segona estructura always
always@(posedge iCLK or negedge go_en)
begin
	if(!go_en)
		cont  <=    0;
	else
	begin
		if(iCLK)
			cont  <=    cont + 1;
	end
end
//Tercera estructura always
always@(posedge iCLK_n)
begin
	if(iCLK_n)
		m_cont      <=    cont;
end
//Quarta estructura always
always@(posedge iCLK_n or negedge go_en)
begin
	if(!go_en)
		data  <=    0;
	else
	begin
		if(iCLK_n)
		begin //Selecció de l'entrada que es vol llegir
			if (cont == 2)
				data  <=    iCH[2];
			else if (cont == 3)
				data  <=    iCH[1];
			else if (cont == 4)
				data  <=    iCH[0];
			else
				data  <=    0;
		end
	end
end
//Cinquena estructura always
always@(posedge iCLK or negedge go_en)
begin
	if(!go_en)
	begin
		adc_data    <=    0;
		led         <=    8'h00;
	end
	else
	begin
		if(iCLK)
		begin  //Es reben les dades
			if (m_cont == 4)
				adc_data[11]      <=    iDOUT;
			else if (m_cont == 5)
```

```
                        adc_data[10]      <=    iDOUT;
                  else if (m_cont == 6)
                        adc_data[9]       <=    iDOUT;
                  else if (m_cont == 7)
                        adc_data[8]       <=    iDOUT;
                  else if (m_cont == 8)
                        adc_data[7]       <=    iDOUT;
                  else if (m_cont == 9)
                        adc_data[6]       <=    iDOUT;
                  else if (m_cont == 10)
                        adc_data[5]       <=    iDOUT;
                  else if (m_cont == 11)
                        adc_data[4]       <=    iDOUT;
                  else if (m_cont == 12)
                        adc_data[3]       <=    iDOUT;
                  else if (m_cont == 13)
                        adc_data[2]       <=    iDOUT;
                  else if (m_cont == 14)
                        adc_data[1]       <=    iDOUT;
                  else if (m_cont == 15)
                        adc_data[0]       <=    iDOUT;
                  else if (m_cont == 1)
                        led   <=    adc_data[11:4];
            end
      end
end
endmodule
```

A.4 DE0_NANO_G_Sensor

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module DE0_NANO_G_Sensor(

      ///////////// CLOCK /////////
      CLOCK_50,

      ///////////// LED /////////
      LED,

      ///////////// KEY /////////
      KEY,

      ///////////// Accelerometer and EEPROM /////////
      G_SENSOR_CS_N,
      G_SENSOR_INT,
      i2C_SCLK,
      I2C_SDAT
);
```

```
//=======================================================
//  PORT declarations
//=======================================================

////////////// CLOCK //////////
input                               CLOCK_50;

////////////// LED //////////
output                [7:0]       LED;

////////////// KEY //////////
input                 [1:0]       KEY;

////////////// Accelerometer and EEPROM //////////
output                              G_SENSOR_CS_N;
input                               G_SENSOR_INT;
output                              i2C_SCLK;
inout                               I2C_SDAT;

//=======================================================
//  REG/WIRE declarations
//=======================================================
wire          dly_rst;
wire          spi_clk, spi_clk_out;
wire  [15:0]  data_x;

//=======================================================
//  Structural coding
//=======================================================
//    Reset, mòdul per reiniciar el programa
reset_delay u_reset_delay     (
            .iRSTN(KEY[0]),
            .iCLK(CLOCK_50),
            .oRST(dly_rst));

//  PLL
spipll u_spipll   (
            .areset(dly_rst),
            .inclk0(CLOCK_50),
            .c0(spi_clk),      // 2MHz
            .c1(spi_clk_out)); // 2MHz phase shift
//  Initial Setting and Data Read Back
spi_ee_config u_spi_ee_config (
                                    .iRSTN(!dly_rst),

                                    .iSPI_CLK(spi_clk),

                                    .iSPI_CLK_OUT(spi_clk_out),

                                    .iG_INT2(G_SENSOR_INT),
                                    .oDATA_L(data_x[7:0]),
                                    .oDATA_H(data_x[15:8]),
                                    .SPI_SDIO(I2C_SDAT),
                                    .oSPI_CSN(G_SENSOR_CS_N),
                                    .oSPI_CLK(i2C_SCLK));
```

```
//    LED
led_driver u_led_driver (
                                    .iRSTN(!dly_rst),
                                    .iCLK(CLOCK_50),
                                    .iDIG(data_x[9:0]),
                                    .iG_INT2(G_SENSOR_INT),
                                    .oLED(LED));

endmodule
```

A.4.1 Spi_param

```
// En aquest fitxer hi han totes les variables que son constants, com poden
ser les adreces de l'acceleròmetre
// Data MSB Bit
parameter   IDLE_MSB          =     14;
parameter   SI_DataL          =     15;
parameter   SO_DataL          =     7;

// Write/Read Mode
parameter   WRITE_MODE        =     2'b00;
parameter   READ_MODE           =   2'b10;

// Initial Reg Number
parameter   INI_NUMBER    =   4'd11;

// SPI State
parameter   IDLE              =     1'b0;
parameter   TRANSFER            =   1'b1;

// Write Reg Address
parameter   BW_RATE                   =   6'h2c;
parameter   POWER_CONTROL     =     6'h2d;
parameter   DATA_FORMAT       =     6'h31;
parameter   INT_ENABLE        =     6'h2E;
parameter   INT_MAP                   =   6'h2F;
parameter   THRESH_ACT        =     6'h24;
parameter   THRESH_INACT      =     6'h25;
parameter   TIME_INACT        =     6'h26;
parameter   ACT_INACT_CTL     =     6'h27;
parameter   THRESH_FF           =   6'h28;
parameter   TIME_FF                   =   6'h29;

// Read Reg Address
parameter   INT_SOURCE        =     6'h30; // INT Status
parameter   X_LB                  = 6'h32; // Low Byte
parameter   X_HB                  = 6'h33; // High Byte
parameter   Y_LB                  = 6'h34; // Low Byte
parameter   Y_HB                  = 6'h35; // High Byte
parameter   Z_LB                  = 6'h36; // Low Byte
parameter   Z_HB                  = 6'h37; // High Byte
```

A.4.2 Spi_ee_config

```
module spi_ee_config (iRSTN, iSPI_CLK, iSPI_CLK_OUT, iG_INT2, oDATA_L,
oDATA_H, SPI_SDIO, oSPI_CSN, oSPI_CLK);

`include "spi_param.h"

//=====================================================
//  PORT declarations
//=====================================================
//    Host Side
input                                   iRSTN;
input                                   iSPI_CLK, iSPI_CLK_OUT;
input                                   iG_INT2;
output reg [SO_DataL:0] oDATA_L;
output reg [SO_DataL:0] oDATA_H;
//    SPI Side
inout                                   SPI_SDIO;
output                                      oSPI_CSN;
output                                      oSPI_CLK;

//=====================================================
//  REG/WIRE declarations
//=====================================================
reg       [3:0]          ini_index;
reg           [SI_DataL-2:0] write_data;
reg           [SI_DataL:0]     p2s_data;
reg                     spi_go;
wire                    spi_end;
wire    [SO_DataL:0]     s2p_data;
reg     [SO_DataL:0]     low_byte_data;
reg                                 spi_state;
reg                     high_byte; // indicate to read the high or low byte
reg                     read_back; // indicate to read back data
reg                     clear_status, read_ready;
reg     [3:0]           clear_status_d;
reg                     high_byte_d, read_back_d;
reg       [IDLE_MSB:0]   read_idle_count; // reducing the reading rate

//=====================================================
//  Sub-module
//=====================================================
spi_controller u_spi_controller (
                                        .iRSTN(iRSTN),
                                        .iSPI_CLK(iSPI_CLK),
                                        .iSPI_CLK_OUT(iSPI_CLK_OUT),
                                        .iP2S_DATA(p2s_data),
                                        .iSPI_GO(spi_go),
                                        .oSPI_END(spi_end),

                                        .oS2P_DATA(s2p_data),

                                        .SPI_SDIO(SPI_SDIO),
```

```
                                        .oSPI_CSN(oSPI_CSN),

                                        .oSPI_CLK(oSPI_CLK));

//=====================================================
//  Structural coding
//=====================================================
// Initial Setting Table
//Primera estructura always
always @ (ini_index)
      case (ini_index)
    0      : write_data = {THRESH_ACT,8'h20};
    1      : write_data = {THRESH_INACT,8'h03};
    2      : write_data = {TIME_INACT,8'h01};
    3      : write_data = {ACT_INACT_CTL,8'h7f};
    4      : write_data = {THRESH_FF,8'h09};
    5      : write_data = {TIME_FF,8'h46};
    6      : write_data = {BW_RATE,8'h09}; // output data rate : 50 Hz
    7      : write_data = {INT_ENABLE,8'h10};
    8      : write_data = {INT_MAP,8'h10};
    9      : write_data = {DATA_FORMAT,8'h40};
        default: write_data = {POWER_CONTROL,8'h08};
      endcase
//Tercera estructura always
always@(posedge iSPI_CLK or negedge iRSTN)
      if(!iRSTN)
      begin
            ini_index   <= 4'b0;
            spi_go            <= 1'b0;
            spi_state   <= IDLE;
            read_idle_count <= 0; // read mode only
            high_byte <= 1'b0; // read mode only
            read_back <= 1'b0; // read mode only
    clear_status <= 1'b0;
      end
      // initial setting (write mode)
      else if(ini_index < INI_NUMBER)
            case(spi_state)
                  IDLE : begin
                              p2s_data  <= {WRITE_MODE, write_data};
                              spi_go            <= 1'b1;
                              spi_state   <= TRANSFER;
                  end
                  TRANSFER : begin
                              if (spi_end)
                              begin
                    ini_index <= ini_index + 4'b1;
                                    spi_go            <= 1'b0;
                                    spi_state   <= IDLE;

                              end
                  end
            endcase
  // read data and clear interrupt (read mode)
```

```
  else
           case(spi_state)
                 IDLE : begin
                         read_idle_count <= read_idle_count + 1;

                             if (high_byte) // multiple-byte read
                         begin
                               p2s_data[15:8] <= {READ_MODE, X_HB};   //
Escull l'adreça que es vol llegir
                               read_back      <= 1'b1;
                             end
                         else if (read_ready)
                         begin
                               p2s_data[15:8] <= {READ_MODE, X_LB};   //
Escull la segon apart de l'adreça llegida.
                               read_back      <= 1'b1;
                             end
                         else if (!clear_status_d[3]&&iG_INT2 ||
read_idle_count[IDLE_MSB])
                         begin
                               p2s_data[15:8] <= {READ_MODE, INT_SOURCE};
                               clear_status   <= 1'b1;
          end

          if (high_byte || read_ready || read_idle_count[IDLE_MSB] ||
!clear_status_d[3]&&iG_INT2)
          begin
                               spi_go          <= 1'b1;
                               spi_state <= TRANSFER;
                             end

                         if (read_back_d) // update the read back data
                         begin
                             if (high_byte_d)
                             begin
                               oDATA_H <= s2p_data;
                               oDATA_L <= low_byte_data;

                             end
                             else
                                   low_byte_data <= s2p_data;
                         end
                 end
                 TRANSFER : begin
                             if (spi_end)
                             begin
                                   spi_go            <= 1'b0;
                                   spi_state   <= IDLE;

                                   if (read_back)
                                   begin
                                         read_back <= 1'b0;
                                    high_byte <= !high_byte;
                                    read_ready <= 1'b0;
```

```
                              end
                              else
                              begin
              clear_status <= 1'b0;
              read_ready <= s2p_data[6];
                                read_idle_count <= 0;
            end
                            end
                 end
            endcase
Segona estructura always
always@(posedge iSPI_CLK or negedge iRSTN)
      if(!iRSTN)
      begin
            high_byte_d <= 1'b0;
            read_back_d <= 1'b0;
            clear_status_d <= 4'b0;
      end
      else
      begin
            high_byte_d <= high_byte;
            read_back_d <= read_back;
            clear_status_d <= {clear_status_d[2:0], clear_status};
      end
endmodule
```

A.4.3 Spi_controler

```
module spi_controller (
                                          iRSTN,
                                          iSPI_CLK,
                                          iSPI_CLK_OUT,
                                          iP2S_DATA,
                                          iSPI_GO,
                                          oSPI_END,

                                          oS2P_DATA,

                                          SPI_SDIO,
                                          oSPI_CSN,

                                          oSPI_CLK);

`include "spi_param.h"

//=============================================================
//  PORT declarations
//=============================================================
//    Host Side
input                                  iRSTN;
input                                  iSPI_CLK;
input                                  iSPI_CLK_OUT;
input       [SI_DataL:0]  iP2S_DATA;
```

```
input                                   iSPI_GO;
output                                    oSPI_END;
output      reg [SO_DataL:0]  oS2P_DATA;
//    SPI Side
inout                                   SPI_SDIO;
output                                    oSPI_CSN;
output                                      oSPI_CLK;

//=============================================================
//  Variables internes REG/WIRE
//=============================================================
wire          read_mode, write_address;
reg           spi_count_en;
reg   [3:0]       spi_count;

//=============================================================
//  Structural coding
//=============================================================
assign read_mode = iP2S_DATA[SI_DataL];
assign write_address = spi_count[3];
assign oSPI_END = ~|spi_count;
assign oSPI_CSN = ~iSPI_GO;
assign oSPI_CLK = spi_count_en ? iSPI_CLK_OUT : 1'b1;
assign SPI_SDIO = spi_count_en && (!read_mode || write_address) ?
iP2S_DATA[spi_count] : 1'bz;

always @ (posedge iSPI_CLK or negedge iRSTN)
      if (!iRSTN)
      begin
            spi_count_en <= 1'b0;
            spi_count <= 4'hf;
      end
      else
      begin
            if (oSPI_END)
                  spi_count_en <= 1'b0;
            else if (iSPI_GO)
                  spi_count_en <= 1'b1;

            if (!spi_count_en)
            spi_count <= 4'hf;
            else
                  spi_count   <= spi_count - 4'b1;

    if (read_mode && !write_address)
              oS2P_DATA <= {oS2P_DATA[SO_DataL-1:0], SPI_SDIO};
      end

endmodule
```

A.4.4 Reset_delay

```
module      reset_delay(iRSTN, iCLK, oRST);
```

```
input           iRSTN;
input           iCLK;
output reg  oRST;
//Variables internes
reg  [20:0] cont;

always @(posedge iCLK or negedge iRSTN)
  if (!iRSTN)
  begin
    cont     <= 21'b0;
    oRST     <= 1'b1;
  end
  else if (!cont[20])
  begin
    cont <= cont + 21'b1;
    oRST <= 1'b1;
  end
  else
    oRST <= 1'b0;
 endmodule
```

A.4.5 Led_driver

```
module led_driver (iRSTN, iCLK, iDIG, iG_INT2, oLED);
input                          iRSTN;
input                          iCLK;
input           [9:0]  iDIG;
input                  iG_INT2;
output          [7:0]  oLED;

//=======================================================
//  Variables internes REG/WIRE declarations
//=======================================================
wire                    [4:0]  select_data;
wire               signed_bit;
wire                    [3:0]  abs_select_high;
reg                       [1:0]  int2_d;
reg           [23:0] int2_count;
reg                   int2_count_en;

//=======================================================
//  Structural coding
//=======================================================
assign select_data = iG_INT2 ? iDIG[9:5] :  // +-2g resolution : 10-bit

(iDIG[9]?(iDIG[8]?iDIG[8:4]:5'h10):(iDIG[8]?5'hf:iDIG[8:4])); // +-g
resolution : 9-bit
assign signed_bit = select_data[4];
assign abs_select_high = signed_bit ? ~select_data[3:0] : select_data[3:0];
// the negitive number here is the 2's complement - 1

assign oLED = int2_count[23] ? ((abs_select_high[3:1] == 3'h0) ? 8'h18 :
```

```
                                                    (abs_select_high[3:1] ==
3'h1) ? (signed_bit?8'h8:8'h10) :
                                                    (abs_select_high[3:1] ==
3'h2) ? (signed_bit?8'hc:8'h30) :
                                                    (abs_select_high[3:1] ==
3'h3) ? (signed_bit?8'h4:8'h20) :
                                                    (abs_select_high[3:1] ==
3'h4) ? (signed_bit?8'h6:8'h60) :
                                                    (abs_select_high[3:1] ==
3'h5) ? (signed_bit?8'h2:8'h40) :
                                                    (abs_select_high[3:1] ==
3'h6) ? (signed_bit?8'h3:8'hc0) :

(signed_bit?8'h1:8'h80)):
                                                    (int2_count[20] ? 8'h0 :
8'hff); // Activity

always@(posedge iCLK or negedge iRSTN)
      if (!iRSTN)
  begin
    int2_count_en <= 1'b0;
    int2_count <= 24'h800000;
  end
      else
      begin
            int2_d <= {int2_d[0], iG_INT2};

            if (!int2_d[1] && int2_d[0])
    begin
      int2_count_en    <= 1'b1;
          int2_count <= 24'h0;
        end
        else if (int2_count[23])
            int2_count_en    <= 1'b0;
    else
            int2_count <= int2_count + 1;
      end

endmodule
```

A.4.6 Spipll

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll
// ============================================================
// File Name: spipll.v
// Megafunction Name(s):
//               altpll
// Simulation Library Files(s):
//               altera_mf
// ============================================================
```

```
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module spipll (
      areset,
      inclk0,
      c0,
      c1);

      input   areset;
      input   inclk0;
      output        c0;
      output        c1;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
      tri0    areset;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

      wire [4:0] sub_wire0;
      wire [0:0] sub_wire5 = 1'h0;
      wire [0:0] sub_wire2 = sub_wire0[0:0];
      wire [1:1] sub_wire1 = sub_wire0[1:1];
      wire  c1 = sub_wire1;
      wire  c0 = sub_wire2;
      wire  sub_wire3 = inclk0;
      wire [1:0] sub_wire4 = {sub_wire5, sub_wire3};

      altpll      altpll_component (
                        .areset (areset),
                        .inclk (sub_wire4),
                        .clk (sub_wire0),
                        .activeclock (),
                        .clkbad (),
                        .clkena ({6{1'b1}}),
                        .clkloss (),
                        .clkswitch (1'b0),
                        .configupdate (1'b0),
                        .enable0 (),
                        .enable1 (),
                        .extclk (),
                        .extclkena ({4{1'b1}}),
                        .fbin (1'b1),
                        .fbmimicbidir (),
                        .fbout (),
                        .fref (),
                        .icdrclk (),
                        .locked (),
                        .pfdena (1'b1),
                        .phasecounterselect ({4{1'b1}}),
                        .phasedone (),
```

```
                    .phasestep (1'b1),
                    .phaseupdown (1'b1),
                    .pllena (1'b1),
                    .scanaclr (1'b0),
                    .scanclk (1'b0),
                    .scanclkena (1'b1),
                    .scandata (1'b0),
                    .scandataout (),
                    .scandone (),
                    .scanread (1'b0),
                    .scanwrite (1'b0),
                    .sclkout0 (),
                    .sclkout1 (),
                    .vcooverrange (),
                    .vcounderrange ());
defparam
      altpll_component.bandwidth_type = "AUTO",
      altpll_component.clk0_divide_by = 25,
      altpll_component.clk0_duty_cycle = 50,
      altpll_component.clk0_multiply_by = 1,
      altpll_component.clk0_phase_shift = "200000",
      altpll_component.clk1_divide_by = 25,
      altpll_component.clk1_duty_cycle = 50,
      altpll_component.clk1_multiply_by = 1,
      altpll_component.clk1_phase_shift = "166667",
      altpll_component.compensate_clock = "CLK0",
      altpll_component.inclk0_input_frequency = 20000,
      altpll_component.intended_device_family = "Cyclone IV E",
      altpll_component.lpm_hint = "CBX_MODULE_PREFIX=spipll",
      altpll_component.lpm_type = "altpll",
      altpll_component.operation_mode = "NORMAL",
      altpll_component.pll_type = "AUTO",
      altpll_component.port_activeclock = "PORT_UNUSED",
      altpll_component.port_areset = "PORT_USED",
      altpll_component.port_clkbad0 = "PORT_UNUSED",
      altpll_component.port_clkbad1 = "PORT_UNUSED",
      altpll_component.port_clkloss = "PORT_UNUSED",
      altpll_component.port_clkswitch = "PORT_UNUSED",
      altpll_component.port_configupdate = "PORT_UNUSED",
      altpll_component.port_fbin = "PORT_UNUSED",
      altpll_component.port_inclk0 = "PORT_USED",
      altpll_component.port_inclk1 = "PORT_UNUSED",
      altpll_component.port_locked = "PORT_UNUSED",
      altpll_component.port_pfdena = "PORT_UNUSED",
      altpll_component.port_phasecounterselect = "PORT_UNUSED",
      altpll_component.port_phasedone = "PORT_UNUSED",
      altpll_component.port_phasestep = "PORT_UNUSED",
      altpll_component.port_phaseupdown = "PORT_UNUSED",
      altpll_component.port_pllena = "PORT_UNUSED",
      altpll_component.port_scanaclr = "PORT_UNUSED",
      altpll_component.port_scanclk = "PORT_UNUSED",
      altpll_component.port_scanclkena = "PORT_UNUSED",
      altpll_component.port_scandata = "PORT_UNUSED",
      altpll_component.port_scandataout = "PORT_UNUSED",
```

```
            altpll_component.port_scandone = "PORT_UNUSED",
            altpll_component.port_scanread = "PORT_UNUSED",
            altpll_component.port_scanwrite = "PORT_UNUSED",
            altpll_component.port_clk0 = "PORT_USED",
            altpll_component.port_clk1 = "PORT_USED",
            altpll_component.port_clk2 = "PORT_UNUSED",
            altpll_component.port_clk3 = "PORT_UNUSED",
            altpll_component.port_clk4 = "PORT_UNUSED",
            altpll_component.port_clk5 = "PORT_UNUSED",
            altpll_component.port_clkena0 = "PORT_UNUSED",
            altpll_component.port_clkena1 = "PORT_UNUSED",
            altpll_component.port_clkena2 = "PORT_UNUSED",
            altpll_component.port_clkena3 = "PORT_UNUSED",
            altpll_component.port_clkena4 = "PORT_UNUSED",
            altpll_component.port_clkena5 = "PORT_UNUSED",
            altpll_component.port_extclk0 = "PORT_UNUSED",
            altpll_component.port_extclk1 = "PORT_UNUSED",
            altpll_component.port_extclk2 = "PORT_UNUSED",
            altpll_component.port_extclk3 = "PORT_UNUSED",
            altpll_component.width_clock = 5;
endmodule
```

A.5 Comptador

```
/=========================================================
//  This code is generated by Terasic System Builder
//=========================================================
module comptador(

      ////////////// LED //////////
      output                  [7:0]        LED,

      ////////////// KEY //////////
      input               [1:0]          KEY
);
//=========================================================
//  Variables internes REG/WIRE declarations
//=========================================================
reg [3:0]     q;

//=========================================================
//  Structural coding
//=========================================================
assign LED[3:0]   = q;

always @(posedge KEY[1] or negedge KEY[0] )
if (!KEY[0])
q <= 8'b00000000;
else if (q<4'b1111)
q<=q+1;
else
q<=q;
```

```
endmodule
```

A.6 Flip flop d

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module flip_flop_d(

      ///////////// CLOCK /////////
      input                           CLOCK_50,

      ///////////// LED /////////
      output            reg    [7:0]          LED,

      ///////////// SW /////////
      input                 [3:0]       SW
);

//=======================================================
//  Structural coding
//=======================================================
always @ (posedge CLOCK_50)
begin
LED[0] <= SW[0];
LED[1] <= !SW[0];
end

endmodule
```

A.7 Multiplexor

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module multiplexor(
      ///////////// LED /////////
      output                [7:0]       LED,
      ///////////// SW /////////
      input                 [3:0]       SW
);
//=======================================================
//  Structural coding
//=======================================================
and #5 porta0   (res1,SW[0],nsel);
and #5 porta1   (res2,SW[1],SW[2]);
or  #5 porta2   (LED, res2, res1);
not               (nsel,SW[2]);
```

```
endmodule
```

A.8 Registre_d

```
//=======================================================
//  This code is generated by Terasic System Builder
//=======================================================

module registre_d(

      ////////////// LED //////////
      output                [7:0]        LED,

      ////////////// KEY //////////
      input                 [1:0]        KEY,

      ////////////// SW //////////
      input                 [3:0]        SW
);
//=======================================================
//  REG/WIRE declarations
//=======================================================
reg [7:0] registre;
wire      enable,a;
reg [3:0] count;

//=======================================================
//  Structural coding
//=======================================================
assign LED [7:0] = registre;
assign enable = SW[3];
assign a= enable&(count<3) ? SW[count]:1'b0;
// Primer procés always
always @(posedge KEY[1] or negedge KEY[0])
begin
if (!KEY[0])
registre <=0;
else
if (enable)
registre <={registre[6:0], a} ;
end
// Segon Procés always
always @(posedge KEY[1] or negedge KEY[0])
begin
      if (!KEY[0])
      count =0;
      else if (count < 10)
                  count <= count+1;
   else if (!enable & KEY[0])
                  count =0;
                  else
                  count =0;
```

```
end
endmodule
```

A.9 DE0_Nano_PLL

```
// megafunction wizard: %ALTPLL%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altpll

// ============================================================
// File Name: spipll.v
// Megafunction Name(s):
//               altpll
//
// Simulation Library Files(s):
//               altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module spipll (
      areset,
      inclk0,
      c0,
      c1);

      input   areset;
      input   inclk0;
      output        c0;
      output        c1;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
      tri0    areset;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

      wire [4:0] sub_wire0;
      wire [0:0] sub_wire5 = 1'h0;
      wire [0:0] sub_wire2 = sub_wire0[0:0];
      wire [1:1] sub_wire1 = sub_wire0[1:1];
      wire  c1 = sub_wire1;
      wire  c0 = sub_wire2;
      wire  sub_wire3 = inclk0;
      wire [1:0] sub_wire4 = {sub_wire5, sub_wire3};

      altpll      altpll_component (
```

```
                        .areset (areset),
                        .inclk (sub_wire4),
                        .clk (sub_wire0),
                        .activeclock (),
                        .clkbad (),
                        .clkena ({6{1'b1}}),
                        .clkloss (),
                        .clkswitch (1'b0),
                        .configupdate (1'b0),
                        .enable0 (),
                        .enable1 (),
                        .extclk (),
                        .extclkena ({4{1'b1}}),
                        .fbin (1'b1),
                        .fbmimicbidir (),
                        .fbout (),
                        .fref (),
                        .icdrclk (),
                        .locked (),
                        .pfdena (1'b1),
                        .phasecounterselect ({4{1'b1}}),
                        .phasedone (),
                        .phasestep (1'b1),
                        .phaseupdown (1'b1),
                        .pllena (1'b1),
                        .scanaclr (1'b0),
                        .scanclk (1'b0),
                        .scanclkena (1'b1),
                        .scandata (1'b0),
                        .scandataout (),
                        .scandone (),
                        .scanread (1'b0),
                        .scanwrite (1'b0),
                        .sclkout0 (),
                        .sclkout1 (),
                        .vcooverrange (),
                        .vcounderrange ());
    defparam
          altpll_component.bandwidth_type = "AUTO",
          altpll_component.clk0_divide_by = 25,
          altpll_component.clk0_duty_cycle = 50,
          altpll_component.clk0_multiply_by = 1,
          altpll_component.clk0_phase_shift = "200000",
          altpll_component.clk1_divide_by = 25,
          altpll_component.clk1_duty_cycle = 50,
          altpll_component.clk1_multiply_by = 1,
          altpll_component.clk1_phase_shift = "166667",
          altpll_component.compensate_clock = "CLK0",
          altpll_component.inclk0_input_frequency = 20000,
          altpll_component.intended_device_family = "Cyclone IV E",
          altpll_component.lpm_hint = "CBX_MODULE_PREFIX=spipll",
          altpll_component.lpm_type = "altpll",
          altpll_component.operation_mode = "NORMAL",
          altpll_component.pll_type = "AUTO",
```

```
        altpll_component.port_activeclock = "PORT_UNUSED",
        altpll_component.port_areset = "PORT_USED",
        altpll_component.port_clkbad0 = "PORT_UNUSED",
        altpll_component.port_clkbad1 = "PORT_UNUSED",
        altpll_component.port_clkloss = "PORT_UNUSED",
        altpll_component.port_clkswitch = "PORT_UNUSED",
        altpll_component.port_configupdate = "PORT_UNUSED",
        altpll_component.port_fbin = "PORT_UNUSED",
        altpll_component.port_inclk0 = "PORT_USED",
        altpll_component.port_inclk1 = "PORT_UNUSED",
        altpll_component.port_locked = "PORT_UNUSED",
        altpll_component.port_pfdena = "PORT_UNUSED",
        altpll_component.port_phasecounterselect = "PORT_UNUSED",
        altpll_component.port_phasedone = "PORT_UNUSED",
        altpll_component.port_phasestep = "PORT_UNUSED",
        altpll_component.port_phaseupdown = "PORT_UNUSED",
        altpll_component.port_pllena = "PORT_UNUSED",
        altpll_component.port_scanaclr = "PORT_UNUSED",
        altpll_component.port_scanclk = "PORT_UNUSED",
        altpll_component.port_scanclkena = "PORT_UNUSED",
        altpll_component.port_scandata = "PORT_UNUSED",
        altpll_component.port_scandataout = "PORT_UNUSED",
        altpll_component.port_scandone = "PORT_UNUSED",
        altpll_component.port_scanread = "PORT_UNUSED",
        altpll_component.port_scanwrite = "PORT_UNUSED",
        altpll_component.port_clk0 = "PORT_USED",
        altpll_component.port_clk1 = "PORT_USED",
        altpll_component.port_clk2 = "PORT_UNUSED",
        altpll_component.port_clk3 = "PORT_UNUSED",
        altpll_component.port_clk4 = "PORT_UNUSED",
        altpll_component.port_clk5 = "PORT_UNUSED",
        altpll_component.port_clkena0 = "PORT_UNUSED",
        altpll_component.port_clkena1 = "PORT_UNUSED",
        altpll_component.port_clkena2 = "PORT_UNUSED",
        altpll_component.port_clkena3 = "PORT_UNUSED",
        altpll_component.port_clkena4 = "PORT_UNUSED",
        altpll_component.port_clkena5 = "PORT_UNUSED",
        altpll_component.port_extclk0 = "PORT_UNUSED",
        altpll_component.port_extclk1 = "PORT_UNUSED",
        altpll_component.port_extclk2 = "PORT_UNUSED",
        altpll_component.port_extclk3 = "PORT_UNUSED",
        altpll_component.width_clock = 5;


endmodule
```