UNIVERSITAT OBERTA DE CATALUNYA

ENGINYERIA TÈCNICA EN INFORMÀTICA DE SISTEMES

TREBALL FI DE CARRERA

Emulador del Microcontrolador Texas Instruments MSP430

Autor: Maximiliano Carlos Pin Senz
Dirigit per: Ermengol Bota Arqué

Curs Primavera 2004

A Cristina.

Resum:

L'objectiu d'aquest projecte és la implementació d'un emulador del microcontrolador MSP430 de Texas Instruments, concretament del model MSP430f149. S'implementa l'emulació no només de la CPU, sinó també d'una part dels mòduls i perifèrics integrats en el microcontrolador. D'aquesta manera, es pot executar qualsevol programa que es pugui executar en la unitat real, sempre i quan no tingui requeriments de mòduls externs, encara que es deixa la porta oberta a l'emulació dels mateixos, ja que l'emulador és fàcilment extensible. A més, permet la depuració dels programes, mitjançant un depurador estàndard, d'una manera més flexible i còmoda que depurant sobre el dispositiu real. Una part important del projecte és la implementació d'un simulador de sistemes digitals no gràfic, sobre el qual es desenvolupa l'emulador. Això permet aconseguir un alt grau de realisme en la simulació, i simplifica alguns aspectes del desenvolupament, encara que dificulta altres. Es fa un ús extensiu del disseny i la programació orientats a objecte, i de les proves automatitzades, per tal de garantir la qualitat del producte.

Resumen:

El objetivo de este proyecto es la implementación de un emulador del microcontrolador MSP430 de Texas Instruments, concretamente del modelo MSP430f149. Se implementa la emulación no sólo de la CPU, sinó también de una parte de los módulos y periféricos integrados en el microcontrolador. De esta manera, se puede ejecutar cualquier programa que se pueda ejecutar en la unidad real, siempre y cuando no tenga requerimientos de módulos externos, aunque se deja la puerta abierta a la emulación de los mismos, ya que el emulador es fácilmente extensible. Además, permite la depuración de los programas, mediante un depurador estándar, de una manera más flexible y cómoda que depurando sobre el dispositivo real. Una parte importante del proyecto es la implementación de un simulador de sistemas digitales no gráfico, sobre el cual se desarrolla el emulador. Esto permite conseguir un alto grado de realismo en la simulación, y simplifica algunos aspectos del desarrollo, aunque dificulta otros. Se hace un uso extensivo del diseño y la programación orientados a objeto, y de las pruebas automatizadas, para garantizar la calidad del producto.

Abstract:

The goal of this project is to create an emulator of the Texas Instruments MSP430 microcontroller, concretely the model MSP430f149. Not only the CPU is emulated, but also many of its integrated modules and peripherals. This way, it's possible to execute any program that can be executed in the real unit, as long as it doesn't require external modules. Anyway, emulators of external modules can be created, since the emulator is easily expandable. Furthermore, it allows debugging of the programs executed, using a standard debugger, in a more flexible manner than using the real device. An important part of the project is the implementation of a non graphic simulator of digital systems. The emulator is developed on top of it. This allows to obtain a higher realism degree, and simplifies some aspects of the development, but makes other aspects more difficult. Object oriented design and programming, as well as automated tests, are extensively used to increase the quality of the product.

Índex

Índex de figures

Capítol 1

Introducció

1.1 Justificació i context

Quan es desenvolupen aplicacions complexes per a un microcontrolador, la depuració es torna cada vegada més important, i a la vegada més complicada. Aquest fet s'accentua en el desenvolupament de sistemes operatius i aplicacions distribuïdes. L'automatització de proves és pràcticament impossible. Els emuladors de microcontroladors ajuden enormement en aquesta tasca. En moltes aplicacions, també es fa molt necessari poder extendre aquests emuladors amb perifèrics que, tot i no ser part del microcontrolador, són necessaris per al correcte funcionament dels programes.

Existeixen emuladors per a gairebé tots els microcontroladors en totes les plataformes. Tot i així, no existeix cap emulador complet del MSP430. El *kit* de desenvolupament entregat per Texas Instruments, propietari de l'empresa IAR, inclou un simulador del microcontrolador, amb una emulació parcial i poc realista dels perifèrics i mòduls integrats. D'altra banda, la versió del depurador lliure GDB per al MSP430 inclou també un simulador, però només de la CPU. Cap dels dos simuladors és extensible amb perifèrics externs, i els dos tenen errades constatades durant la realització d'aquest projecte.

Els desenvolupadors de sistemes operatius i programes en general del MSP430 que treballen amb GNU/Linux (entre els que l'autor s'inclou), disposen d'excel.lents utilitats lliures per al desenvolupament, i d'un bon suport de la comunitat, tot gràcies al projecte MSPGCC [1]. Aquest grup de treball s'encarrega d'adaptar les eines de

desenvolupament de GNU (gcc, g++, gdb...) [2] per al seu ús amb els microcontroladors de la família MSP430. També és una bona font de documentació sobre aquests microcontroladors. La única peça que falta és un bon emulador del MSP430.

1.2 Objectius

Es pretén desenvolupar un emulador complet del microcontrolador MSP430f149 que funcioni sobre plataformes lliures (en principi, GNU/Linux). Aquest ha d'emular no només la CPU, sinó tots els mòduls i perifèrics integrats. En el capítol 2 veurem quins són aquests mòduls.

L'emulador tindrà les següents característiques:

1. El seu codi font serà lliurement accessible. Això permet, per exemple, convertir-lo fàcilment en una eina docent gràfica que mostri com funciona un microcontrolador internament.

2. Emularà de manera realista el comportament dels mòduls i perifèrics integrats (a diferència dels altres emuladors d'aquest dispositiu existents actualment, que ho fan de manera molt simplificada).

3. Serà extensible amb emuladors de perifèrics externs o mòduls electrònics, de manera que els programes de l'usuari puguin accedir-hi com si es tractessi de dispositius reals connectats al microcontrolador.

4. De la mateixa manera, es podran connectar varies instàncies del microcontrolador, mitjançant línies elèctriques virtuals, o fins i tot mitjançant emuladors d'equipament de radio-freqüència. Es podria, per exemple, simular una xarxa sense fil de dispositius basats en el MSP430, i estudiar com es comportaria al llarg d'un cert temps. També es poden desenvolupar emuladors d'altres microcontroladors, fent servir com a base el simulador genèric de sistemes digitals que es desenvolupa en aquest projecte. D'aquesta manera es podran simular comunicacions entre diferents microcontroladors.

1.3 Enfocament i mètode

Com ja s'ha anticipat, una part important del projecte és un simulador genèric de sistemes digitals. Aquest simulador ens permetrà no només connectar l'emulador del MSP430 amb emuladors de dispositius externs, sinó també desenvolupar l'emulador com un conjunt de mòduls independents connectats entre si.

Cada un d'aquests mòduls, així com el simulador de sistemes digitals, es desenvoluparan i provaran independentment. Finalment es farà un mòdul que connecti tots els altres mòduls, i es provarà aquest mòdul, que serà pròpiament l'emulador del MSP430.

Per a la interfície amb l'usuari s'aprofitarà el depurador GDB [6], que, amb pegats del projecte MSPGCC, soportasuporta el MSP430 com a objectiu remot. GDB no distingirà entre un dispositiu real i l'emulador. S'implementarà un servidor GDB remot amb les mateixes capacitats que el servidor corresponent al dispositiu real.

El llenguatge de programació triat és C++, ja que en aquest tipus de projecte l'orientació a objectes és bàsica, però també la velocitat d'execució (aspecte en el que C++ supera a qualsevol altre llenguatge orientat a objectes). Les proves s'automatitzaran fent servir l'eina CPPUNIT [3].

1.4 Planificació del projecte

La següent és la planificació proposada:

Setmana	Dates	Tasca	Esdeveniment
1	15/03 - 21/03	Pla de treball, estudi eines desenvolupament	PAC1
2	22/03 - 28/03	Simulador de sistemes digitals	
3	29/03 - 04/04	Simulador de sistemes digitals	
4	05/04 - 11/04	CPU	
5	12/04 - 18/04	CPU	PAC2
6	19/04 - 25/04	Flash, RAM, JTAG	
7	26/04 - 02/05	JTAG	
8	03/05 - 09/05	JTAG	
9	10/05 - 16/05	Ports E/S, Temporitzadors, Watchdog	
10	17/05 - 23/05	Ports USART	PAC3
11	24/05 - 30/05	Ports USART	
12	31/05 - 06/06	Comparador i ADC	
13	07/06 - 13/06	ADC	
14	14/06 - 18/06	Integració, conclusions	FINAL

Hi ha dues tasques que es faran si sobra temps, i sinó, es proposaran en les conclusions.

1. Mòdul MPY (multiplicador). És un mòdul que no existeix en els models anteriors del microcontrolador, i es fa servir en aplicacions molt concretes, per tant no es considera imprescindible (tot i que és molt desitjable per a la completesa de l'emulador).

2. Control voltatges entrada ADC. Consisteix en una aplicació (gràfica o no) per a controlar el voltatge aplicat a les entrades de l'ADC, o actuar com a generador de funcions.

1.5 Productes obtinguts

Els productes obtinguts són:

- Aquesta memòria, amb el disseny de les diferents parts de l'aplicació, així com els resultats de la investigació sobre el microcontrolador MSP430.
- Una llibreria genèrica per a la simulació de sistemes digitals.
- Un emulador complet del microcontrolador MSP430f149, extensible amb emuladors de perifèrics, mitjançant la llibreria mencionada.

1.6 Contingut de la memòria

Aquesta memòria conté els següents capítols i annexos:

Capítol 1 Introducció.

Capítol 2 El microcontrolador MSP430. Una introducció a l'estructura i funcionament de la unitat.

Capítol 3 Arquitectura de l'emulador. Descriu a grans trets les diferents parts del programa, i la relació entre elles.

Capítol 4 Simulador de sistemes digitals. Disseny del simulador genèric de sistemes digitals.

Capítol 5 CPU. Descripció i disseny de l'emulador de la CPU del MSP430.

Capítol 6 Mòduls i perifèrics integrats. Descripció i disseny dels emuladors de mòduls i perifèrics integrats en el MSP430.

Capítol 7 Integració. Es descriu la composició de la classe Msp430 a partir dels mòduls.

Capítol 8 Interfície amb GDB. Es descriu la connexió amb el depurador GDB.

Capítol 9 Exemple d'utilització. Instal.lació de les eines necessàries per a la utilització de l'emulador, i exemple de compilació i depuració d'un programa.

Capítol 10 Conclusions.

Annèx A Documentació del codi font, generada amb Doxygen [9].

Capítol 2

El microcontrolador MSP430

El microcontrolador MSP430 de Texas Instruments es caracteritza per un consum ultra baix i per un disseny molt senzill d'entendre. Es fa servir principalment en aplicacions de *metering* (comptadors d'aigua, gas, electricitat...), però també en molts altres àmbits. Per exemple, una aplicació que s'ha popularitzat en els últims anys són les xarxes sense fil amb el transceptor TRF6900 de Texas Instruments.

La CPU del MSP430 està bassada en tecnologia RISC (*Reduced Instruction Set Computer*), i és totalment ortogonal. Això vol dir que totes les instruccions funcionen amb tots els modes d'adreçament.

Existeix una gran varietat de models, que es diferencien en la quantitat de mòduls o perifèrics integrats que contenen, i en la capacitat dels mateixos. Entre aquests mòduls destaquen els temporitzadors, els ports de comunicació serie (UART/SPI), i l'ADC (*Analog to Digital Converter*). Els últims models inclouen també entre d'altres un DAC (*Digital to Analog Converter*), DMA (*Direct Memory Access*), i LCD (*Liquid Crystal Display*).

Per a aquest projecte s'ha triat el model MSP430f149, un model bastant avançat que inclou dos ports UART/SPI, un ADC de 12 bits, dos temporitzadors principals i un multiplicador, entre altres. No inclou DAC, DMA ni LCD. En la figura 2.1 es pot veure un diagrama dels seus components. En capítols posteriors veurem cada mòdul en detall.

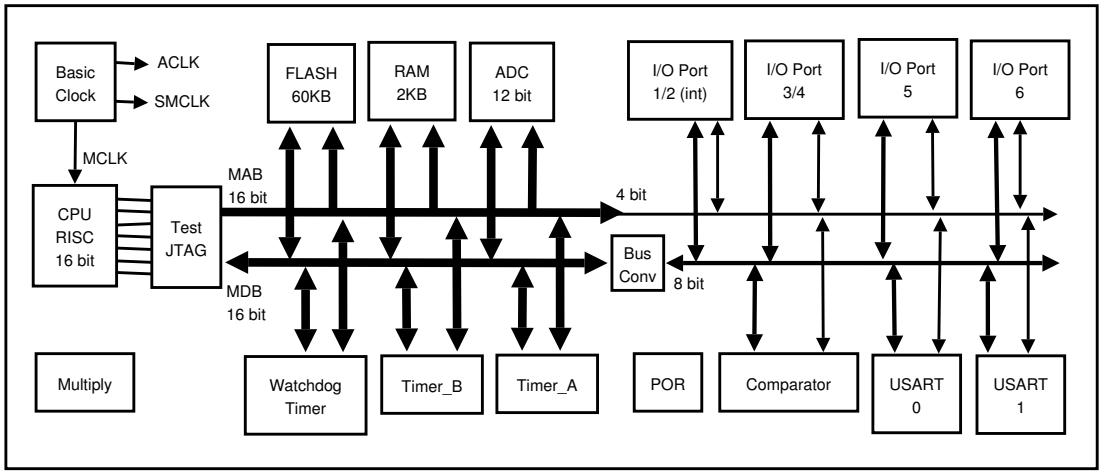


Figura 2.1: Components del MSP430f139

Capítol 3

Arquitectura de l'emulador

3.1 Simulador de sistemes digitals

Prèviament al desenvolupament de l'emulador, es crearà un simulador genèric de sistemes digitals. Això ens permetrà:

- Desenvolupar independentment emuladors dels diferents mòduls del microcontrolador. Cada mòdul estarà connectat dinàmicament als altres mitjançant ports connectats a línies del simulador de sistemes digitals.
- Proporcionar funcionalitats comunes que necessiten els mòduls, com per exemple el control de les sortides i les entrades, i el control del temps.
- Desenvolupar el propi emulador del MSP430 com un mòdul que forma part d'una placa emulada, on poden existir perifèrics externs o fins i tot altres instàncies de l'emulador. D'aquesta manera podem simular comunicacions entre més d'un microcontrolador.

Aquest simulador genèric s'implementa en forma d'una llibreria totalment reutilitzable anomenada "libdigisim". En el capítol 4 es detalla l'arquitectura d'aquesta llibreria. Les classes més importants de cara a la implementació de l'emulador són:

- Route: Representa un conjunt de connexions entre mòduls (línies).
- Module: Representa un mòdul electrònic digital amb entrades i sortides (ports).

- Composite: Subclasse de Module. És un mòdul que conté un conjunt d'objectes Module, connectats entre si mitjançant un objecte Route. Els mòduls inclosos poden ser objectes Composite.

3.2 Emulador del microcontrolador

L'emulador en si mateix és una subclasse de Composite, anomenada Msp430. Els mòduls que componen aquest Composite són la CPU i els mòduls i perifèrics integrats que componen el microcontrolador. Aquests són els components vistos en la figura 2.1. També són subclasses de Module o Composite.

Gairebé tots els perifèrics integrats tenen la necessitat de gestionar registres interns, manipulats mitjançant els busos d'adreces, dades i control, i amb diferents tipus de comportament per a cada bit. Tota aquesta funcionalitat s'encapsula en una classe anomenada Peripheral, subclasse de Module, i de la qual tots els perifèrics integrats hereten.

La figura 3.1 dóna una visió general de l'arquitectura.

3.3 Interfície amb GDB

La principal via que tindrà l'usuari per a interactuar amb l'emulador serà el programa GDB (GNU Debugger). Amb GDB es poden carregar, executar i depurar programes. GDB permet la utilització de dispositius remots. S'aprofitarà gran part del codi que implementa la comunicació entre GDB i el MSP430 real per a implementar la comunicació entre GDB i l'emulador. Per aconseguir això, es modificarà el programari lliure gdbproxy.

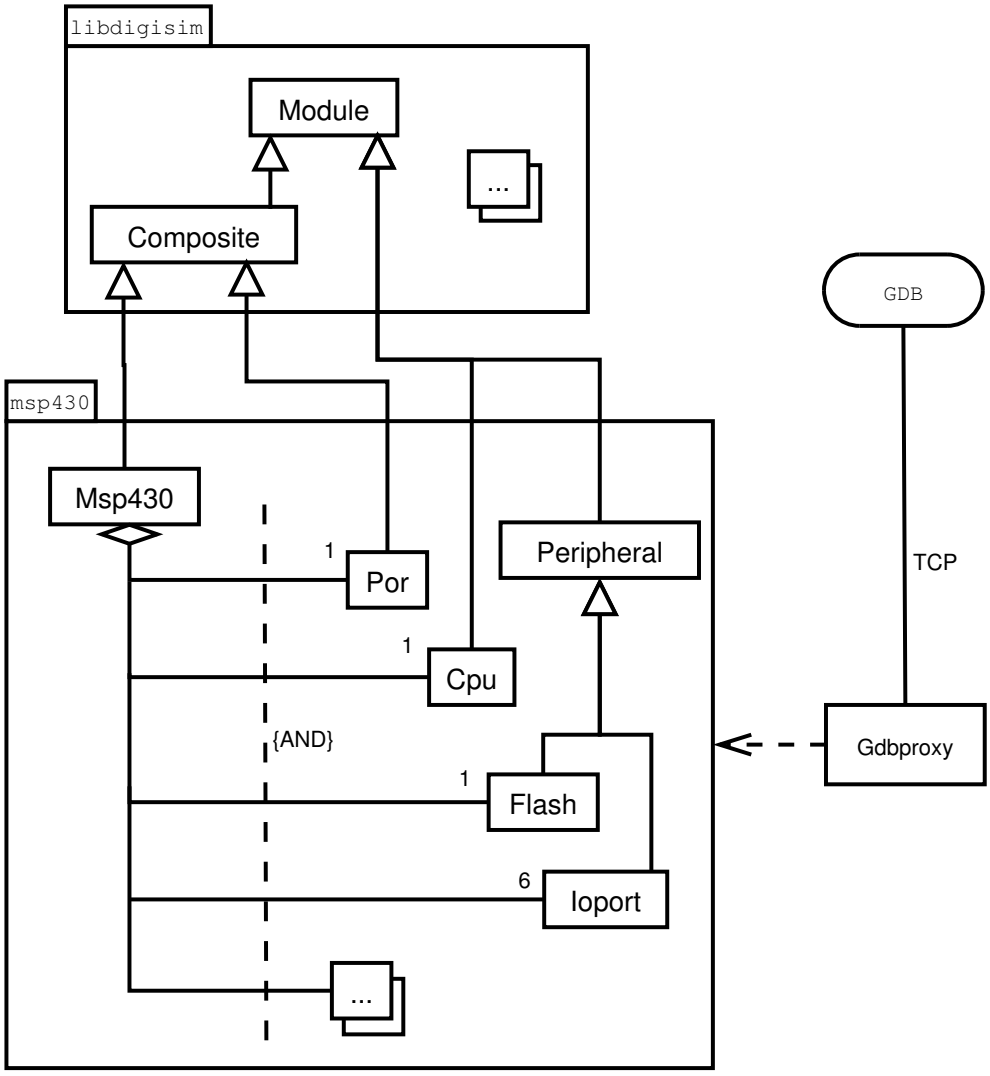


Figura 3.1: Esquema general de l'emulador.

3.4 Programa principal

El programa principal haurà d'utilitzar el simulador de sistemes digitals per a crear un circuit que únicament contingui una font d'alimentació, opcionalment un oscilador o dos (per als rellotges), i un objecte de la classe Msp430. Aquest circuit es pot complementar amb emuladors de perifèrics externs, altres microcontroladors, etc, sempre que aquests estiguin programats sobre "libdigisim".

Part del programa principal implementa la comunicació amb GDB, descrita anteriorment. La classe Msp430 haurà de tenir els mètodes necessaris per a permetre implementar les comandes de GDB.

Capítol 4

Simulador de sistemes digitals

4.1 Visió general

El simulador de sistemes digitals és una llibreria reutilitzable, anomenada "libdigisim", que servirà de base per a crear els diferents mòduls del microcontrolador. Cada mòdul es desenvoluparà independentment, i es comunicarà amb els altres mòduls mitjançant un conjunt de ports connectats a línies d'una placa emulada. Aquests mòduls són en realitat emuladors de mòduls físics. En la figura 4.1 podem veure la relació entre les classes d'aquesta llibreria.

Els objectes de la classe Route emmagatzemen les línies i la relació de ports connectats a aquestes línies. Els mòduls emulats, com els mòduls reals, no saben a quins altres mòduls estan connectats. El programa usuari estableix quins ports es connecten a quines línies.

Els mòduls que creem seran subclasses de la classe Module. Aquesta classe implementa els mètodes necessaris per a gestionar les sortides i les entrades. La classe Composite és una subclasse de Module que està composta d'altres mòduls. Així podrem crear l'equivalent a un circuit integrat. El mateix emulador del MSP430 serà una subclasse de Composite, i com que també és un Module, podrà formar part d'un Composite més gran amb perifèrics externs, per exemple.

La classe Real_time permet mantenir el temps del món simulat. Permet als mòduls programar esdeveniments. A part dels esdeveniments necessaris en la programació de

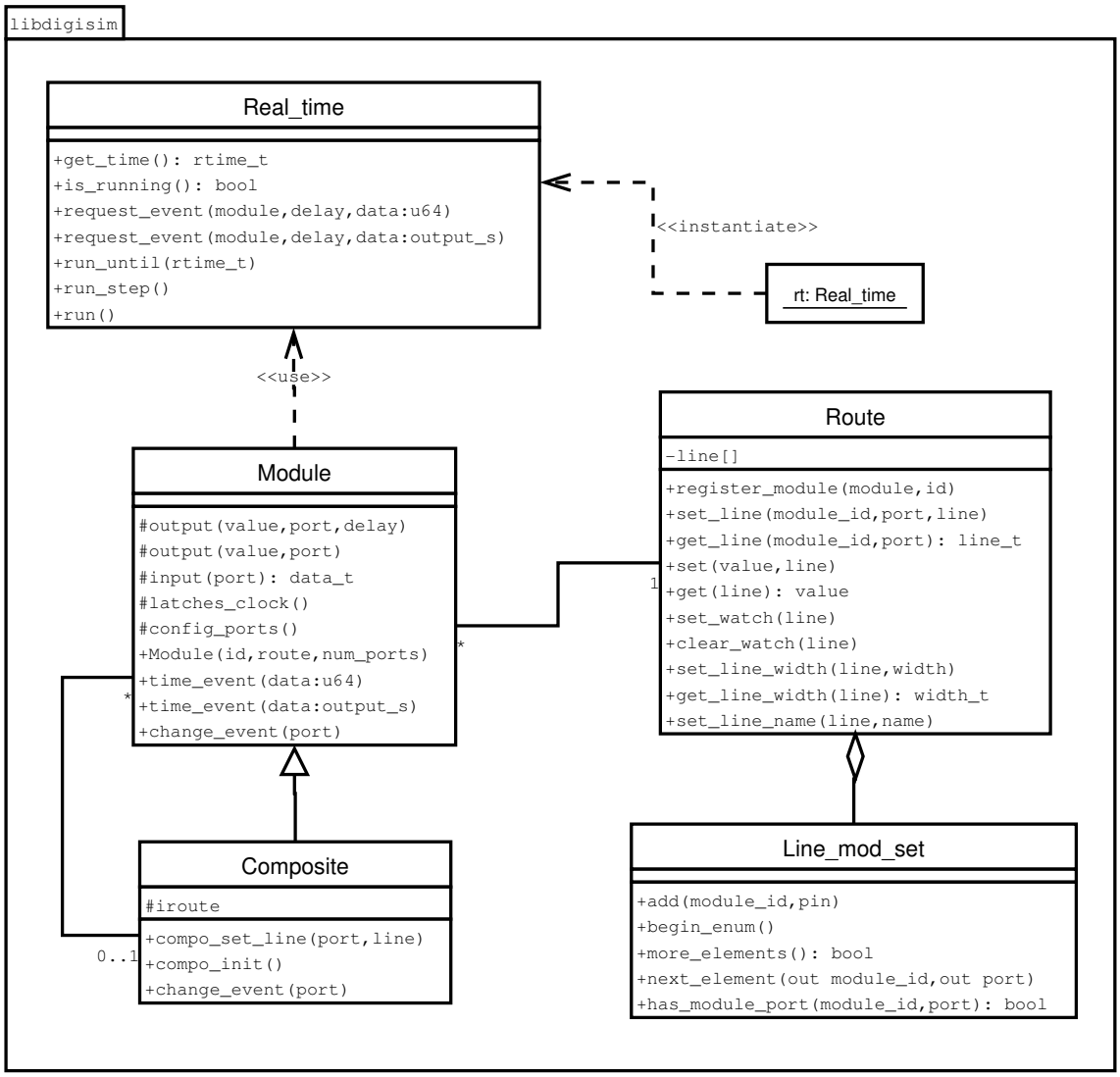


Figura 4.1: Diagrama estàtic de libdigisim.

cada mòdul, també es fa servir internament per la mateixa classe Module per a generar sortides retardades (*delay*).

A continuació veurem cada classe amb més detall.

4.2 Classe Module

Aquesta és la classe base de tots els emuladors de mòduls digitals. Des d'una simple porta lògica fins a un microcontrolador, tots hereten d'aquesta classe. Això és degut a que tots ells tenen ports d'entrada i de sortida, que serveixen per comunicar-se amb la resta de mòduls, tant en el món real com a l'emulador. Un port, tal com s'entén en aquest i altres simuladors de circuits digitals, pot tenir un o més bits, és a dir, pot representar un conjunt de pins del mòdul.

A més, també tenen en comú la necessitat de tenir una referència del temps actual, i poder actuar de manera retardada o periòdica. Amb aquest objectiu, tenim una altra classe anomenada Real_time, que veurem més endavant, i que s'encarrega de mantenir la referència del temps actual. Els objectes Module poden demanar-li que generi esdeveniments en el futur, que es materialitzen en forma de *callbacks*: els mètodes time_event.

L'usuari de la llibreria libdigisim ha de crear els seus mòduls heretant de la classe Module. Després, pot redefinir el mètode time_event(data: u64), que és cridat per l'objecte 'rt' (única instància de la classe Real_time) quan arriba un esdeveniment demanat. El paràmetre 'data' conté el mateix valor passat a 'rt' quan el mòdul va demanar l'esdeveniment. El mòdul d'usuari pot fer servir aquest paràmetre lliurement.

L'usuari també pot redefinir el mètode change_event. Aquest mètode és cridat per l'objecte Route on està el mòdul, quan la línia a la qual està connectat el port canvia de valor.

En el constructor es pot definir quins ports són de sortida i quins d'entrada, i d'aquests, quins tenen *latch* i quins no. Els ports que tenen *latch* només canvien de valor quan l'usuari crida el mètode protegit latches_clock(). Normalment ho farà quan detecti un flanc del rellotge principal.

En qualsevol moment el mòdul de l'usuari pot consultar o modificar l'estat de qualsevol port, mitjançant els mètodes protegits input i output. Les sortides poden tenir

retardament (*delay*). Per defecte tenen un retardament definit en general.h de 10 nanosegons. Normalment farem servir aquest valor per defecte, a menys que sapiguem exactament quin retardament té una sortida. Per implementar aquestes sortides retardades, existeixen els mètodes request_event (module, delay, data: output_s) en Real_time, i el *callback* time_event (data: output_s) en Module. Funcionen igual que els altres mètodes request_event i time_event, però l'usuari no les ha de cridar directament: serveixen únicament per generar les sortides retardades.

4.3 Classe Composite

Aquesta classe derivada de Module, des d'un punt de vista extern, funciona exactament igual que Module. Però internament el seu funcionament és molt diferent. Quan es crea un Composite, automàticament es crea un objecte Route anomenat 'iroute', que permet connectar els mòduls interns del Composite.

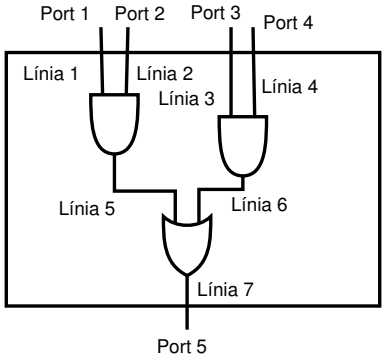


Figura 4.2: Exemple de Composite

L'usuari crearà un mòdul compost derivant de Composite. En el constructor crearà els mòduls interns, i els connectarà a 'iroute' (per això cridarà els mètodes apropiats d'aquest objecte). Després, ha de determinar les connexions entre els ports exteriors del Composite amb les línies internes de 'iroute'. Per exemple, el circuit integrat de la figura 4.2 hauria de connectar el port 1 a la línia interna 1, el port 2 a la línia 2, el port 3 a la línia 3, el port 4 a la línia 4, i el port 5 a la línia 7. Això es fa mitjançant el mètode compo_set_line(port,line). S'ha de notar que la connexió dels ports amb les línies externes es independent d'això, i es fa des de fora del Composite, però el resultat

final és que les línies internes i externes estan connectades. El mètode change_event, que ja no es pot redefinir (com passava en la classe Module), s'encarrega únicament de propagar els canvis entre les línies internes i externes.

4.4 Classe Route

La classe Route s'encarrega de la connexió dels mòduls entre si. Conté un conjunt de línies a les que els mòduls es connecten, a través dels seus ports. En cada línia normalment hi ha un mòdul connectat amb un port configurat com sortida, i un o més mòduls connectats amb un port configurat com entrada.

El programa usuari crea un objecte Route, que representa la placa del circuit emulat. En el constructor especifica quantes línies hi ha. A continuació, determina per a cada port de cada mòdul, a quina línia està connectat. Això es fa mitjançant el mètode set_line. A continuació presentem un exemple per a un circuit consistent en una font d'alimentació, un multiplexador, i un LED. Els símbols que comencen per ID_ són els identificadors dels mòduls, i els símbols que comencen per L_ són els identificadors de les línies. L'usuari pot crear aquests identificadors fàcilment fent servir *enum*'s.

```
Route route( NUM_LINES, NUM_MODS );
route.set_line( ID_SUPPLY, 0, L_GND );  // Ground
route.set_line( ID_SUPPLY, 1, L_VDD );  // Vdd
route.set_line( ID_MUX,    0, L_I0 );
route.set_line( ID_MUX,    1, L_I1 );
route.set_line( ID_MUX,    2, L_S );
route.set_line( ID_MUX,    3, L_Z );
route.set_line( ID_LED,    0, L_Z );
```

Després el programa usuari crea els mòduls. Aquests automàticament es registren en l'objecte Route, per tal d'integrar-se en la placa emulada. Per això fan servir el mètode register_module de la classe Route.

Les operacions get i set serveixen per obtenir i canviar l'estat d'una línia del Route. Es fan servir en la implementació dels mètodes input i output de la classe Module.

La resta de mètodes de la classe Route serveixen per depurar. Els mètodes set_watch i clear_watch permeten activar o desactivar la vigilància de canvis d'estat en una línia. Si la vigilància està activada, s'imprimeixen els canvis en la sortida estàndard, junt al nom de la línia, que es configura amb el mètode set_line_name.

A part de l'objecte Route creat per l'usuari, també existeix un objecte Route dintre de cada objecte Composite. Com ja s'ha vist, això es degut a que un Composite internament funciona com un circuit independent (és l'equivalent a un circuit integrat). Quan l'usuari crea un Composite nou, heretant de la classe Composite, ha de seguir els mateixos passos que s'han descrit, però l'objecte Route ja està creat pel constructor de la classe Composite.

4.5 Classe Line_mod_set

Cada línia d'un objecte Route té associat un objecte Line_mod_set, que no és més que un conjunt de parells mòdul-port. Representa quins mòduls estan connectats a la línia i mitjançant quin port. Aquesta classe només es fa servir en la implementació de la classe Route.

El mètode add permet afegir un parell mòdul-port. Els mètodes begin_enum, more_elements, i next_element permeten recorre el conjunt. Això permet als objectes Route activar els mètodes change_event de tots els objectes Module que estan connectats a la línia que ha canviat d'estat. El mètode has_module_port permet trobar si existeix un parell mòdul-port en el conjunt. Només es fa servir per part de Route per detectar que un mòdul no estigui connectat a varies línies mitjançant el mateix port (en cas de detectar-ho, mostra un error).

4.6 Classe Real_time

Només existeix un objecte Real_time per a tota l'aplicació, anomenat 'rt'. Tots els mòduls fan servir aquest objecte per tenir una referència del moment en que es troben (mesurat en nanosegons des de l'inici de la simulació), i solicitar esdeveniments futurs. D'altra banda, el programa usuari de la llibreria fa servir l'objecte 'rt' per iniciar la simulació, executar-la pas a pas, executar-la fins a un moment donat, o detenir-la.

Un esdeveniment futur es programa mitjançant els mètodes request_event. La diferència entre els dos mètodes request_event és que un té el paràmetre data de tipus u64, i l'altra el té de tipus output_s. El primer es fa servir lliurement pels mòduls d'usuari, mentre que el segon es fa servir internament per la classe Module, per generar sortides retardades. Ja s'ha parlat en detall d'això en la descripció de la classe Module, en la secció 4.2.

El moment actual s'obté mitjançant el mètode get_time de l'objecte 'rt'. La simulació s'inicia cridant el mètode run. El mètode no retorna fins que l'usuari no prem Control+C. També es pot executar la simulació pas a pas: el mètode run_step executa només el següent esdeveniment (sortida retardada o esdeveniment d'usuari). El mètode run_until executa fins al nanosegon especificat.

Si en qualsevol moment no queden esdeveniments pendents, la simulació s'atura avisant de que el circuit s'ha estabilitzat. En un microcontrolador això no pot passar, ja que el *clock* principal o l'oscilador extern al qual està connectat programa esdeveniments contínuament.

Capítol 5

CPU

5.1 Registres de la CPU

Els microcontroladors de la família MSP430 inclouen una CPU de 16 bits, amb 16 registres interns d'aquesta mida. Els registres s'anomenen R0, R1... R15. Els quatre primers tenen funcions específiques:

- R0 és el comptador de programa (PC). Sempre és parell.
- R1 és el punter de pila (SP). Sempre és parell.
- R2 és el registre d'estat (SR), i també és generador de constants.
- R3 és un altre generador de constants.

Els altres 12 registres són d'ús lliure. El registre SR inclou els següents bits:

- V: Overflow. Canvi de signe no desitjat.
- SCG1,SCG0: System Clock Generator.
- OscOff: Oscillator Off.
- CPUOff: CPU Off.
- GIE: Global Interrupt Enable.

- N: Negative. El resultat d'una operació és negatiu (MSB actiu).
- Z: Zero. El resultat d'una operació és zero.
- C: Carry. Suma desbordada. Inversa de Z en algunes operacions lògiques.

Els bits V, N, Z, i C es modifiquen com efecte secundari de les operacions aritmètiques i lògiques. Els bits SCG1, SCG0, OscOff i CPUOff defineixen els diferents modes de baix consum (Low Power Modes) en que és capaç de treballar el microcontrolador (parlarem d'això en el capítol 6.5). El bit GIE permet desactivar les interrupcions globalment (les interrupcions es poden activar o desactivar individualment, però si aquest bit està a zero, totes estan desactivades, excepte les no-enmascarables).

5.2 Modes d'adreçament

Existeixen cinc modes d'adreçament:

1. Directe: S'accedeix directament al registre. Exemple: MOV R4,R5
2. Indexat: Es llegeix un word d'extensió que se suma al contingut del registre. El resultat es tracta com una adreça de memòria. Exemple: SXT 24(R6)
3. Indirecte: S'accedeix a l'adreça de memòria indicada en el registre. En instruccions de dos operands, només és valid per a l'origen. Per a la destinació, es pot emular amb el mode indexat amb offset 0. Exemple: MOV @R10,0(R11)
4. Indirecte amb post-increment: Igual que indirecte, però el registre s'incrementa després de llegir l'operand, i abans de processar la destinació en instruccions de dos operands. L'increment és en dues unitats en mode word, i en una unitat en mode byte, excepte si el registre utilitzat és PC o SP, que sempre s'incrementen en dues unitats. Exemple: PUSH @R7+
5. Absolut: Es llegeix un word d'extensió que es tracta com una adreça de memòria. Aquest mode es codifica com indexat amb el registre SR.

Els diferents modes d'adreçament permeten accedir a les constants generades pels registres R2 i R3. Això permet reduir la mida del programa i augmentar la velocitat d'execució, ja que les sis constants més comunament utilitzades no requereixen un word d'extensió. Aquestes combinacions són:

Registre	Mode	Tractament
R2	Directe	Accés normal a R2 (status register)
R2	Indexat	Adreçament absolut (&<location>)
R2	Indirecte	Constant #4
R2	Postincrement	Constant #8
R3	Directe	Constant #0
R3	Indexat	Constant #1
R3	Indirecte	Constant #2
R3	Postincrement	Constant #-1

5.3 Instruccions

5.3.1 Formats d'instrucció

Hi ha tres formats d'instrucció, tots de 16 bits, encara que si fan servir el mode d'adreçament indexat poden tenir una o dues paraules d'extensió (de 16 bits cada una). Els tres formats són:

1. Instruccions d'un operand: Tenen un *opcode* de 3 bits, 1 bit per a indicar si treballa en mode byte o word, 2 bits per a indicar el mode d'adreçament, i 4 bits per a referenciar el registre utilitzat.

2. Instruccions de dos operands: Tenen un *opcode* de 4 bits, 1 bit per a indicar si treballa en mode byte o word, 2 bits per a indicar el mode d'adreçament de l'origen, 4 bits per a referenciar el registre utilitzar per a l'origen, 1 bit per a indicar el mode d'adreçament de la destinació, i 4 bits per a referenciar el registre utilitzat per a la destinació.

3. Instruccions de salt relatiu: Tenen un *opcode* de 3 bits, que defineix la condició de salt, i un *offset* amb signe respecte a PC de 10 bits, que indica quants words saltar si la condició es compleix (l'*offset* es multiplica per dos). Permeten saltar entre -1024 i 1022 bytes.

Hi ha 27 instruccions. Algunes poden funcionar en mode byte, cosa que s'indica amb el sufix ".B". Per exemple, "MOV.B R4,R5" copia el byte baix del registre R4 al byte baix del registre R5.

5.3.2 Instruccions d'un operand

Les instruccions d'un operand són:

RRC(.B)	Rotació a la dreta amb el carry.
RRA(.B)	Desplaçament aritmètic a la dreta.
SWPB	Intercanviar els bytes.
SXT	Extendre el signe del byte baix a tot el word.
PUSH(.B)	Introduir l'operand a la pila.
CALL	Introduir PC a la pila i saltar a l'adreça indicada per l'operand.
RETI	Finalitzar rutina de servei a interrupció, traient SR i PC de la pila.

5.3.3 Instruccions de dos operands

Les instruccions de dos operands són (totes accepten el sufix .B):

MOV	Copiar origen a destinació.
ADD	Sumar origen a destinació.
ADDC	Sumar origen i carry a destinació.
SUB	Restar origen a destinació.
SUBC	Restar origen a destinació, fent servir el carry com .not.borrow.
CMP	Comparar (equivalent a SUB però sense guardar el resultat).
DADD	Sumar origen i carry a destinació, en codi BCD.
AND	AND bit a bit.
BIT	Fer un AND sense guardar el resultat.
BIC	Rentar bits de la destinació actius en origen. No actualitza flags.
BIS	Activar bits de la destinació actius en origen (OR). No actualitza flags.
XOR	XOR bit a bit.

5.3.4 Instruccions de salt relatiu

Les instruccions de salt relatiu són:

JNE/JNZ	Saltar si Z==0.
JEQ/JZ	Saltar si Z==1.
JNC/JLO	Saltar si C==0.
JC/JHS	Saltar si C==1.
JN	Saltar si N==1.
JGE	Saltar si N==V.
JL	Saltar si N!=V.
JMP	Saltar incondicionalment.

5.3.5 Instruccions emulades

Hi ha un bon nombre d'instruccions que no existeixen en codi màquina, sinó que l'ensamblador les tradueix a altres instruccions. Tot i que no ens interessen per al desenvolupament de l'emulador, cal tenir-les en compte ja que els depuradors (GDB, IAR...) les representen així. Les més importants, i les seves equivalents, són:

NOP	No operation: MOV R3,R3
POP dst	Treure capçalera de la pila: MOV @SP+,dst
BR dst	Salt absolut: MOV dst,PC
DINT	Deshabilitar interrupcions: BIC #8,SR
EINT	Habilitar interrupcions: BIS #8,SR
RLA dst	Desplaçament aritmètic a l'esquerra: ADD dst,dst
RLC dst	Rotació a l'esquerra amb el carry: ADDC dst,dst

TST dst Comprovar si és zero: CMP #0,dst

INC dst Incrementar: ADD #1,dst

DEC dst Decrementar: SUB #1,dst

5.4 Estats de la CPU

5.4.1 Funcionament general

El número de cicles de rellotge que triga en executar-se una instrucció depèn del tipus d'instrucció, i dels modes d'adreçament utilitzats, entre d'altres coses. Emular el número de cicles de rellotge utilitzats és part fonamental d'aquest projecte. A més, s'ha intentat simular exactament el que fa la CPU en cada un d'aquests cicles. Això és necessari per a la correcta comunicació amb els altres mòduls a través dels busos del sistema (per exemple, només podem fer una lectura/escriptura per cada cicle de rellotge).

De l'estudi en profunditat de les especificacions de la CPU, s'han identificat o deduït una sèrie d'estats en que es pot trobar la CPU en cada *tick* del rellotge. En la figura 5.1 podem veure un diagrama d'estats de la CPU.

Els estats d'aquest diagrama representen cicles de rellotge. Un *tick* de rellotge provoca una transició. Les etiquetes de les transicions tenen la forma "condició / acció". En tots els ticks es provoca alguna de les transicions possibles des d'un estat. Si no hi ha condició, és perquè només hi ha una alternativa. Els cercles negres són pseudo-estats que no consumeixen cicle de rellotge: serveixen només per simplificar el diagrama.

Aquest diagrama té l'avantatge de que permet la programació immediata de la màquina d'estats de la CPU, ja que es veuen clarament totes les accions a realitzar sempre que s'entra o surt d'un estat, i en transicions concretes.

Quan el microcontrolador comença a rebre energia, però encara no té un voltatge suficient, la circuitreria de reset manté actives les línies POR (Power On Reset), i PUC (Power Up Clear). Aquesta última manté la CPU en l'estat inicial, PUC, fins que el voltatge és suficient. Més tard podem tornar a aquest estat si el senyal PUC es torna a activar (s'ha provocat un reset).

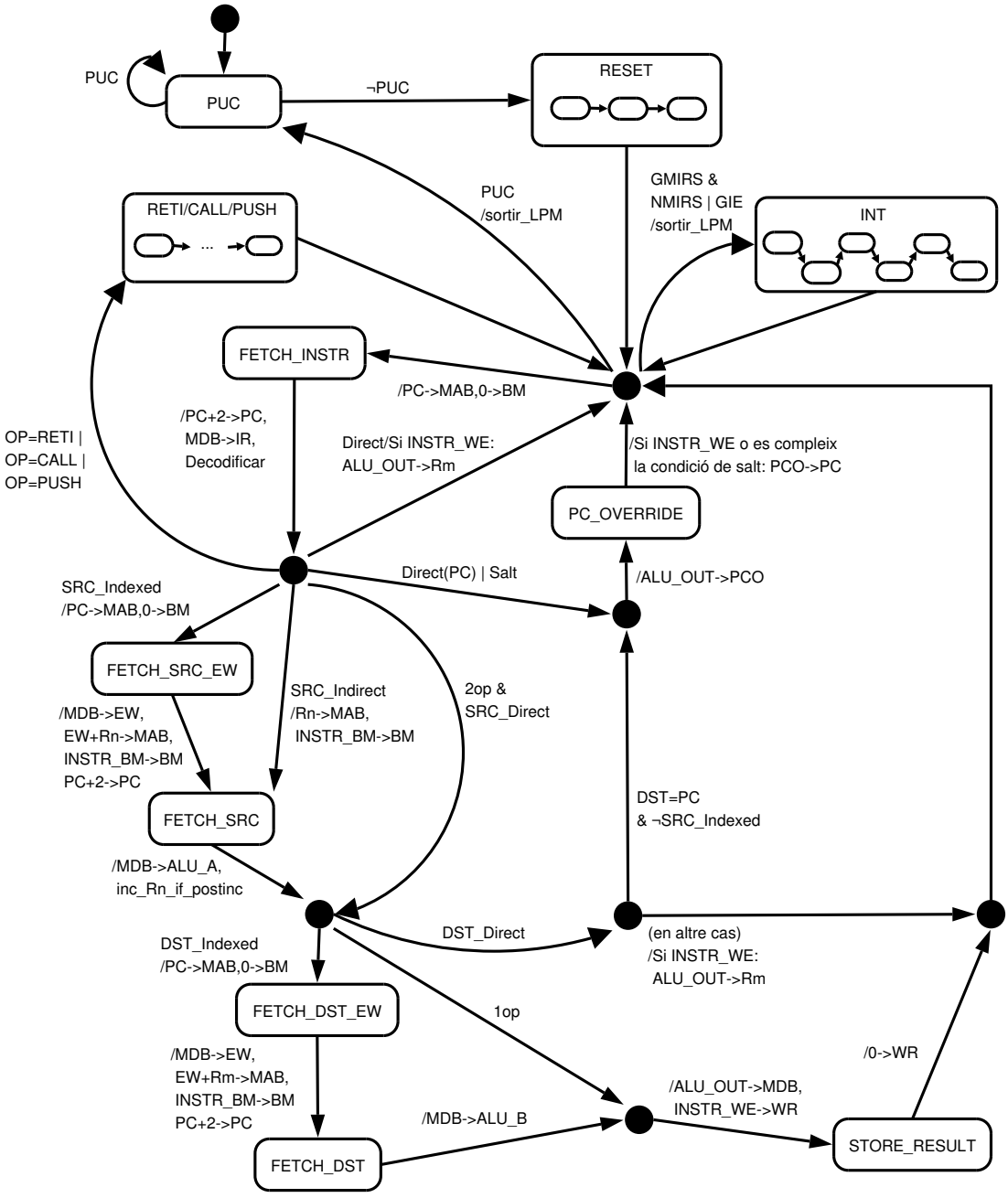


Figura 5.1: Diagrama d'estats de la CPU

Quan el senyal PUC baixa, passem a l'estat compost RESET, que com veiem, consumeix tres cicles (té tres subestats). També podem veure que abans d'entrar en l'estat FETCH_INSTR, es comprova si s'ha d'atendre una interrupció, i en tal cas entrem en l'estat compost INT. Parlarem del reset i de les interrupcions en la secció 5.4.3.

L'estat central del diagrama és FETCH_INSTR. Representa que la CPU està fent el *fetch* de la pròxima instrucció. Abans d'entrar en aquest estat sempre s'introdueix el contingut del registre PC en el MAB (Main Address Bus). D'aquesta manera el mòdul de memòria ROM (Flash) o RAM (aquest microcontrolador permet execució des de RAM) posarà la següent instrucció en el MDB (Main Data Bus).

El següent cicle de rellotge sempre provoca la lectura del MDB al IR (Instruction Register, un registre intern on s'emmagatzema la instrucció actual), l'increment del PC (de manera que apunti a la següent instrucció o paraula d'extensió), i la decodificació de la instrucció. En aquest emulador, la decodificació consisteix en extreure informació bàsica com el tipus d'operació, els modes d'adreçament, els registres utilitzats, etc. Aquesta informació s'emmagatzema en una estructura anomenada "di" (decoded instruction).

A continuació podem veure que si tant l'origen com la destinació fan servir el mode d'adreçament directe (Direct), es torna a FETCH_INSTR sense consumir cap cicle addicional (aquestes instruccions només triguen un cicle en executar-se, ja que no han d'accedir a memòria). La notació que es fa servir és: Rn és el registre utilitzat per a l'origen, i Rm és el registre utilitzat per a la destinació. En instruccions d'un sol operand, Rm coincideix amb Rn. ALU_OUT és el resultat de fer l'operació demanada (sortida de la unitat lògico-aritmètica).

Podem veure que en aquest cas, el resultat s'introdueix directament en Rm, però només si INSTR_WE està actiu. INSTR_WE (instruction write enable) correspon a un *flag* de l'estructura "di" que indica si en la decodificació de la instrucció s'ha decidit que el resultat s'ha d'emmagatzemar en la destinació o no. Per exemple, la instrucció CMP (CoMPare) fa una resta però no emmagatzema el resultat en la destinació.

Podem observar que si la destinació és el PC, es consumeix un cicle addicional. S'ha deduït que això és degut a que el PC ja s'ha actualitzat amb la sortida d'un sumador, i fa falta un altre cicle per poder canviar el selector del multiplexador que hi ha a l'entrada del PC, per seleccionar la sortida d'un teòric registre PCO (Program Counter

Override). En aquest registre és on possem temporalment el resultat de l'operació. Per tot això, fem servir l'estat PC_OVERRIDE.

En el cas anterior s'inclouen les instruccions de salt relatiu ("Salt" en el diagrama), que com es pot veure en el diagrama, consumeixen dos cicles fins i tot quan la condició no es compleix.

Si l'operand d'origen fa servir el mode indexat (SRC_Indexed en el diagrama), necessitem una paraula d'extensió (EW, Extension Word). Per això possem PC en el MAB, i entrem en l'estat FETCH_SRC_EW. En el següent cicle de rellotge entrem en FETCH_SRC, que ja s'encarrega d'obtenir de memòria el valor a utilitzar, el qual guardem en el registre intern ALU_A (una de les entrades de la unitat lògico-aritmètica).

L'acció "0->BM" indica que hem de posar un zero en el port BM de la CPU. Aquest port, connectat al bus de control, indica si treballem en mode byte (Byte Mode) o en mode word. Hem de notar que els words d'extensió sempre es llegeixen en mode word, encara que la instrucció sigui en mode byte. Per això al llegir el word d'extensió possem aquesta línia de control a zero. D'altra banda, al entrar en FETCH_SRC fem "INSTR_BM->BM". Això indica que hem de fer servir el mode indicat en la codificació de la instrucció, que tenim en un flag de l'estructura "di".

Si l'operand d'origen fa servir un dels modes indirectes (SRC_Indirect en el diagrama), es fa el mateix que en el cas anterior però sense passar per l'estat FETCH_SRC_EW. Això inclou el mode indirecte i el mode indirecte amb postincrement.

A continuació, en instruccions de dos operands, es repeteix la mateixa idea amb la destinació (estats FETCH_DST_EW i FETCH_DST), tot i que per a la destinació no existeix la possibilitat de fer servir els modes indirectes.

Si la destinació no fa servir el mode directe (incloent instruccions d'un operand), es consumeix un cicle addicional per a guardar el resultat en memòria (estat STORE_RESULT). Això es fa encara que el resultat no s'hagi d'emmagatzemar (per exemple amb la instrucció CMP). La diferència és que en aquests casos no s'activa l'escriptura en el bus de control. En el diagrama veiem INSTR_WE->WE, que vol dir activar WE (Write Enable) si i només si en la decodificació de la instrucció s'ha deduït que s'havia d'emmagatzemar el resultat.

Tota la informació sobre temporització d'instruccions s'ha obtingut de la secció 3.4.4 de la guia d'usuari del microcontrolador [4], però els estats existents i les accions realitzades en cada cicle s'han hagut de deduir, i potser no són exactes, tot i que això

és transparent per als programes executats en l'emulador, sempre i quan el resultat final de l'execució de cada instrucció sigui el mateix que en la unitat real.

5.4.2 Operacions especials

En la figura 5.1 podem veure que després de la decodificació de la instrucció, si l'operació és RETI, PUSH o CALL, entrem en un estat compost que ocupa un nombre variable de cicles. Això s'ha fet per a no complicar el diagrama amb els detalls d'aquestes operacions especials, detalls que exposem en aquesta secció.

5.4.2.1 Instrucció RETI

Com que el funcionament d'aquesta instrucció és pràcticament idèntic a executar "POP SR" i després "POP PC", s'han aprofitat els estats corresponents al funcionament general de la CPU per a implementar aquesta instrucció. Sempre consumeix cinc cicles:

1. Fetch de la instrucció (estat FETCH_INSTR). La instrucció es decodifica com "POP SR", que és un alias de "MOV @SP+,SR".
2. Fetch de l'origen (estat FETCH_SRC). Es llegeix la capçalera de la pila, s'incrementa SP, i es guarda el valor llegit en el registre SR.
3. En comptes de tornar a FETCH_INSTR, com correspondria a la instrucció "POP SR", s'entra en un retard d'un cicle (estat RETI_DELAY), probablement degut al postincrement de SP, i es canvia la decodificació de la instrucció per la corresponent a "POP PC".
4. Fetch de l'origen (estat FETCH_SRC). Es llegeix la capçalera de la pila, s'incrementa SP, i es guarda el valor llegit en el registre SP.
5. Com que la destinació és el registre PC, es consumeix un cicle addicional (estat PC_OVERRIDE).

5.4.2.2 Instrucció PUSH

Normalment una instrucció d'un operand llegeix l'operand d'un registre o posició de memòria i emmagatzema el resultat en el mateix lloc. La instrucció PUSH segueix aquest esquema, amb dues diferències:

1. En sortir de l'estat FETCH_INSTR, i abans de llegir l'operand, entrem en l'estat PUSH_DEC_SP, durant el qual decrementem el SP. Això implica consumir un cicle addicional.

2. En comptes d'emmagatzemar el resultat en el mateix lloc d'on s'ha llegit l'operand, s'ha d'emmagatzemar en la nova capçalera de la pila. Això fa que entrem en l'estat STORE_RESULT inclús quan el mode d'adreçament és directe. També fa que possem en el MAB el contingut de SP. Aquesta diferència obliga a consumir un cicle addicional respecte a l'esquema general quan es fa servir el mode d'adreçament directe.

5.4.2.3 Instrucció CALL

Aquesta instrucció és semblant a PUSH, ja que equival (amb algunes diferències) a "PUSH PC" seguit de "BR dst". Les accions realitzades són:

1. Fetch i decodificació de la instrucció (estat FETCH_INSTR).

2. Si no es fa servir el mode directe, es segueixen els estats normals per a l'obtenció de l'operand (estats FETCH_SRC_EW i/o FETCH_SRC). Sigui quin sigui el mode d'adreçament empleat, l'operand s'emmagatzema temporalment en el registre PCO, el contingut del registre PC es copia a ALU_A, i es decrementa el SP.

3. Si es fa servir el mode directe o indirecte amb postincrement, existeix un retard d'un cicle (estat CALL_DELAY). Això pot ser degut a que, amb el mode directe, no es pot aprofitar el cicle de fetch de l'operand per a decrementar SP, i amb el mode indirecte amb postincrement, el cicle de fetch de l'operand s'aprofita per incrementar el registre utilitzat, i això ho deu fer el mateix sumador que ha de decrementar SP.

4. Emmagatzemar PC en la pila (estat STORE_RESULT). Podem aprofitar l'estat STORE_RESULT de la mateixa manera que amb la instrucció PUSH.

5. En sortir de l'estat STORE_RESULT, en comptes de tornar a l'estat FETCH_INSTR com faríem seguint l'esquema general, entrem en l'estat PC_OVERRIDE. Com que havíem guardat temporalment l'operand en el registre PCO, això provocarà el salt a la subrutina.

5.4.3 Interrupcions i reset

El sistema d'interrupcions del MSP430 és una variació de l'esquema "Daisy Chain" [8]. El podem observar en la figura 5.2, extreta de la guia d'usuari del microcontrolador, on també podem veure les possibles fonts de reset.

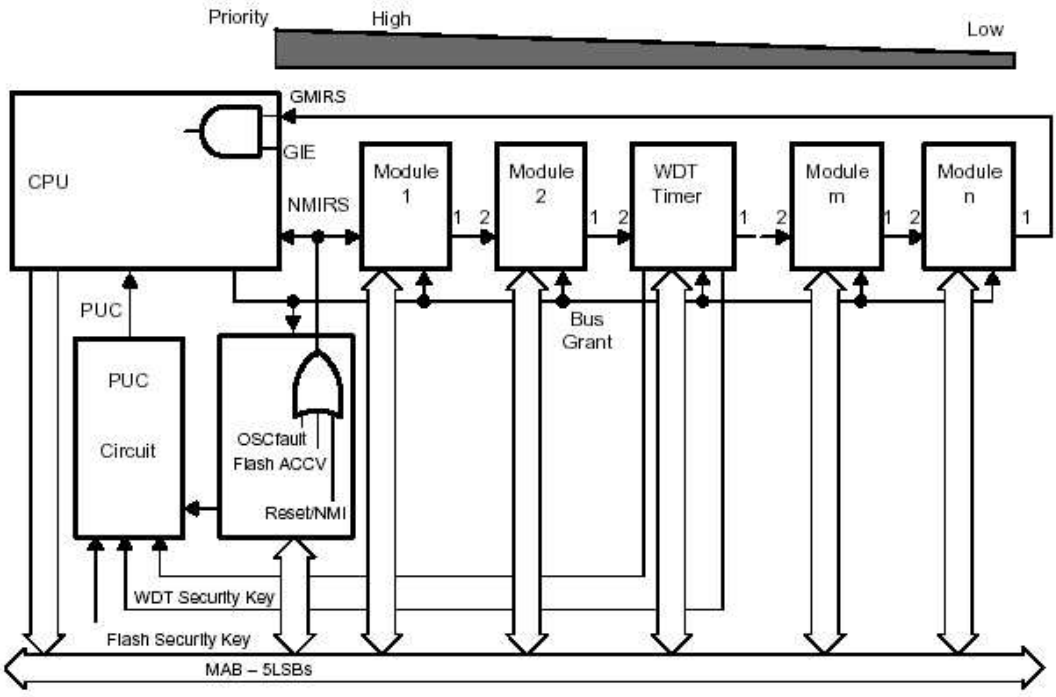


Figura 5.2: Esquema d'interrupcions i reset

Com veiem, existeixen tres tipus d'interrupció:

1. Interrupció enmascarable. Es produeix quan s'activa el senyal GMIRS (suposadament, General Maskable Interrupt Request), i està actiu el bit GIE (General Interrupt Enable) en el registre d'estat (SR). El senyal GMIRS s'activa quan qualsevol dels perifèrics solicita una interrupció. Qualsevol perifèric sap si algun perifèric més prioritari (més proper a la CPU), ha solicitat una interrupció. En tal cas, ignorarà el senyal "Bus Grant".

2. Interrupció no-enmascarable. Es produeix quan s'activa el senyal GMIRS, i està actiu el senyal NMIRS (suposadament, Non-Maskable Interrupt Request). El senyal NMIRS és produït per un mòdul auxiliar, al qual hem anomenat *nmi_gen* (descrit en la secció 6.4). Aquest senyal activa el senyal GMIRS en propagar-se per tota la cadena de perifèrics.

3. Reset del sistema. Es produeix quan s'activa el senyal PUC. Veurem aquest senyal i les condicions que el produeixen en la secció 6.1 (mòdul POR).

Quan es produeix una interrupció del tipus 1 o 2, es produeixen les següents accions:

1. Es desactiva qualsevol mode de baix consum.

2. Es completa l'execució de la instrucció actual, si ja havia començat.

3. Entrem en l'estat INT_START. Decrementem el SP i entrem en l'estat INT_DEC_SP.

4. Guardem el PC en la capçalera de la pila, i decrementem SP. Entrem en l'estat INT_STORE_PC.

5. Guardem el SR en la nova capçalera de la pila. Desactivem tots els bits del registre d'estat excepte SCG0. Entrem en l'estat INT_STORE_SR.

6. Posem l'adreça base del vector d'interrupcions en el bus d'adreces. Aquesta adreça és 0xFFE0. Activem la línia INTBG (INTerrupt Bus Grant, "Bus Grant" en la figura 5.2). Això provoca que el perifèric més prioritari dels que estan solicitant interrupció possi en el bus d'adreces els cinc bits baixos de l'adreça del vector d'interrupció corresponents a aquell perifèric. Entrem en l'estat FETCH_VECTOR.

7. El mòdul Flash posa l'adreça continguda en el vector d'interrupció en el bus de dades. La CPU llegeix aquesta adreça i la introdueix en el registre PCO. Entrem en l'estat PC_OVERRIDE.

8. Copiem el contingut de PCO a PC, possem el contingut de PC en el bus d'adreces, i entrem en FETCH_INSTR. Això executarà la primera instrucció de la rutina d'atenció a la interrupció.

Podem veure que una interrupció sempre té una latència de 6 cicles. Quan comença a executar-se la ISR, GIE està desactivat (ja que s'han desactivat tots els bits de SR excepte SCG0). Per tant, no existiran noves interrupcions enmascarables fins que finalitzi la rutina (RETI recupera el SR). Tampoc existiran noves interrupcions no-enmascarables, ja que el mòdul nmi_gen desactiva els "enable" individuals de les fonts de NMI. L'usuari ha de tornar-les a activar abans del RETI.

Quan es produeix un reset (senyal PUC), es produeixen les següents accions:

1. La CPU entra en l'estat PUC, on roman fins que el senyal es desactiva. Si la font de reset és interna, això serà només un cicle.

2. Quan el senyal PUC es desactiva, entrem en l'estat RESET.

3. Possem 0xFFFE en el bus d'adreces. Aquesta és l'adreça del vector d'interrupció corresponent al reset, i és on s'indica l'adreça on comença el programa. Entrem en l'estat INT_FETCH_VECTOR, i continuem com una interrupció normal.

La latència provocada per un reset és de 4 cicles (PUC, RESET, INT_FETCH_VECTOR, PC_OVERRIDE).

5.5 Prova del mòdul

S'ha dissenyat una prova consistent en un circuit amb un rellotge de baixa velocitat, una CPU i un mòdul de memòria RAM, on es carrega un programa que posa a prova totes les instruccions en diverses situacions. D'algunes d'aquestes situacions s'ha pogut deduir l'ordre en que s'havien de realitzar les accions en la màquina d'estats.

En el codi de la prova existeix una taula amb totes les instruccions del programa en forma de codi màquina, el número de cicles que ha de consumir la instrucció, el registre o posició de memòria modificat per la instrucció, el nou valor esperat en aquest registre o posició de memòria, el nou valor esperat en el registre d'estat (SR), i la pròxima instrucció de la taula que s'hauria d'executar.

A continuació es mostra la definició d'aquesta taula, i les seves primeres entrades. El contingut complet es troba en el fitxer tests/test_cpu_alone.cpp. Les entrades amb el comentari "EW" corresponen a paraules d'extensió (Extension Word) de la instrucció que les precedeix.

```
/* Programa de prova, amb resultats esperats per a cada instrucció. */
static struct inst_s {
        u16    instr;   /* instrucció en codi màquina */
        int    cycles;  /* número de cicles que ha de trigar */
        int    dst;     /* destinació (positiu: RAM, negatiu: Registre) */
        int    res;     /* resultat esperat en destinació */
        int    sr;      /* estat esperat del registre d'estat */
        int    next;    /* índex de la següent instrucció a executar */
} inst[] = {
        { 0x4302, 1,    -2, 0x0000, 0x0000,  1 },  /* 0.  mov #0{CG2},SR */
        { 0x403f, 2,   -15, 0x27b4, 0x0000,  3 },  /* 1.  mov #0x27b4,r15 */
        { 0x27b4, 0,     0,      0,      0,  0 },  /* 2.  EW */
        { 0x108f, 1,   -15, 0xb427, 0x0000,  4 },  /* 3.  swpb r15 */
        { 0x108f, 1,   -15, 0x27b4, 0x0000,  5 },  /* 4.  swpb r15 */
        ...
```

El programa s'ha executat pas a pas en el microcontrolador real, i s'han anotat els resultats de l'execució de cada instrucció. Aquests resultats s'han introduït en la taula de la prova, i s'ha desenvolupat el mòdul de manera que el resultat de la prova sigui satisfactori.

El flux de la prova és el següent:

1. Crear el circuit.
2. Carregar el programa en el mòdul RAM.
3. Executar el circuit durant 4 milisegons (temps del simulador, no del món real), és a dir, 4 cicles de rellotge (el rellotge té una freqüència de 1kHz). Això és el que triga en executar-se l'operació de reset.
4. Per a cada instrucció:
   (a) Verificar que la CPU està en fase de fetch (estat FETCH_INSTR).
   (b) Verificar que el registre PC conté l'adreça de la següent instrucció a executar (segons el camp "next" de la taula).
   (c) Executar el circuit durant N milisegons (N cicles de rellotge), on N és el número de cicles especificats en la taula.
   (d) Verificar que el registre o posició de memòria indicat a la taula té el valor esperat.
   (e) Verificar que el registre d'estat té el valor esperat.

Si el número de cicles que ha consumit una instrucció és incorrecte, la comprovació 4.a fallarà. Si la longitud d'una instrucció és incorrecta, o una instrucció de salt no funciona bé, la comprovació 4.b fallarà. Si l'operació lògico-aritmètica, el fetch dels operands, o l'escriptura del resultat són incorrectes, la comprovació 4.d fallarà. Si els flags del registre d'estat no s'actualitzen correctament, la comprovació 4.e fallarà.

5.6 Programació del mòdul

Com ja s'ha comentat, la programació es pot fer directament des del diagrama de la figura 5.1, implementant una màquina d'estats. El mòdul CPU és una subclasse de Module, i les transicions tenen lloc quan el mòdul rep un missatge (mitjançant el mètode change_event) indicant que hi ha un *tick* del rellotge.

En rebre aquest missatge, es crida el mètode privat state_machine. Aquest mètode consisteix en un *switch* que, segons quin sigui l'estat actual, fa la transició al nou estat, tot portant a terme les comprovacions i accions pertinents. Els estats són els membres de l'enum *cpu_state_t*, definit en cpu.hpp.

Una de les accions més importants és la decodificació de la instrucció. En el mètode *decode_instruction* es decideix quin dels tres formats possibles té la instrucció actual, i en funció d'això es crida a *decode_1op_instruction*, *decode_2op_instruction*, o *decode_jump_instruction*. Aquests mètodes plenen l'estructura "di" de la qual ja hem parlat:

```
/* instrucció decodificada */
struct {
        op_t       op;            /* operació */
        op_type_t  op_type;       /* tipus d'operació */
        bool       byte_mode;     /* mode byte (true) o word */
        bool       write_enable;  /* escriure el resultat */
        int        src_reg;       /* registre origen */
        int        dst_reg;       /* registre destinació */
        sam_t      src_addr_mode; /* mode d'adreçament src */
        dam_t      dst_addr_mode; /* mode d'adreçament dst */
        int        jump_offset;   /* offset en salt relatiu */
} di;
```

Aquesta estructura es fa servir durant tots els cicles que dura la instrucció. El mètode *decode_instruction* també s'encarrega de tractar els generadors de constants, vistos en la secció 5.2, i que modifiquen els modes d'adreçament originals.

Un altre mètode molt important és *alu_process*. S'han suposat tres registres interns anomenats *alu_a*, *alu_b* i *alu_out* (són atributs de la classe Cpu), connectades respectivament a les entrades i la sortida de la unitat lògico-aritmètica (ALU). Durant els estats de *fetch* es carreguen els registres *alu_a* i *alu_b*, i en el moment de guardar el

resultat es crida el mètode *alu_process*, que calcula el valor de *alu_out*, i actualitza els *flags* del registre d'estat.

També és important el mètode *check_jump_condition*, cridat quan s'ha de decidir si una condició de salt es compleix.

Existeix una bona quantitat de detalls que no s'han tingut en compte en el diagrama d'estats per a no sobrecarregar-lo. A continuació es descriuen alguns d'aquests detalls:

- Quan la destinació és PC o SP, rentem el bit menys significatiu del registre.
- Quan la destinació és R3, es descarta el resultat.
- Quan la destinació és SR, els bits d'aquest registre que defineixen els modes de baix consum (i per tant, com veurem més endavant, el funcionament del mòdul basic_clock) es propaguen a les línies corresponents.
- En el mètode change_event, a banda de detectar els ticks del rellotge principal, també detectem si s'activa la línia de petició d'interrupció (GMIRS) o reset (PUC). Si la petició s'accepta, desactivem els modes de baix consum. Això és necessari perquè en el moment de la petició la CPU podria estar detinguda (si el mode de baix consum ho requereix). Amb la CPU en marxa, dins la màquina d'estats atendrem la petició.
- El mode d'adreçament absolut, que es codifica com indexat a R2 (SR), es té en compte en sortir dels estats FETCH_SRC_EW i FETCH_DST_EW. El word d'extensió es fa servir directament, sense sumar el contingut de R2, com passaria amb qualsevol altre registre.
- Amb el mode indirecte amb postincrement, PC i SP sempre s'incrementen en dues unitats.
- Les sortides en els ports WR (escriptura), BM (byte mode), AB (bus d'adreces), DB (bus de dades) i INTBG (interrupt bus grant), es fan amb uns retards adequats per a que les escriptures i reconeixements d'interrupció tinguin lloc sense problemes.

Capítol 6

Mòduls i perifèrics integrats

6.1 La classe Peripheral

Molts dels perifèrics o mòduls integrats tenen en comú l'existència en els mateixos de registres de configuració als quals la CPU pot accedir. Aquest registres estan mapejats en el mapa de memòria del microcontrolador, de manera que s'accedeix mitjançant els busos del sistema com si es tractés de la memòria RAM o Flash.

A més, els bits d'aquests registres tenen certs comportaments que es repeteixen, com per exemple l'estat després d'un reset, o la connexió a un port de sortida.

Per a no repetir la funcionalitat relativa a la gestió dels busos i el comportament dels bits dels registres, s'ha implementat la classe Peripheral, de la qual heretaran tots els perifèrics integrats que tinguin registres interns.

Peripheral és subclasse de Module. Els 5 primers ports d'un mòdul que hereta de Peripheral són fixos i estan configurats en el constructor de Peripheral, de manera que corresponen a:

- Bus d'adreces (16 bits, entrada).
- Bus de dades (16 bits, entrada/sortida).
- Write (1 bit, entrada).
- Power Up Clear (1 bit, entrada).

- Power On Reset (1 bit, entrada).

Els paràmetres del constructor de Peripheral inclouen, a més dels ja existents en la classe Module, els següents:

- L'adreça base del conjunt de registres (adreça del primer registre).
- El número de bytes que ocupen els registres, en total.
- Un flag que indica si es tracta d'un perifèric de 8 o de 16 bits. Això determina la manera de tractar el bus de dades en relació als registres.
- Un array amb la configuració del comportament de cada bit.

El comportament de cada bit es descriu amb una estructura que determina:

- Accessos: lectura/escriptura, només lectura, lectura sempre com 0, lectura sempre com 1, les escriptures generen un pols, etc.
- Estat inicial després de reset: no canvia, 0 després de PUC, 1 després de PUC, 0 després de POR, 1 després de POR. Per a més informació sobre les línies PUC i POR veure la secció 6.3.
- Port de sortida connectat. Pot ser nul. El valor del bit sempre es propagarà al port indicat.

La informació sobre el comportament de cada bit dels registres es pot extreure de les taules de registres existents al final de cada capítol corresponent a un perifèric, en la guia d'usuari del microcontrolador [4], excepte el port de sortida connectat, que es dedueix dels diversos esquemes de la guia. Una línia que tingui el mateix nom que un bit d'un registre és el resultat de la propagació d'aquell bit.

S'ha dissenyat un test automatitzat d'aquesta classe, del mòdul SFR (tractat en la secció següent), i del mòdul POR (tractat en la secció 6.3). S'ha triat el mòdul SFR ja que és el perifèric més senzill de tots. També es proba el mòdul POR, ja que aquest mòdul era necessari per a probar el perifèric. El test es pot trobar en el fitxer test_cpu_por_periph.cpp. És similar als tests de la CPU, en el sentit de que es carrega un programa en la RAM, i després de cada instrucció es comprova l'estat d'algun

registre o posició de memòria (en aquest cas, registres del mòdul SFR). Aquest programa, al final, força un reset (i torna a començar). D'aquesta manera es proba el funcionament del mòdul POR, i la correcta reinicialització dels registres del perifèric.

6.2 Registres de funció especial

Existeix un banc de 16 registres de funció especials (SFRs, *special function registers*), ubicats en les 16 adreces més baixes del mapa de memòria, que són utilitzats per diversos perifèrics. Només s'utilitzen alguns bits d'aquests registres. Es tracta de diversos bits dels tipus "Interrupt Enable", "Interrupt Flag" i "Module Enable". Alguns perifèrics tenen aquests bits en els SFRs en comptes de tenir-los en els propis registres interns.

El banc de SFRs s'ha implementat en la classe Sfr, hereten de Peripheral, i afegint un port per cada bit, ja que tots aquests bits es fan servir com línies en altres mòduls.

Com a cas especial, aquest mòdul permet, a través del mètode "mod_bit", la modificació directa de bits per part d'altres mòduls, sense fer servir els busos del sistema. Això s'ha fet perquè els "Interrupt Flag" han de ser activats per part dels mòduls en el moment de generar una interrupció.

Com ja s'ha mencionat en la secció anterior, aquest mòdul es proba conjuntament amb la classe Peripheral i el mòdul POR, en el test "cpu_por_periph".

6.3 Power-On Reset

El microcontrolador MSP430 té dues línies internes de reset: POR (Power On Reset) i PUC (Power Up Clear). L'activació de la línia POR provoca l'activació de la línia PUC, però no al revés.

Com ja hem vist en descriure la classe Peripheral, alguns bits dels registres dels perifèrics tornen al seu estat inicial quan es produeix un PUC, i d'altres només quan es produeix un POR. La principal raó de que un bit no es reinicii amb PUC és permetre a la CPU consultar què ha provocat el reset. Com ja hem vist en la secció 5.4.3, el reset de la CPU es produeix quan s'activa PUC (i per tant, també POR).

POR és un reset del dispositiu. Es genera en els següents casos:

- El dispositiu s'encén. En l'emulador, això vol dir que el valor de la línia d'entrada DVcc (*digital voltage*) passa de 0 a 1.
- La línia !RST/NMI passa de 1 a 0 i està configurada en mode reset. Aquesta línia està connectada a un port extern del MSP430, i permet implementar un botó de reset.

PUC és un reset provocat per POR o per alguna font de reset interna. Es genera en els següents casos:

- S'activa POR.
- Expira el *watchdog timer* i està configurat en mode *watchdog*.
- S'intenta accedir als registres del Watchdog sense la clave apropiada.
- S'intenta accedir als registres del controlador Flash sense la clave apropiada (veure secció 6.7).

El progama executat en la CPU pot provocar un reset (PUC) simplement escrivint en un registre del controlador Flash sense posar la clau correcta (0xa5) en el byte alt.

El mòdul POR és l'encarregat de generar els senyals POR i PUC. Està documentat en el capítol 2 de la guia d'usuari del microcontrolador [4]. L'emulador del mòdul s'ha implementat en la classe Por, derivada de Composite, amb els següents components:

- Por_core: Implementa la lògica de generació dels senyals.
- Por_delay: Genera un senyal intermedi que manté POR actiu durant 250 microsegons (temps del simulador de circuits). En el dispositiu real aquest temps és variable, entre 150 i 250 microsegons. L'únic requisit és que aquest temps sigui superior a un cicle de qualsevol dels rellotges del sistema, perquè els mòduls integrats puguin reaccionar al POR en el primer *tick* del rellotge.
- Delay: Es tracta d'un component electrònic estàndard. En aquest cas es fa servir per retardar la realimentació del senyal PUC a les entrades R dels latches POR i PUC.

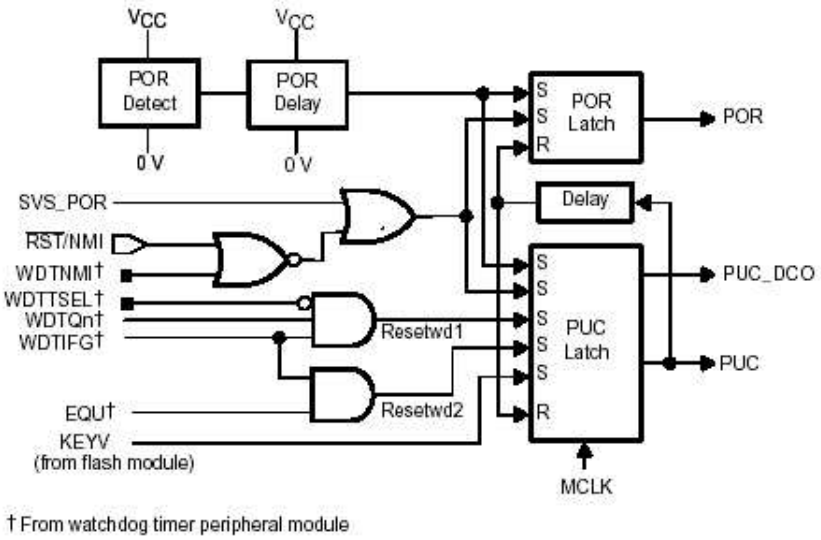


Figura 6.1: Esquema del mòdul POR

El Composite POR s'ha construït amb aquests tres components, de la mateixa manera que en la figura 6.1, extreta de la guia d'usuari. Els latches i les portes lògiques estan implementats en la classe Por_core, el mòdul "POR Delay" en la classe Por_Delay, el component "Delay" en la classe Delay del directori src/blocks, i el mòdul "POR Detect" no és necessari ja que en l'emulador el voltatge no varia gradualment (és 0 o 1).

Com ja s'ha mencionat en la secció 6.1, aquest mòdul es proba conjuntament amb la classe Peripheral i el mòdul SFR, en el test "cpu_por_periph".

6.4 Generador de NMI

S'ha identificat la necessitat de crear un mòdul auxiliar que no apareix en la documentació del dispositiu, el generador d'interrupcions no enmascarables (NMI, *Non-Maskable Interrupts*). S'ha programat directament a partir de l'esquema de la pàgina 2-8 de la guia d'usuari del microcontrolador [4].

Només cal afegir que quan la CPU accepta la interrupció (activa la línia INTBG, anomenada IRQA en l'esquema mencionat), i aquest mòdul havia generat la interrupció, s'ha de posar el vector d'interrupció NMI (0x1C) en els 5 bits baixos del bus d'adreces. Per a més informació sobre la generació i acceptació d'interrupcions, i les interrupcions no enmascarables, veure la secció 5.4.3.

6.5 Basic Clock

6.5.1 Senyals de rellotge

Aquest mòdul és l'encarregat de generar els tres senyals de rellotge del sistema:

- MCLK: Main Clock.
- SMCLK: Sub-Main Clock.
- ACLK: Auxiliary Clock.

Els tres rellotges són utilitzats per els diversos mòduls del microcontrolador. Alguns mòduls permeten al programador seleccionar (a través dels registres de configuració) quin dels tres rellotges utilitzar. La CPU utilitza només el rellotge MCLK.

El mòdul basic clock és ell mateix un perifèric amb registres interns de configuració, que donen al programador una gran flexibilitat sobre la manera de generar els senyals de rellotge.

6.5.2 Fonts de rellotge

Existeixen tres possibles fonts de rellotge:

- Oscilador de baixa freqüència (LFXT1), connectat a un cristall, oscilador o rellotge extern.
- Oscilador de baixa o alta freqüència (XT2), connectat a un cristall, oscilador o rellotge extern.
- Oscilador intern controlat digitalment (DCO, *Digitally Controlled Oscillator*). La seva freqüència és ajustable mitjançant els registres de configuració, i no requereix cap component extern.

ACLK es pot generar només a partir de LFXT1. MCLK es pot generar a partir de qualsevol de les tres fonts. SMCLK es pot generar a partir de XT2 o DCO. La freqüència

de qualsevol dels rellotges es pot dividir per 1, 2, 4 o 8, mitjançant divisors electrònics de rellotge.

En la figura 6.2, extreta de la guia d'usuari del microcontrolador, podem observar les diferents opcions de configuració, tenint en compte que totes les línies corresponen a bits dels registres de configuració, excepte CPUOFF, OSCOFF, SCG1 i SCG0, que corresponen als modes de baix consum configurats en el registre SR de la CPU.

La figura 6.2 és incompleta, ja que els senyals que provoquen la desconnexió d'algun oscilador, no son efectives en algunes condicions, documentades al llarg del capítol 4 de la guia d'usuari. Això s'ha tingut en compte a l'hora d'implementar l'emulador del mòdul.

El funcionament del DCO i els components associats és complex, però s'ha deduït, a partir de la informació del datasheet [5], una fórmula per a trobar la freqüència aproximada resultant en funció dels valors dels bits de configuració:

$$f_{DCO} = f_0 \cdot (S_R)^{RSEL} \cdot (S_{DCO})^{DCO} \cdot 2^{\frac{MOD}{32}}$$

On:
f_0 és la freqüència corresponent a RSEL=0, DCO=0, MOD=0. Aprox. 72 kHz.
S_R és un paràmetre especificat en el datasheet (*step* entre valors de RSEL). Aprox. 1.65.
S_{DCO} és un paràmetre especificat en el datasheet (*step* entre valors de DCO). Aprox. 1.12.
RSEL és el valor del grup de bits de configuració RSELx (3 bits).
DCO és el valor del grup de bits de configuració DCOx (3 bits).
MOD és el valor del grup de bits de configuració MODx (5 bits).

El valor de f_0 no està documentat. S'ha calculat el valor tal que la configuració inicial (RSEL=4, DCO=3, MOD=0) resulti en la freqüència adequada (que si que està documentada, aprox. 750 kHz).

Tenint en compte que la freqüència resultant del DCO en el dispositiu real varia molt amb la temperatura, el voltatge, i inclús entre diferents unitats del mateix model, s'ha trobat més adequat utilitzar la fórmula deduïda en comptes d'emular tots els components implicats (DC Generator, DCO, Modulador i multiplexor del modulador).

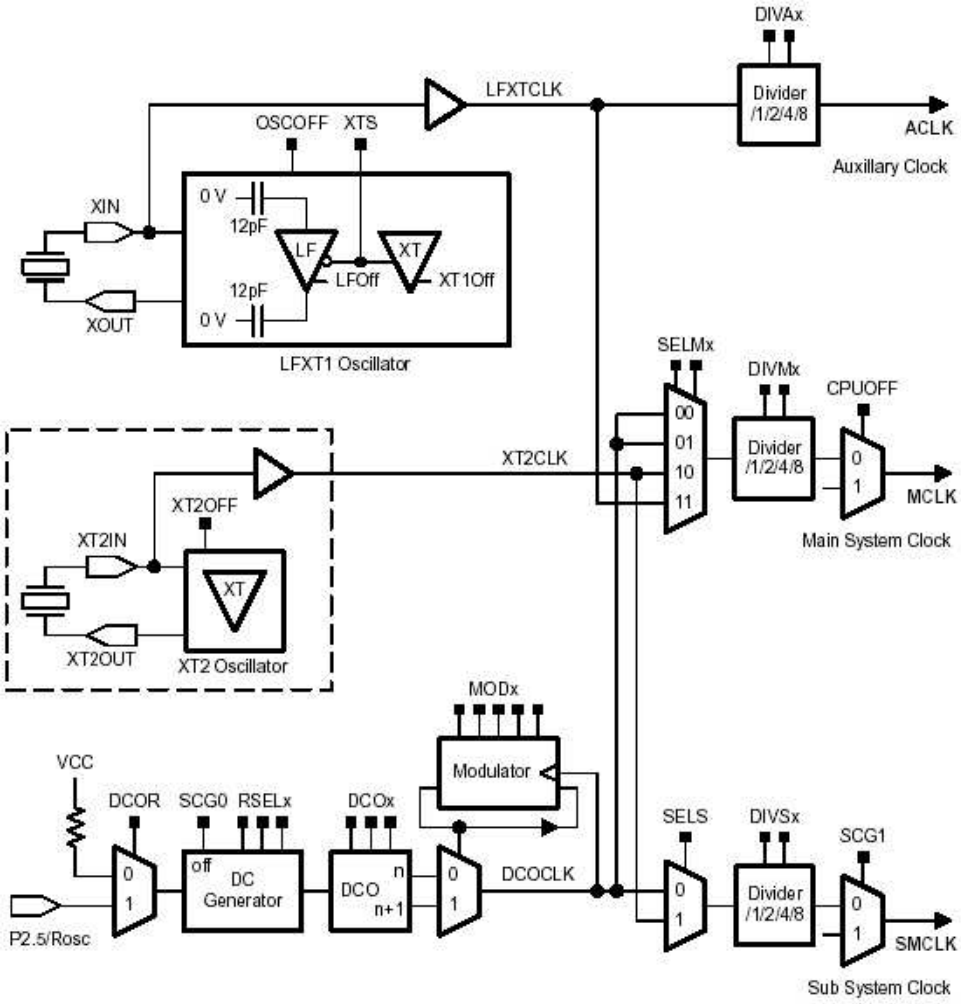


Figura 6.2: Esquema del mòdul Basic Clock

6.5.3 Modes de baix consum

Com ja hem vist, quatre dels bits de configuració del basic clock es troben en el registre SR de la CPU. Són els bits corresponents als modes de baix consum (LPMs, *Low Power Modes*), que permeten desactivar osciladors o sortides de rellotge per a estalviar energia.

Recordem que el registre SR es guarda en la pila quan la CPU comença a executar una rutina de servei a interrupció. Després, la CPU desactiva els modes de baix consum. D'aquesta manera, durant l'execució de la rutina, el sistema es troba en estat actiu. Quan finalitza la rutina, la instrucció RETI recupera el valor del registre SR de la pila (com en gaire bé qualsevol CPU). En aquell moment, gràcies a que aquests bits estan

en SR, el microcontrolador tornarà al mode de baix consum on es trobava quan es va generar la interrupció.

A més, abans de cridar a la instrucció RETI, la rutina de servei a interrupció pot modificar el valor de la posició de la pila on està emmagatzemat el valor de SR, de manera que RETI activarà el nou mode de baix consum. Una aplicació molt típica és la del següent exemple, consistent en un ”sleep” d'un segon durant el qual es redueix dràsticament el consum d'energia:

1. La CPU programa un temporitzador, alimentat amb ACLK, que provoqui una interrupció al cap d'un segon.
2. La CPU desactiva, mitjançant el registre d'estatus (SR), els rellotges MCLK i SMCLK.
3. Com que MCLK s'ha desactivat, la CPU roman adormida.
4. Quan ha passat un segon, es genera una interrupció.
5. La rutina de servei a la interrupció modifica el valor de SR guardat en la pila, de manera que estiguin desactivats els bits dels modes de baix consum.
6. RETI restaura el valor modificat. Ara MCLK està actiu.
7. Continua l'execució en la instrucció següent a la del pas 2.

Els modes de baix consum estan definits com LPM0, LPM1, LPM2, LPM3 i LPM4, de la manera que podem veure en la següent taula:

SCG1	SCG0	OSCOFF	CPUOFF	Mode	Descripció
0	0	0	0	Actiu	Tots els rellotges habilitats estan actius.
0	0	0	1	LPM0	MCLK desactivat (CPU dormida).
0	1	0	1	LPM1	Com LPM0, però pot desactivar DC Generator.
1	0	0	1	LPM2	MCLK, SMCLK i DCO desactivats.
1	1	0	1	LPM3	Com LPM3 però pot desactivar DC Generator.
1	1	1	1	LPM4	Tots els rellotges desactivats.

6.5.4 Prova del mòdul

Aquest mòdul es prova en el test automatitzat "test_bclock". El test crea un circuit amb un mòdul Basic Clock, un oscilador de 32768 Hz (*standard watch clock*), un oscilador d'alta freqüència (4 MHz), i un mòdul que conta ticks dels tres rellotges generats (MCLK, SMCLK, ACLK). També existeixen les línies corresponents als busos d'un MSP430 i als bits dels modes de baix consum.

Durant la prova, es modifiquen els registres interns i els modes de baix consum, s'executa el circuit durant 10 milisegons de temps del simulador, es comprova que els valors dels comptadors de ticks siguin els esperats, i es repeteix la prova amb una altra configuració.

Les configuracions provades són:

1. Configuració inicial del microcontrolador (després de POR/PUC).
2. Generar MCLK amb l'oscilador d'alta freqüència (SELM=2, XT2OFF=0), i canviar la freqüència del DCO posant RSEL a 0.
3. Mode de baix consum LPM0.
4. Mode de baix consum LPM1.
5. Mode de baix consum LPM2.
6. Mode de baix consum LPM3.
7. Mode de baix consum LPM4.
8. Dividir ACLK per 2, MCLK per 4, i SMCLK per 8 (DIVA=1, DIVM=2, DIVS=3), posar el DCO a màxima freqüència (RSEL=7, DCO=7, MOD=31).

En el codi font de la prova (src/test/test_bclock.cpp), es poden trobar més detalls sobre les probes i els càlculs realitzats per a trobar els valors esperats dels comptadors.

6.5.5 Programació del mòdul

Aquest mòdul està implementat en la classe Basic_clock, derivada de Peripheral, de manera que Peripheral s'encarrega de la gestió dels registres de configuració. Es defineixen ports per a les entrades i sortides als dos osciladors externs, així com quatre ports d'entrada corresponents als bits de configuració dels modes de baix consum, provinents de la CPU.

Quan es detecta un tick d'un oscilador (en el mètode change_event), es comprova si l'oscilador està habilitat (combinació adequada de bits de configuració), i si és així, es crida als mètodes tick_lfxt o tick_xt2 (segons de quin oscilador es tracti).

Els ticks del DCO es generen amb esdeveniments de temps del simulador de circuits. En el mètode time_event es comprova si el DCO està habilitat, i si és així es crida al mètode tick_dco, i es programa l'esdeveniment del següent tick.

Els mètodes tick_* comproven els selectors dels multiplexors que seleccionen la font de cada rellotge (veure la figura 6.2). Si coincideix amb la font del tick, es crida als mètodes aclk_divider, mclk_divider o smclk_divider (es propaga el tick a un o més dels divisors).

Els mètodes *_divider incrementen un comptador (hi ha un comptador per cada rellotge), i si el comptador coincideix amb 1, 2, 4, o 8 (segons la configuració del divisor en els registres del mòdul), es propaga el tick al port corresponent al rellotge generat (ACLK, MCLK o SMCLK).

6.6 Memòria RAM

La memòria RAM s'ha implementat en la classe Ram, derivada de Module. Els seus ports són:

- Bus d'adreces (16 bits, entrada).
- Bus de dades (16 bits, entrada/sortida).
- Write (1 bit, entrada). Indica que s'ha d'escriure una posició de memòria, i no llegir-la.
- Byte Mode (1 bit, entrada). Indica que s'ha de llegir o escriure en mode byte, i no word.

Els senyals corresponents a aquests ports els hem tractat amplament en el capítol 5 (CPU).

La memòria RAM en si està en l'atribut ram de la classe, consistent en un array de 2048 bytes (la mida de la RAM en aquest microcontrolador). Aquesta memòria, com en el dispositiu real, conserva el seu valor en els resets (sempre que no caigui el voltatge). Per aquesta raó, el mòdul no té ports POR i PUC, com altres perifèrics.

La implementació del mòdul és bastant senzilla. En el mètode change_event es detecta quan canvia el valor del bus d'adreces o de la línia Write. Si canvia l'adreça del bus d'adreces, i el valor de la línia Write és 0, es produeix una lectura: el valor de la posició de memòria demanada es posa en el bus de dades. Si el valor de la línia Write passa de 0 a 1, s'escriu el valor llegit del bus de dades en la posició indicada per el bus d'adreces.

Durant totes dues operacions, s'observa l'estat de la línia BM (Byte Mode). Si el mode byte està activat, es llegeix o modifica el byte corresponent a l'adreça indicada. Sinó, s'ignora el bit més baix del bus d'adreces, i es llegeix o modifica el word corresponent a l'adreça parella resultant.

No s'ha dissenyat cap prova específica per a aquest mòdul, ja que es fa servir en gairebé totes les altres proves, i es considera suficientment provat.

6.7 Memòria Flash

La memòria Flash és un tipus de memòria no volàtil, similar a la EEPROM (*Electrically Erasable Programmable Read-Only Memory*), però que només pot ser esborrada en blocs o per complet. L'estat d'un bit esborrat és 1, i l'escriptura consisteix en canviar el valor d'un bit de 1 a 0. Si cal tornar a canviar el valor d'un bit de 0 a 1, cal esborrar el bloc on està aquest bit.

El microcontrolador MSP430 inclou un controlador de memòria Flash, amb un generador de voltatge capaç de generar el voltatge necessari per a esborrar blocs i escriure bits. Els esborrats i les escriptures són controlats mitjançant els registres interns del mòdul Flash, per part del programa d'usuari que s'executa en la CPU.

El mòdul s'ha implementat en la classe Flash, derivada de Peripheral. Com que es tracta de memòria no volàtil, en el constructor s'intenta llegir el contingut de la memòria Flash d'una imatge en disc (mspflash.img). Si no existeix, el contingut serà indeterminat fins que es carregui un programa amb GDB. En el destructor, el contingut de la memòria Flash es desa en el mateix fitxer.

Per defecte el mòdul Flash està en mode lectura, en el qual presenta un comportament idèntic a la memòria RAM, però els intents d'escriptura activen el bit ACCVIFG (Access Violation Interrupt Flag), propagat a una línia connectada al mòdul NMI_GEN, i que pot generar una interrupció si està habilitada (veure secció 6.4).

El mètode change_event està sobrecarregat. Si el port modificat és un dels gestionats per la classe Peripheral, cridem el mètode change_event d'aquella classe (superclasse de Flash). Hi ha dues excepcions:

- Si la línia modificada és el bus d'adreces, i l'adreça indicada pertany a la memòria Flash (i no a un registre del controlador Flash), es fa el mateix que en el mòdul RAM, com ja hem dit. Hi ha, però, una excepció: Si el controlador Flash està ocupat (bit BUSY actiu), no és llegeix la memòria Flash, sinó que es posa el valor 0x3fff en el bus de dades. Això correspon a la instrucció "jmp PC", que deixarà la CPU en un bucle fins que el controlador deixi d'estar ocupat (si el programa està executant des de RAM, el bucle s'ha d'implementar manualment, consultant el bit BUSY d'aquest perifèric).

- Si la línia modificada és WR (Write), comprovem el següent:

    1. Si l'adreça del bus d'adreces correspon a un registre, comprovem que el byte alt del bus de dades sigui la contrasenya d'accés a la Flash (0xa5). Si ho és, propaguem la crida al mètode change_event de la superclasse (Peripheral). Si no, activem el bit ACCVIFG, com hem vist que passava quan s'intentava escriure en mode lectura.
    2. Si l'adreça correspon a la memòria Flash, es comprova l'estat del controlador Flash a partir dels bits de configuració. Es poden donar les següents situacions:
        (a) Ocupat (bit BUSY actiu). S'està processant un esborrat. S'activa el bit ACCVIFG.
        (b) Mode escriptura (bit WRT actiu). Es fa una escriptura AND (només s'escriuen els zeros, com ja hem vist). La gestió de la línia BM (Byte Mode) és idèntica que en el mòdul RAM.
        (c) Esborrat total (bits ERASE i MERAS actius). S'esborra tota la memòria Flash (es posen tots els bits a 1).
        (d) Esborrat de bloc (bit ERASE actiu). S'esborra el bloc on està el byte indicat en el bus d'adreces. Els blocs són de 512 bytes, excepte els dos primers, que són de 128 bytes, i el tercer, que és de 256 bytes.
        (e) Esborrat massiu (bit MERAS actiu). Idèntic a l'esborrat total, però no s'esborren els dos primers blocs (de 128 bytes), anomenats també "Information Memory". Cal destacar que normalment el programa d'usuari sempre es col.loca a partir del tercer bloc, de manera que es pot reprogramar el dispositiu sense perdre la "Information Memory".

En la implementació actual no s'han emulat els temps d'esborrat i escriptura. Aquestes operacions es fan de manera immediata, de manera que mai es donarà el cas a). Es proposarà en les conclusions una emulació més realista d'aquest mòdul com a treball futur.

6.8 Ports d'entrada/sortida

Existeixen 6 ports d'entrada sortida, amb 8 bits cada un, els dos primers amb capacitat d'interrupció. S'ha implementat una classe derivada de Peripheral anomenada Ioport, amb dos paràmetres que permeten crear, mitjançant l'instanciament de 6 objectes, els ports del microcontrolador:

- L'adreça base dels registres del perifèric.
- La capacitat d'interrupció (tipus bool).

Per a cada un dels 8 bits existeix un port d'entrada/sortida en el mòdul. Els bits dels registres de configuració són (canviant 'x' pel número de port i 'n' pel número de bit):

- PxIN.n: Valor llegit. Utilitzat en mode entrada. En el mètode change_event detectem el canvi en la línia, i si està configurada en mode entrada (amb PxDIR.n), modifiquem el valor d'aquest bit.
- PxOUT.n: Valor a escriure. Utilitzat en mode sortida. La classe Peripheral s'encarrega de propagar el valor del bit a la línia quan la CPU modifica el bit a través dels busos del sistema.
- PxDIR.n: Selecciona el mode (0: entrada, 1: sortida).
- PxIFG.n (només en ports 1 i 2): Flag d'interrupció. Quan s'activa, provoca una interrupció en la CPU. No s'ha implementat per falta de temps, i es proposa com a treball futur.
- PxIES.n (només en ports 1 i 2): Selecció del flanc que provoca interrupció.
- PxIE.n (només en ports 1 i 2): Habilitació de les interrupcions.
- PxSEL.n: Seleccionar la funció alternativa de la línia (per exemple, propagar una línia de rellotge interna). No s'ha implementat per falta de temps, i es proposa com a treball futur.

6.9 Mòduls no implementats

A continuació es llisten els mòduls que queden per implementar (es proposaran com a treballs futurs). La no existència d'aquests mòduls no afecta a l'execució de programes que no els facin servir explícitament:

- Multiplicador: Permet executar ràpidament operacions numèriques relacionades amb el tractament digital de senyal.
- Temporitzadors: Watchdog Timer, Timer_B7, i Timer_A3. Capaços de provocar interrupcions (o reset, en el cas del watchdog) quan un comptador arriba a cert valor. Són molt configurables.
- ADC i Comparador: Comparen voltatges. Requereix modificar el simulador de circuits digitals per a poder representar línies amb valors analògics de voltatge.
- USARTs: Permeten la comunicació serie, asíncrona (mode UART) o síncrona (mode SPI). Existeixen dues unitats.

Capítol 7

Integració

La CPU i tots els mòduls i perifèrics integrats són els components del Composite Msp430. Aquesta classe és pròpiament l'emulador del microcontrolador MSP430. La composició és la de la figura 7.1.

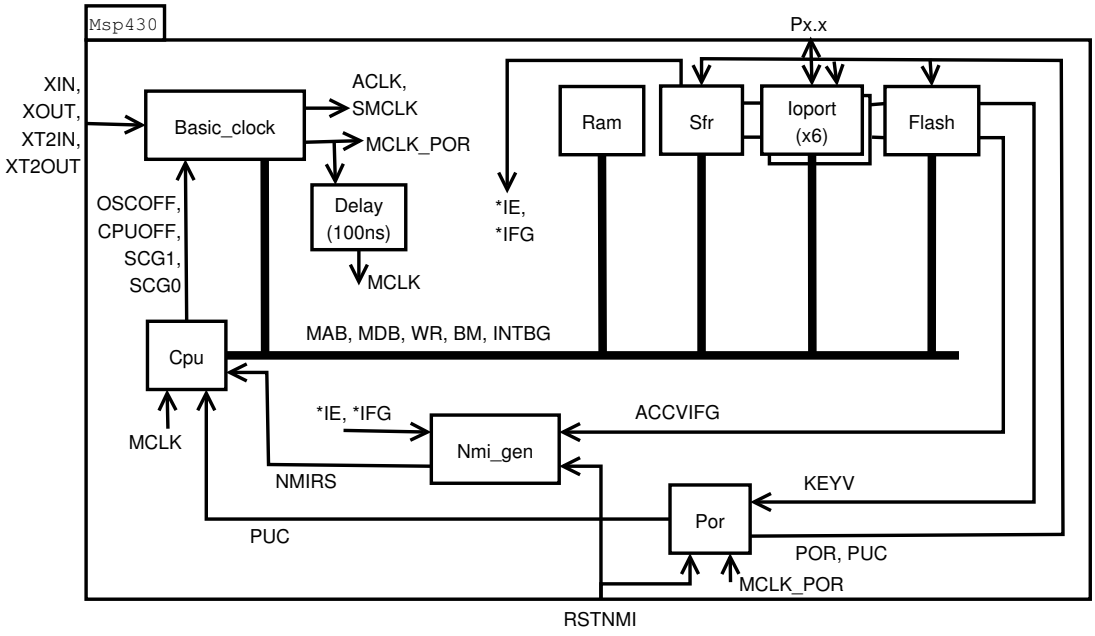


Figura 7.1: Composite Msp430

Podem observar que el senyal de rellotge MCLK està retardat per a tots els mòduls excepte per al mòdul POR. Es desconeix si el dispositiu real ho fa, però ha resultat necessari per a permetre que els mòduls (especialment la CPU) reaccionin immediatament al senyal de reset PUC. Aquest senyal és síncron amb MCLK, de manera que, sense aquest retardament, els mòduls trigarien un cicle de rellotge en reaccionar, i la latència de 4 cicles documentada per al reset no es compliria.

Capítol 8

Interfície amb GDB

8.1 Introducció

GDB suporta, amb els pegats del projecte MSPGCC, la depuració de programes sobre microcontroladors de la família MSP430. GDB es comunica via TCP o port serie amb un programa anomenat gdbproxy (també del projecte MSPGCC), que és qui realment es comunica a través del port paral.lel amb el microcontrolador (a través d'una placa conversora de port paral.lel a JTAG).

El mòdul de gdbproxy que implementa la comunicació amb el dispositiu és de codi tancat, ja que s'ha fet amb informació confidencial de Texas Instruments, però la resta del codi té una llicència tipus BSD, i per tant podem aprofitar-ho. Només hem d'implementar el mòdul de comunicació amb l'emulador.

Per a simplificar la utilització de l'emulador, s'ha optat per enllaçar gdbproxy amb l'emulador, de manera que una vegada compilat gdbproxy, aquest portarà l'emulador integrat. Llavors, la interfície entre gdbproxy i l'emulador es redueix a crides a funció.

8.2 Interfície amb gdbproxy

Hem de notar que gdbproxy està programat en C, per tant en gdbproxy no podrem instanciar l'emulador (la classe Msp430) directament. Necessitem, per tant, un mòdul programat en C++ que tingui funcions de C com a interfície externa (definides com extern "C"). Aquest mòdul s'ha anomenat emu_fet, per similituds amb el FET (*Flash Emulation Tool*, la placa de desenvolupament on es programa i depura el microcontrolador).

Les funcions de la interfície d'aquest mòdul són:

fet_init Crea un circuit amb un Msp430, una font d'alimentació, un oscilador de 32768 kHz connectat al LFXT1 del microcontrolador, i un LED connectat al port 1.0. Aquest circuit es pot modificar tant com sigui necessari, afegint osciladors, components externs (programats amb libdigisim), altres instàncies del microcontrolador (executant un programa existent en una imatge de la Flash en disc), etc. Es pot utilitzar el codi d'algun dels mòduls del directori src/blocks (com el mòdul LED) com a base per a desenvolupar altres perifèrics externs.

fet_cleanup Allibera tota la memòria utilitzada, reinicia el temps del simulador, i elimina tots els breakpoints. Es cridada cada cop que GDB desconnecta de gdbproxy.

fet_reset Provoca un reset en el microcontrolador, treient l'alimentació momentàniament.

fet_run Habilita l'execució del circuit, amb step o sense (segons un paràmetre). En mode step, el circuit executa només fins que comença la següent instrucció.

fet_wait_partial Executa el circuit durant un cert temps, o fins arribar a un breakpoint. GDB solicita repetidament a gdbproxy l'execució d'aquesta funció. El temps d'execució està limitat perquè si l'usuari polsa Control+C, s'haurà de cridar a fet_stop per a detenir l'execució. Per a donar una resposta ràpida a l'usuari, aquest temps no hauria de ser superior a un segon.

fet_stop Para l'execució del circuit. Simplement provoca que la pròxima crida a fet_wait_partial retorni immediatament indicant la parada de l'execució.

fet_read_reg Torna el valor d'un registre. Es fa a través del mètode test_reg de Msp430, que simplement propaga la crida al mètode de mateix nom de la classe Cpu.

fet_write_reg Permet la modificació d'un registre. Implementat de manera similar a l'anterior funció.

fet_read_mem Llegeix un bloc de memòria. Per cada word del bloc demanat, es crida al mètode test_mem de la classe Msp430, que comprova a quin mòdul o perifèric pertany l'adreça, i delega la lectura al mòdul corresponent.

fet_mem_write Similar a l'anterior però per a escriure. Cal destacar que l'escriptura en Flash és immediata (el temps del simulador està parat, no s'està emulant). Això representa una avantatja respecte al dispositiu real, ja que la reprogramació de la Flash és relativament lenta.

fet_add_break Afegir un breakpoint. Els breakpoints s'emmagatzemen en una llista encadenada, no hi ha límit al número de breakpoints que es poden posar. Això representa una gran avantatja respecte al microcontrolador real, que només permet dos breakpoints.

fet_remove_break Elimina un breakpoint de la llista.

fet_erase_all Esborrar tota la memòria Flash (posar tots els bits a 1). Això prové de la comanda de GDB "monitor erase all", necessària abans de carregar un programa.

Per a la detecció dels breakpoints, s'ha afegit un paràmetre opcional al constructor de la classe Cpu, consistent en un punter a funció on la CPU ha de reportar l'adreça de la següent instrucció a executar, abans d'executar-la. Aquest punter apunta a una funció estàtica d'aquest mòdul, cb_cpu_report, que comprova si l'adreça correspon a algun breakpoint, o si estem en mode step. En aquests casos, es para l'execució del circuit, i per tant fet_wait_partial finalitzarà.

Cal destacar un important avantatge d'aquest mètode de depuració: quan executem pas a pas, o detenim la simulació, el temps del món simulat és detingut. Això és impossible

si es depura sobre el dispositiu real. Per exemple, si estem depurant la comunicació amb un perifèric extern, quan executem pas a pas el perifèric segueix funcionant, i la depuració pot ser gairebé impossible. En canvi, amb l'emulador, el perifèric també "s'executa" pas a pas. Podem veure el seu estat en cada instant de temps, a mesura que executem el nostre programa pas a pas.

8.3 Target en gdbproxy

El programa gdbproxy està dissenyat de manera que és molt fàcil l'extensió amb nous objectius (*targets*). Un target és un dispositiu per al qual es vol implementar la comunicació. Amb el microcontrolador real es fa servir el target msp430. Per a aquest projecte, s'ha implementat el target mspemu.

En el codi font de gdbproxy s'inclou un fitxer de codi anomenat target_skeleton.c. Aquí trobem el codi d'un target genèric, que no connecta amb cap dispositiu, però en el codi s'indica les parts que cal completar per a implementar la comunicació.

A partir de target_skeleton.c s'ha creat target_mspemu.c. El codi afegit es redueix majorment a la propagació de les comandes que arriben de GDB cap a les funcions descrites en la secció anterior. Ha sigut molt útil l'apèndix D del llibre "Debugging with GDB" [7].

Tot el codi de gdbproxy, amb les modificacions fetes, es troba en el directori src/gdbproxy. A més de crear el fitxer target_mspemu.c s'ha hagut de modificar init.c per afegir una referència al nou target, i el fitxer Makefile.am, per a enllaçar amb l'emulador.

Capítol 9

Exemple d'utilització

9.1 Introducció

En aquest capítol es descriuen tots els passos necessaris per a depurar un programa de MSP430 amb l'emulador.

9.2 Instal.lar l'emulador

Per a instal.lar l'emulador, cal seguir els passos següents:

1. Descomprimir el codi font, per exemple:
   $ cd /usr/local/src
   $ tar xzvf ~/msp430-emu.tgz

2. Compilar l'emulador:
   $ cd msp430-emu/src
   $ make

3. Opcionalment, executar les proves:
   $ cd test
   $ make check

4. Compilar el programa gdbproxy modificat (requereix automake-1.6):
   $ cd ../gdbproxy

$ ./configure
(NOTA: si falla, executar ./configure –build=i386)
$ make

5. Instal.lar gdbproxy, que incorpora l'emulador enllaçat estàticament. S'instal.larà en /usr/local/bin, cal tenir accés o convertir-se en root:
$ make install

9.3 Instal.lar eines de MSPGCC

Necessitem les eines de MSPGCC per a poder compilar i depurar programes per al MSP430. Gran part de les eines de MSPGCC són pegats d'eines del projecte GNU, que cal descarregar:

1. Binutils: ftp://sources.redhat.com/pub/binutils/releases/binutils-2.14.tar.bz2
2. Core de GCC: fitxer /releases/gcc-3.2.3/gcc-core-3.2.3.tar.bz2 en algun dels miralls (*mirrors*) de http://gcc.gnu.org/mirrors.html.
3. Insight (GDB amb interfície gràfica): ftp://sources.redhat.com/pub/gdb/old-releases/insight-5.1.1.tar.bz2.

Les eines i pegats de MSPGCC són accessibles mitjançant CVS anònim:

1. Autentificar-se en el CVS:
$ export CVSROOT=:pserver:anonymous@cvs.mspgcc.sourceforge.net:/cvsroot/mspgcc
$ export CVS_RSH=ssh
$ cvs login
(polsar intro quan es demani contrasenya)
2. Obtenir les eines i pegats:
$ mkdir /usr/local/src/mspgcc
$ cd /usr/local/src/mspgcc
$ cvs checkout gcc
$ cvs checkout gdb
$ cvs checkout msp430-libc
$ cvs checkout packaging

3. Instal.lar binutils:
   $ cd /usr/local/src
   $ tar xjf ~/binutils-2.14.tar.bz2
   $ cd binutils-2.14
   $ ./configure –target=msp430 –prefix=/usr/local
   $ make
   $ make install

4. Instal.lar GCC aplicant pegats de MSPGCC:
   $ cd /usr/local/src
   $ tar xjf ~/gcc-core-3.2.3.tar.bz2
   $ cp -a mspgcc/gcc/gcc-3.3/* gcc-3.2.3
   $ cd gcc-3.2.3
   $ ./configure –target=msp430 –prefix=/usr/local
   $ make
   $ make install

5. Instal.lar la libc de MSPGCC:
   $ cd /usr/local/src/mspgcc/msp430-libc/src
   (editar Makefile, canviar /usr/local/msp430 per /usr/local)
   $ make
   $ make install

6. Instal.lar GDB amb la interfície gràfica Insight, aplicant pegats de MSPGCC:
   $ cd /usr/local/src
   $ tar xjf insight-5.1.1.tar.bz2
   $ cp -a mspgcc/gdb/gdb-5.1.1/* insight-5.1.1
   $ cd insight-5.1.1
   $ ./configure –target=msp430 –prefix=/usr/local
   $ make
   $ make install

7. Copiar la configuració de Insight entregada amb el codi del projecte (millora notablement la qualitat de les fonts utilitzades):
   $ cp /usr/local/src/msp430-emu/src/iface/gdbtkinit ~/.gdbtkinit

9.4 Compilar un programa de prova

Compilem el programa d'exemple entregat amb el codi del projecte:

```
$ cd /tmp
$ cp /usr/local/src/msp430-emu/src/test/natiu/exemple.c .
$ msp430-gcc -mmcu=msp430x149 -g -o exemple exemple.c
```

9.5 Executar el programa en l'emulador

Els passos a seguir són els següents:

1. En una terminal, arrancar gdbproxy, que està enllaçat amb l'emulador:
   $ gdbproxy mspemu
2. En una altra terminal, arrancar msp430-gdb. Això obrirà la interfície gràfica Insight.
3. En el menú File seleccionar Open... i carregar el programa compilat (/tmp/exemple).
4. Connectar amb l'emulador:
   (a) En el menú Run seleccionar "Connect to target".
   (b) En "Target:" seleccionar "Remote/TCP".
   (c) En "Hostname" introduir "127.0.0.1", i en "Port:" introduir "2000".
   (d) Polsar OK.
5. Hauria d'aparèixer el missatge "Successfully connected". Polsar OK.
6. Carregar el programa en l'emulador:
   (a) Obrir la consola de GDB (en el menú View seleccionar Console).
   (b) En la consola introduir les següents comandes:
   (gdb) monitor erase all
   (gdb) load
7. Ja podem executar el programa, detenir el temps, executar pas a pas, etc. En la sortida de gdbproxy veurem com es va encenent i apagant el LED.

Capítol 10

Conclusions

10.1 Consecució dels objectius

Durant la realització del projecte, i d'acord amb el director del mateix, s'ha donat més prioritat a la qualitat del producte, que no al compliment de la planificació. Això es tradueix en la falta d'alguns mòduls que s'havia previst implementar (temporitzadors, ports USART, comparador i ADC). No obstant, la major part dels mòduls implementats estan ben provats i implementen tota la funcionalitat dels mòduls físics equivalents, amb un grau de realisme tan alt com ha sigut possible.

S'ha pogut observar que el disseny i programació de proves automatitzades requereix al menys tant de temps com la implementació del mòdul provat. Com que en un principi no s'havia previst crear proves automatitzades, aquest temps no s'havia tingut en compte. De totes maneres, a la llarga és clar que aquestes proves, a part d'assegurar la qualitat del producte, redueixen el temps dedicat a manteniment correctiu, ja que la major part de les errades es troben durant el desenvolupament, i no de manera aleatòria al llarg del temps.

La decisió de crear un simulador de circuits digitals, i després implementar l'emulador sobre ell, ha tingut diverses conseqüències:

- S'ha aconseguit un realisme difícil d'assolir d'una altra manera, ja que el simulador de circuits obliga a resoldre els mateixos problemes que han tingut els dissenyadors del microcontrolador.

- El codi és fàcil d'entendre, en molts casos prové d'esquemes de la guia d'usuari.
- En alguns casos s'ha allargat el temps de desenvolupament, especialment amb la CPU, on s'han hagut de deduir i simular les operacions realitzades en cada cicle de rellotge.

Per a finalitzar, cal destacar la importància de l'orientació a objectes en aplicacions de simulació. En aquest projecte s'han explotat amb gran satisfacció les nocions d'herència, polimorfisme, sobrecàrrega, etc., aconseguint un disseny i un codi molt més nets del que seria possible amb programació estructurada.

10.2 Tasques futures

El producte resultant d'aquest projecte es posarà a disposició de la comunitat de programari lliure, sota la llicència GPL, per tal de garantir la seva continuïtat. S'espera la col.laboració del projecte MSPGCC, i de qualsevol persona o entitat interessada en col.laborar.

Les tasques proposades són:

1. Optimitzar el simulador de circuits digitals, redissenyant-lo si cal. És necessari per a aplicacions que passen molt de temps en mode actiu.

2. Augmentar el realisme del mòdul Flash. Actualment els esborrats i les escriptures són immediates, i en la realitat no és així. A més, crear un test automatitzat d'aquest mòdul.

3. Emular els mòduls que falten, descrits en la secció 6.9. Implementar la capacitat d'interrupció i les funcions alternatives dels ports.

4. Dissenyar més tests automatitzats, per a garantir la qualitat. Si és possible, dissenyar-los amb ajuda d'un oscil.loscopi, per a augmentar el realisme.

5. Com que tots els models de la família MSP430 són relativament semblants, convindria parametritzar l'emulador per a que es pugui triar (en temps d'execució o compilació), el model emulat.

6. Modificar la interfície amb GDB per a permetre la connexió de diverses instàncies del microcontrolador en el mateix circuit, cadascun controlat per una instància de GDB.

7. Crear una interfície gràfica que permeti visualitzar els mòduls, les línies, els valors de les línies, informació específica de cada mòdul (per exemple els registres interns), i que permeti "entrar" en els mòduls Composite. Això permetria disposar d'una plataforma de depuració molt potent.

Glossari

ADC *Analog to Digital Converter*. Mòdul electrònic capaç de discretitzar un senyal analògic, convertint-lo en un conjunt de bits que descriuen el seu valor.

ALU *Arithmetic and Logic Unit*. Unitat lògico-aritmètica. Part de la CPU on es processen les operacions lògiques i aritmètiques.

Callback Rutina cridada per un mòdul extern quan té lloc un cert esdeveniment.

Clock Senyal quadrat utilitzat per a la sincronització d'elements electrònics. Normalment generat per un oscilador.

CPU *Central Processing Unit*. Mòdul electrònic que executa les instruccions d'un programa, per tal de controlar altres mòduls.

DCO *Digitally Controlled Oscillator*. Oscilador controlat digitalment. Oscilador de freqüència configurable programàticament.

Flash *Flash Erasable Programmable Read-Only Memory*. Un tipus de dispositiu d'emmagatzament no volàtil, similar a EEPROM (*Electrically Erasable Programmable Read-Only Memory*), però on l'esborrat només es pot fer en blocs o en tot el dispositiu a la vegada.

Interrupció Esdeveniment asíncron que modifica temporalment el flux d'execució, per tal de que s'executi una rutina de servei a la interrupció.

JTAG *Join Test Action Group*. Estàndard que especifica com controlar i monitoritzar els pins dels dispositius que el segueixen, sobre una placa de circuit imprès. Amb alguns microcontroladors es fa servir aquest estàndard per a carregar i depurar programes.

Latch Mòdul electrònic que emmagatzema el valor d'un o més bits. Si és síncron, només s'actualitzen amb els *ticks* d'un rellotge. S'utilitza en ports d'entrada síncrons (també anomenats *latched ports*).

LED *Light Emitting Diode*. Diode que emet llum quan passa corrent a través d'ell.

MAB *Main Address Bus*. Bus on la CPU posa l'adreça del mapa de memòria d'on necessita llegir o escriure.

MDB *Main Data Bus*. Bus utilitzat per a l'intercanvi de dades entre la CPU i els perifèrics (o entre perifèrics en alguns sistemes).

Microcontrolador Circuit integrat que inclou una CPU, memòria RAM, PROM, circuitreria d'entrada/sortida, i altres perifèrics integrats.

Multiplexor Component electrònic on una entrada (el selector, de n bits) selecciona quina de les 2^n altres entrades s'ha de propagar a la sortida.

NMI *Non Maskable Interrupt*. Interrupció que no es deshabilita quan es deshabiliten les interrupcions globalment. Corresponen a errors greus en el sistema.

Offset Desplaçament aplicat a una adreça.

Opcode Codi d'operació. Part de la codificació d'una instrucció que determina l'operació a realitzar.

Oscilador Dispositiu electrònic que genera un senyal quadrat de període estable.

PC *Program Counter*. Registre de la CPU on s'emmagatzema l'adreça de la instrucció a executar.

Pila Part de la memòria que normalment s'accedeix en ordre LIFO (*Last-In First-Out*).

POR *Power On Reset*. Senyal de *reset* que s'activa a causa d'un esdeveniment extern, com l'activació del voltatge d'entrada, o la pulsació d'un botó de reset.

PUC *Power Up Clear*. Senyal de *reset* secundaria, activada per *POR* o per un error intern del sistema.

RAM *Random Access Memory*. Dispositiu d'emmagatzament volàtil de lectura i escriptura on el fet d'accedir a adreces no consecutives no repercuteix en la velocitat d'accés.

SFR *Special Function Register*. Registre de funció especial. No pot ser utilitzat per a emmagatzemar dades, normalment conté bits de configuració.

SP *Stack Pointer*. Registre de la CPU que conté l'adreça de la capçalera de la pila.

SR *Status Register*. Registre de la CPU on s'emmagatzemen resultats secundaris de les operacions realitzades, i altres *flags* com l'habilitació d'interrupcions o els modes de baix consum d'energia.

USART *Universal Synchronous/Asynchronous Receive/Transmit*. Mòdul electrònic utilitzat per comunicacions serie, que conté un transmissor (conversor paral.lel a sèrie), i un receptor (conversor sèrie a paral.lel).

Watchdog Temporitzador que provoca un *reset* del sistema quan expira. El programa executat ha d'actualitzar el *watchdog* periòdicament. Si no ho fa, s'assumeix que ha quedat en un estat erroni.

Bibliografia

[1] Projecte MSPGCC. http://mspgcc.sourceforge.net/

[2] Projecte GNU. http://www.gnu.org/

[3] CPPUNIT. http://sourceforge.net/projects/cppunit/

[4] MSP430x1xx Family User's Guide (SLAU049D).
http://www.ti.com/msp430

[5] MSP430x13x, MSP430x14x, MSP430x14x1
Mixed Signal Microcontroller (datasheet, SLAS272E).
http://www.ti.com/msp430

[6] GDB. http://sources.redhat.com/gdb/

[7] Debugging with GDB.
http://sources.redhat.com/gdb/current/onlinedocs/gdb.html

[8] Fonaments de Computadors II, Universitat Oberta de Catalunya.
ISBN: 84-8429-259-2.

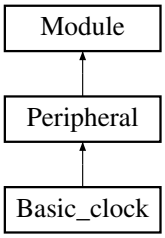
[9] Doxygen. http://www.doxygen.org

Apèndix A
Documentació del codi

A.1 Referència de la Classe Basic_clock

`#include <basic_clock.hpp>`

Diagrama d'Herència per a Basic_clock::



Mètodes públics

- **Basic_clock** (module_t _id, **Route** ∗_route, bool _printing=false)
- void **change_event** (port_t nport)
- void **time_event** (u64 data)

A.1.1 Descripció detallada

Basic Clock. Generador programable de les línies de rellotge del MSP430. MCLK: Main Clock. Utilitzat per la CPU. ACLK: Auxiliary Clock. SMCLK: Sub-Main Clock.

A.1.2 Documentació de les Funcions membre

A.1.2.1 void Basic_clock::change_event (port_t *nport*) `[virtual]`

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat de **Peripheral** (p. 92).

A.1.2.2 void Basic_clock::time_event (u64 *data*) `[virtual]`

Cridat per **Real_time**(p. 98) per a un esdeveniment programat.

Reimplementat de **Module** (p. 86).

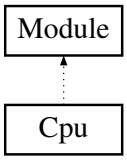
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- basic_clock.hpp
- basic_clock.cpp

A.2 Referència de la Classe Cpu

`#include <cpu.hpp>`

Diagrama d'Herència per a Cpu::



Mètodes públics

- **Cpu** (module_t _id, **Route** ∗_route, bool _printing=false, void(∗_cb_report)(u16)=0)
- ∼**Cpu** ()
- void **change_event** (port_t nport)
- u16 **test_reg** (int reg)
- void **test_reg_write** (int reg, u16 val)
- cpu_state_t **test_state** ()

A.2.1 Descripció detallada

CPU del microcontrolador MSP430.

A.2.2 Documentació del Constructor i el Destructor

A.2.2.1 Cpu::Cpu (module_t *_id*, Route ∗ *_route*, bool *_printing* = false, void(∗ *_cb_report*)(u16) = 0)

Constructor.

Paràmetres:

_id Identificador d'aquest mòdul.

_route Enrutador on està aquest mòdul.

_printing Imprimir missatges de depuració.

_cb_report Funció on reportar fetch (a gdbproxy).

A.2.2.2 Cpu::∼Cpu ()

Destructor.

A.2.3 Documentació de les Funcions membre

A.2.3.1 void Cpu::change_event (port_t *nport*) [virtual]

Mètode de **Module**(p. 83) redefinit. Cridat quan hi ha un canvi en una entrada. Es fa servir per a detectar els ticks del rellotge, i les peticions d'interrupció i reset.

Reimplementat de **Module** (p. 84).

A.2.3.2 u16 Cpu::test_reg (int *reg*)

Retorna el valor d'un registre. Només per a depurar.

A.2.3.3 void Cpu::test_reg_write (int *reg*, u16 *val*)

Canvia el valor d'un registre. Només per a depurar. Com que es supossa que només es depura quan s'està entrant en FETCH_INSTR, si es modifica el PC, el nou valor es propaga al MAB.

A.2.3.4 cpu_state_t Cpu::test_state ()

Retorna l'estat de la CPU. Només per a depurar.

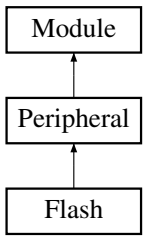
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- cpu.hpp
- cpu.cpp

A.3 Referència de la Classe Flash

`#include <flash.hpp>`

Diagrama d'Herència per a Flash::



Mètodes públics

- **Flash** (module_t _id, **Route** ∗_route, bool _printing=false)
- void **change_event** (port_t nport)
- u16 **test_read_flash** (u16 addr)
- void **test_write_flash** (u16 addr, u8 byte)

A.3.1 Descripció detallada

Memòria Flash del MSP430f139.

A.3.2 Documentació de les Funcions membre

A.3.2.1 void Flash::change_event (port_t *nport*) `[virtual]`

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat de **Peripheral** (p. 92).

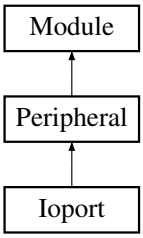
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- flash.hpp
- flash.cpp

A.4 Referència de la Classe Ioport

`#include <ioport.hpp>`

Diagrama d'Herència per a Ioport::



Mètodes públics

- **Ioport** (module_t _id, **Route** ∗_route, u16 addr, bool _interrupt=false, bool _printing=false)
- void **change_event** (port_t nport)

A.4.1 Descripció detallada

Port d'entrada sortida de 8 línies amb configuració independent, i opcionalment capacitat d'interrupció. A partir d'aquesta classe s'instancien els 6 ports de la unitat.

A.4.2 Documentació del Constructor i el Destructor

A.4.2.1 Ioport::Ioport (module_t *_id*, Route ∗ *_route*, u16 *addr*, bool *_interrupt* = false, bool *_printing* = false)

Constructor.

Paràmetres:

_id Identificador d'aquest mòdul.

_route Enrutador on està aquest mòdul.

addr Adreça base dels registres de configuració.

_interrupt Indica si té capacitat d'interrompre.

_printing Imprimir missatges de depuració.

A.4.3 Documentació de les Funcions membre

A.4.3.1 void Ioport::change_event (port_t *nport*) [virtual]

Mètode de **Module**(p. 83) redefinit. Cridat quan hi ha un canvi en una entrada.

Reimplementat de **Peripheral** (p. 92).

La documentació d'aquesta classe es va generar a partir dels següents arxius:

- ioport.hpp
- ioport.cpp

A.5 Referència de la Classe Line_mod_set

`#include <line_mod_set.hpp>`

Mètodes públics

- **Line_mod_set** ()
- ∼**Line_mod_set** ()
- void **add** (line_mod_s line_mod)
- void **begin_enum** ()
- bool **more_elements** ()
- void **next_element** (line_mod_s ∗p_line_mod)
- bool **has_module_port** (module_t id, port_t nport)

A.5.1 Descripció detallada

Conjunt dels mòduls connectats a una línea, incloent el número de port amb el que hi estan connectats.

A.5.2 Documentació del Constructor i el Destructor

A.5.2.1 Line_mod_set::Line_mod_set () `[inline]`

Constructor.

A.5.2.2 Line_mod_set::∼Line_mod_set ()

Destructor.

A.5.3 Documentació de les Funcions membre

A.5.3.1 void Line_mod_set::add (line_mod_s *line_mod*)

Afegir un port d'un mòdul a la línia.

A.5.3.2 void Line_mod_set::begin_enum ()

Començar l'enumeració dels mòduls connectats.

A.5.3.3 bool Line_mod_set::has_module_port (module_t *id*, port_t *nport*)

Retorna true ssi el mòdul 'id' té el port 'nport' connectat a la línia.

A.5.3.4 bool Line_mod_set::more_elements ()

Torna true si n'hi ha més elements, false si no.

A.5.3.5 void Line_mod_set::next_element (line_mod_s * *p_line_mod*)

Possa el següent element en *p_line_mod.

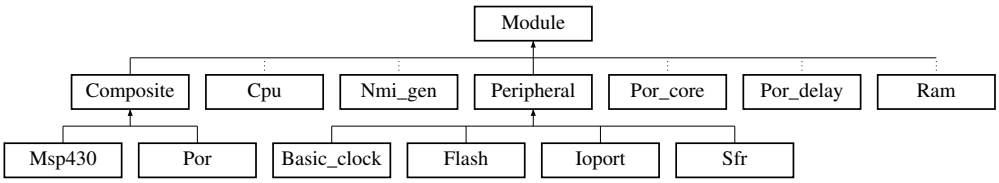
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- line_mod_set.hpp
- line_mod_set.cpp

A.6 Referència de la Classe Module

`#include <module.hpp>`

Diagrama d'Herència per a Module::



Mètodes públics

- **Module** (module_t _id, **Route** ∗_route, port_t _num_ports)
- **Module** (module_t _id, **Route** ∗_route, port_t _num_ports, bool compo)
- virtual ∼**Module** ()
- virtual void **time_event** (u64 data)
- void **time_event** (output_s data)
- virtual void **change_event** (port_t nport)

Mètodes Protegits

- void **output** (data_t value, port_t nport, delay_t delay)
- void **output** (data_t value, port_t nport)
- data_t **input** (port_t nport)
- void **latches_clock** ()
- void **set_port** (port_t nport, direction_t dir, bool latched=false, width_t bits=1)
- void **config_ports** ()

Atributs Protegits

- port_s ∗ **port**
- port_t **num_ports**
- delay_t **default_delay**

A.6.1 Descripció detallada

Classe base de tots els emuladors de mòduls digitals.

A.6.2 Documentació del Constructor i el Destructor

A.6.2.1 Module::Module (module_t *_id*, Route * *_route*, port_t *_num_ports*)

Constructor.

Paràmetres:

_id Identificador d'aquest mòdul (dins _route).

_route El **Route**(p. 101) on està aquest mòdul.

_num_ports Número de ports d'aquest mòdul.

A.6.2.2 Module::Module (module_t *_id*, Route * *_route*, port_t *_num_ports*, bool *compo*)

Constructor amb un paràmetre adicional que indica si és un **Composite**(p. **??**).

A.6.2.3 Module::~Module () [virtual]

Destructor.

A.6.3 Documentació de les Funcions membre

A.6.3.1 void Module::change_event (port_t *nport*) [virtual]

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat a **Composite** (p. **??**), **Basic_clock** (p. 74), **Cpu** (p. 77), **Flash** (p. 78), **Ioport** (p. 80), **Nmi_gen** (p. 90), **Peripheral** (p. 92), **Por_core** (p. 95) i **Por_delay** (p. 96).

A.6.3.2 void Module::config_ports () [protected]

Configurar els ports. S'ha de cridar des del constructor de la subclasse, després de cridar **set_port()**(p. 85) per a cada port del mòdul.

A.6.3.3 data_t Module::input (port_t *nport*) [protected]

Si el port té latch, obtenir el valor del latch. Sinò, obtenir l'estat de la línia a la que està connectat.

A.6.3.4 void Module::latches_clock () [protected]

Enviar un senyal de clock als latches de totes les entrades amb latch.

A.6.3.5 void Module::output (data_t *value*, port_t *nport*) [protected]

Canviar l'estat d'un port, amb el delay per defecte.

A.6.3.6 void Module::output (data_t *value*, port_t *nport*, delay_t *delay*) [protected]

Canviar l'estat d'un port, en 'delay' nanosegons.

A.6.3.7 void Module::set_port (port_t *nport*, direction_t *dir*, bool *latched* = false, width_t *bits* = 1) [protected]

Configurar un port. Cridar en el constructor per a cada port del mòdul.

A.6.3.8 void Module::time_event (output_s *data*)

Cridat per **Real_time**(p. 98) tras un delay en una sortida.

A.6.3.9 void Module::time_event (u64 *data*) [virtual]

Cridat per **Real_time**(p. 98) per a un esdeveniment programat.

Reimplementat a **Basic_clock** (p. 74) i **Por_delay** (p. 96).

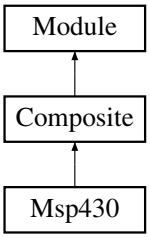
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- module.hpp
- module.cpp

A.7 Referència de la Classe Msp430

`#include <msp430.hpp>`

Diagrama d'Herència per a Msp430::



Mètodes públics

- **Msp430** (module_t _id, **Route** ∗_route, void(∗cpu_report)(u16)=0)
- u16 **test_reg** (int reg)
- void **test_reg_write** (int reg, u16 val)
- u16 **test_mem** (u16 addr)
- void **test_mem_write** (u16 addr, u8 byte)

A.7.1 Descripció detallada

Microcontrolador MSP430f149.

A.7.2 Documentació de les Funcions membre

A.7.2.1 u16 Msp430::test_mem (u16 *addr*)

Retorna el valor del word especificat en el mapa de memòria. Només per a depurar.

A.7.2.2 void Msp430::test_mem_write (u16 *addr*, u8 *byte*)

Escriure un byte. Només per a depurar.

A.7.2.3 u16 Msp430::test_reg (int *reg*) [inline]

Retorna el valor d'un registre. Només per a depurar.

A.7.2.4 void Msp430::test_reg_write (int *reg*, u16 *val*) [inline]

Escriure un registre. Només per a depurar.

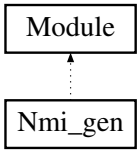
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- msp430.hpp
- msp430.cpp

A.8 Referència de la Classe Nmi_gen

`#include <nmi_gen.hpp>`

Diagrama d'Herència per a Nmi_gen::



Mètodes públics

- **Nmi_gen** (module_t _id, **Route** ∗_route, **Sfr** ∗_sfr)
- void **change_event** (port_t nport)

A.8.1 Descripció detallada

NMI Generator. Mòdul auxiliar per a generar interrupcions no enmascarables. Veure punt 5.4.3 de la memòria.

A.8.2 Documentació del Constructor i el Destructor

A.8.2.1 Nmi_gen::Nmi_gen (module_t *_id*, Route ∗ *_route*, Sfr ∗ *_sfr*)

Constructor.

Paràmetres:

_id Identificador d'aquest mòdul.

_route Enrutador on està aquest mòdul.

_sfr El mòdul SFR.

A.8.3 Documentació de les Funcions membre

A.8.3.1 void Nmi_gen::change_event (port_t *nport*) [virtual]

Mètode de **Module**(p. 83) redefinit. Cridat quan hi ha un canvi en una entrada.

Reimplementat de **Module** (p. 84).

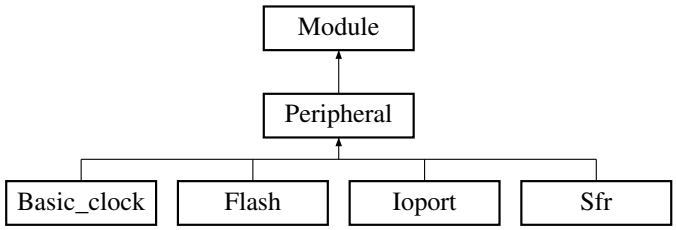
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- nmi_gen.hpp
- nmi_gen.cpp

A.9 Referència de la Classe Peripheral

`#include <peripheral.hpp>`

Diagrama d'Herència per a Peripheral::



Mètodes públics

- **Peripheral** (module_t _id, **Route** ∗_route, port_t _num_ports, bool _byte_periph, const bit_cfg_s ∗_bit_cfg, u16 base, int nbytes, bool _printing=false)
- ∼**Peripheral** ()
- void **change_event** (port_t nport)
- u16 **test_read** (u16 addr)

Mètodes Protegits

- void **mod_bit** (u16 addr, int bit, bool value)

Atributs Protegits

- const int **mem_size**
- u8 ∗ **mem**

A.9.1 Descripció detallada

Classe base de tots els perifèrics integrats del MSP430, amb registres mapejats en el mapa de memòria. En aquesta classe s'encapsula la funcionalitat relativa al comportament d'aquests registres.

A.9.2 Documentació del Constructor i el Destructor

A.9.2.1 Peripheral::Peripheral (module_t *_id*, Route * *_route*, port_t *_num_ports*, bool *_byte_periph*, const bit_cfg_s * *_bit_cfg*, u16 *base*, int *nbytes*, bool *_printing* = false)

Constructor.

Paràmetres:

_id Identificador d'aquest mòdul (dins _route).

_route El **Route**(p. 101) on està aquest mòdul.

_num_ports Número de ports d'aquest mòdul.

_byte_periph true si és un perifèric de 8 bits, false si és de 16 bits.

_bit_cfg Punter a l'array que descriu el comportament de cada bit.

base Adreça base d'aquest perifèric en el mapa de memòria.

nbytes Tamany d'aquest perifèric en el mapa de memòria, en bytes.

A.9.2.2 Peripheral::~Peripheral ()

Destructor.TODO Virtual pur per fer la classe abstracta.

A.9.3 Documentació de les Funcions membre

A.9.3.1 void Peripheral::change_event (port_t *nport*) [virtual]

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat de **Module** (p. 84).

Reimplementat a **Basic_clock** (p. 74), **Flash** (p. 78) i **Ioport** (p. 80).

A.9.3.2 void Peripheral::mod_bit (u16 *addr*, int *bit*, bool *value*) [protected]

Modificar el valor del bit indicat de l'adreça 'addr'

Reimplementat a **Sfr** (p. 104).

A.9.3.3 u16 Peripheral::test_read (u16 *addr*)

Retorna el valor del word amb adreça especificada, si l'adreça pertany a aquest perifèric. Sinó, torna 0.

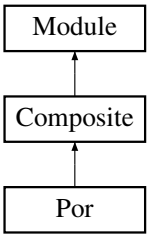
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- peripheral.hpp
- peripheral.cpp

A.10 Referència de la Classe Por

`#include <por.hpp>`

Diagrama d'Herència per a Por::



Mètodes públics

- **Por** (module_t _id, **Route** ∗_route)

A.10.1 Descripció detallada

Generador de les línies de reset: POR: Power On Reset PUC: Power Up Clear Veure capítol 2 del slau049d.

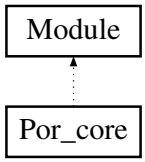
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- por.hpp
- por.cpp

A.11 Referència de la Classe Por_core

`#include <por_core.hpp>`

Diagrama d'Herència per a Por_core::



Mètodes públics

- **Por_core** (module_t _id, **Route** ∗_route)
- void **change_event** (port_t nport)

A.11.1 Descripció detallada

Submòdul central del mòdul POR.

A.11.2 Documentació de les Funcions membre

A.11.2.1 void Por_core::change_event (port_t *nport*) `[virtual]`

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat de **Module** (p. 84).

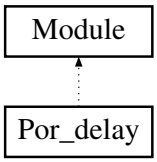
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- por_core.hpp
- por_core.cpp

A.12 Referència de la Classe Por_delay

`#include <por_delay.hpp>`

Diagrama d'Herència per a Por_delay::



Mètodes públics

- **Por_delay** (module_t _id, **Route** ∗_route)
- void **change_event** (port_t nport)
- void **time_event** (u64 data)

A.12.1 Descripció detallada

Submòdul "POR Delay" del mòdul POR. Veure manual d'usuari, pàgina 2-2.

A.12.2 Documentació de les Funcions membre

A.12.2.1 void Por_delay::change_event (port_t *nport*) `[virtual]`

Cridat per **Route**(p. 101) quan hi ha un canvi en una entrada.

Reimplementat de **Module** (p. 84).

A.12.2.2 void Por_delay::time_event (u64 *data*) `[virtual]`

Cridat per **Real_time**(p. 98) per a un esdeveniment programat.

Reimplementat de **Module** (p. 86).

La documentació d'aquesta classe es va generar a partir dels següents arxius:

- por_delay.hpp
- por_delay.cpp

A.13 Referència de la Classe Real_time

`#include <real_time.hpp>`

Mètodes públics

- **Real_time** ()
- ∼**Real_time** ()
- rtime_t **get_time** ()
- bool **is_running** ()
- void **request_event** (**Module** ∗module, rtime_t delay, u64 data)
- void **request_event** (**Module** ∗module, rtime_t delay, output_s data)
- void **run_until** (rtime_t when)
- void **run_step** ()
- void **run** ()
- void **stop** ()
- void **reset** ()

Atributs Públics

- float **accel**

A.13.1 Descripció detallada

Temps real. Controla el temps en el món simulat. Els mòduls l'utilitzen per comprovar el temps i programar esdeveniments futurs. Quan ocorre un esdeveniment de temps, el mètode time_event() del mòdul és cridat. La unitat de temps és el nanosegon, i es representa en 64 bits, per tant es poden representar més de 584 anys.

A.13.2 Documentació del Constructor i el Destructor

A.13.2.1 Real_time::Real_time () `[inline]`

Constructor.

A.13.2.2 Real_time::∼Real_time ()

Destructor.

A.13.3 Documentació de les Funcions membre

A.13.3.1 rtime_t Real_time::get_time () [inline]

Obtenir el nanosegon actual.

A.13.3.2 bool Real_time::is_running () [inline]

Veure si s'està executant la simulació.

A.13.3.3 void Real_time::request_event (Module ∗ *module*, rtime_t *delay*, output_s *data*)

Programar una sortida amb delay. Quan transcorrin 'delay' nanosegons, provocarem el canvi ('data') en les sortides de 'module'.

A.13.3.4 void Real_time::request_event (Module ∗ *module*, rtime_t *delay*, u64 *data*)

Programar un esdeveniment. Quan transcorrin 'delay' nanosegons, passarem 'data' al mètode 'time_event' de 'module'.

A.13.3.5 void Real_time::reset ()

Resetejar el temps. Cridar abans de canviar el circuit.

A.13.3.6 void Real_time::run ()

Executar el circuit fins que s'estabilitzi o l'usuari polsi ctrl+c.

A.13.3.7 void Real_time::run_step ()

Executar un esdeveniment.

A.13.3.8 void Real_time::run_until (rtime_t *when*)

Executar el circuit fins al nanosegon 'when'.

A.13.3.9 void Real_time::stop ()

Parar el temps.

La documentació d'aquesta classe es va generar a partir dels següents arxius:

- real_time.hpp
- real_time.cpp

A.14 Referència de la Classe Route

`#include <route.hpp>`

Mètodes públics

- **Route** (line_t _nlines, module_t _nmods)
- ∼**Route** ()
- void **register_module** (**Module** ∗module, module_t id)
- void **set_line** (module_t id, port_t nport, line_t nline)
- line_t **get_line** (module_t id, port_t nport)
- void **set** (data_t value, line_t nline)
- data_t **get** (line_t nline)
- void **set_watch** (line_t nline)
- void **clear_watch** (line_t nline)
- void **set_line_width** (line_t nline, width_t width)
- width_t **get_line_width** (line_t nline)
- void **set_line_name** (line_t nline, const char ∗name)

A.14.1 Descripció detallada

S'encarrega de la connexió dels mòduls entre si. Conté un conjunt de línies a les que els mòduls es connecten. En cada línia normalment hi ha un mòdul connectat amb un port configurat com sortida, i un o més mòduls connectats amb un port configurat com entrada.

A.14.2 Documentació del Constructor i el Destructor

A.14.2.1 Route::Route (line_t *_nlines*, module_t *_nmods*)

Constructor. Per defecte totes les línies són d'un bit. Es pot canviar amb 'set_line_-width'.

Paràmetres:

_nlines Número de línies.

_nmods Número de mòduls.

A.14.2.2 Route::∼Route ()

Destructor.

A.14.3 Documentació de les Funcions membre

A.14.3.1 void Route::clear_watch (line_t *nline*) [inline]

Deixar de vigilar canvis.

A.14.3.2 data_t Route::get (line_t *nline*)

Obtenir l'estat d'una línia.

A.14.3.3 line_t Route::get_line (module_t *id*, port_t *nport*)

Obtenir en quina línia està connectat un port d'un mòdul.

A.14.3.4 width_t Route::get_line_width (line_t *nline*)

Obtenir el nombre de bits en una línia.

A.14.3.5 void Route::register_module (Module ∗ *module*, module_t *id*)

Associar un mòdul a aquest route. L'identificador ha de ser únic dins el route, i ha d'estar en el rang 0..(nmods-1).

A.14.3.6 void Route::set (data_t *value*, line_t *nline*)

Canviar l'estat d'una línia.

A.14.3.7 void Route::set_line (module_t *id*, port_t *nport*, line_t *nline*)

Establir a quina línia està connectat un port d'un mòdul.

A.14.3.8 void Route::set_line_name (line_t *nline*, const char * *name*)

Establir el nom d'una línia (per depurar).

A.14.3.9 void Route::set_line_width (line_t *nline*, width_t *width*)

Establir el nombre de bits en una línia.

A.14.3.10 void Route::set_watch (line_t *nline*) [inline]

Vigilar canvis en una línia (per depurar).

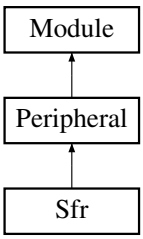
La documentació d'aquesta classe es va generar a partir dels següents arxius:

- route.hpp
- route.cpp

A.15 Referència de la Classe Sfr

`#include <sfr.hpp>`

Diagrama d'Herència per a Sfr::



Mètodes públics

- **Sfr** (module_t _id, **Route** ∗_route, bool printing=false)
- void **mod_bit** (u16 addr, int bit, bool value)

A.15.1 Descripció detallada

Banc de registres de funció especial.

A.15.2 Documentació de les Funcions membre

A.15.2.1 void Sfr::mod_bit (u16 *addr*, int *bit*, bool *value*)

Modificar el bit indicat. Aquest mòdul permet modificar directament als latches interns perquè alguns d'ells (com els flags d'interrupció) estan implementats en altres mòduls. Notar que no es permet la lectura (que s'ha de fer amb les línies connectades).

Reimplementat de **Peripheral** (p. 92).

La documentació d'aquesta classe es va generar a partir dels següents arxius:

- sfr.hpp
- sfr.cpp