Laboratorul 01 - Memorii

În acest laborator vom învăța despre memorii, vom implementa memorii ROM si RAM, si vom construi o aplicație simplă folosind o memorie.

Introducere

Exista situatii in care vrem sa folosim logica secventiala pentru a stoca mai multa informatie decat simpla stare curenta a unui automat finit, ci mai degraba informatii/date de dimensiuni mari si uz general. Astfel au aparut aceste blocuri de circuite secventiale de dimensiuni mari, numite memorii, si caracterizate de urmatoarele trasaturi:

- **dimensiune** (masurata in biti (b), octeti (B), si multiplii acestora)
- **timp de acces** (masurat in submultipli ai secundei, in general nanosecunde)
- **structura interna** (sau cum un bloc mare de memorie este construit din mai multe chip-uri mai mici)
- **volatilitatea datelor** (datele raman scrise in memorie dupa un ciclu power on-off-on?)
- **persistenta datelor** (vom vedea mai tarziu)
- **posibilitatea de modificare post-fabricatie a datelor**
- **modul de acces al datelor** (date care nu sunt in ordine in memorie se pot accesa consecutiv in timp?)
- **pret** ($)
- **tehnologia de fabricatie** (care, de fapt, joaca un rol important in determinarea tuturor celor de mai sus)
- **numar de porturi** (cate unitati hardware pot avea acces simultan la memorie, in acelasi ciclu de ceas)

Fiecare solutie este una de compromis[1)] iar in aplicatiile de dimensiuni mari se adopta o tehnica hibrida, numita **ierarhie de memorii**: o memorie foarte rapida (dar extrem de mica, pentru a mentine costul scazut) este "backed up" de una de dimensiuni ceva mai mari, dar performante ceva mai mici, care este "backed up" la randul ei de alta de dimensiuni si mai mari si performante si mai mici, etc.

Criterii de clasificare

Mod de acces

Unul din cele mai simple criterii de clasificare este cel dupa modul de acces:

- memorii cu **acces aleator** (RAM - random access memory)
- memorii cu **acces secvential** (spre exemplu benzile magnetice, care, pentru a accesa o anumita portiune a benzii necesita derularea acesteia).

In cadrul acestui laborator ne vor interesa doar memoriile de tip RAM.

Posibilitatea de modificare a datelor

Din acest punct de vedere exista:

- memorii care au **continut fixat** din momentul fabricatiei (ROM - read only memory)
- memorii care au un **continut pseudo-fixat** (dintre ele amintim EEPROM/Flash, o tehnologie pentru care scrierile degradeaza in timp celulele de memorie)

- memorii care au un **continut alterabil**.

Desi exista multe tipuri de memorie (categorie in care intra si device-urile de mass storage, precum HDD si SSD), in continuare ne vom concentra asupra memoriilor de tip RAM volatile (al caror continut se sterge dupa caderea tensiunii de alimentare).

Static vs. dinamic

Din alt punct de vedere, de data aceasta unul strict tehnologic, exista:

- **RAM static**: fiecare bit de memorie este implementat folosind bistabile sau latch-uri. Acest design are un consum extrem de eficient in idle, insa acesta creste odata cu frecventa acceselor. Costul per bit este ridicat si densitatea de biti per chip este scazuta. Pentru fiecare bit de stocare sunt necesari intre 4 si 6 tranzistori pentru a implementa efectiv bistabilul care memoreaza starea.
- **RAM dinamic**: un bit de memorie este implementat folosind un tranzistor si un condensator. Conform principiului de functionare al condensatoarelor, acestea pot retine o sarcina electrica pe termen nedefinit asa ca, cel putin in teorie, un bit de 0 poate fi stocat ca un condensator descarcat (tensiune 0 intre borne) respectiv un bit de 1 ca un condensator incarcat (tensiune mare intre borne). Din cauza imperfectiunilor[2)] condensatoarele din chip-urile de RAM dinamic se descarca singure dupa un timp (de ordinul milisecundelor). Din acest motiv celulele trebuie citite periodic iar acelea care erau scrise cu "1" trebuie rescrise. Acest procedeu se numeste **refresh** si reprezinta un overhead necesar doar pentru intretinere, RAM-ul fiind indisponibil in acest timp pentru accese utile ("inchis pentru curatenie"). Pe de alta parte densitatea de biti per chip este mult mai buna decat la SRAM si se pot obtine memorii de ordinul sutelor de MB (2^{27} - 2^{29}).

Memorii SRAM sincrone

In cadrul acestui laborator vom implementa o **memorie RAM statica sincrona**.

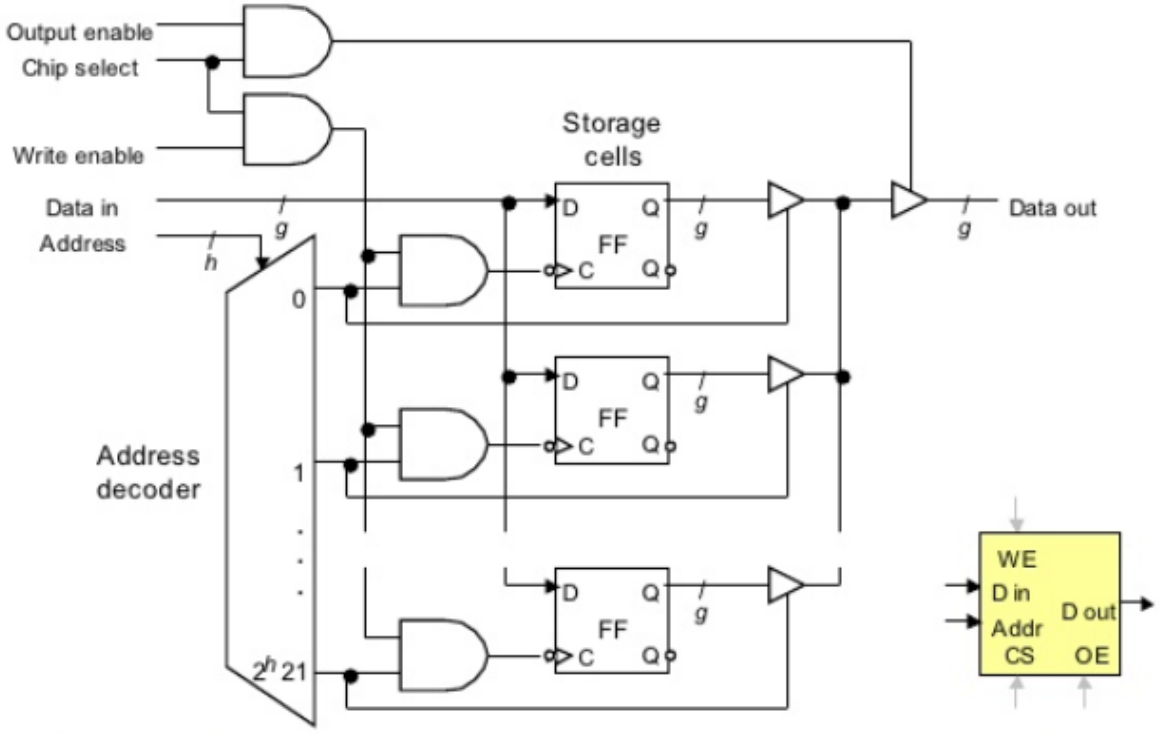
Descrierea semnalelor

Toate operatiile necesare pentru interfatarea unui chip SSRAM sunt controlate de unul sau mai multe semnale externe de ceas. Pentru a functiona corect toate semnalele de control trebuie sa fie valide atunci cand are loc tranzitia respectiva a ceasului, adica sa fie respectati timpii de setup si hold.

Functionarea unui SSRAM poate fi descrisa prin urmatoarele semnale:

- **Address (ADDR or SAx)**: acest input este folosit pentru a selecta o locatie de memorie de pe chip. In realitate, pentru memorii mari construite din mai multe chip-uri, atunci cand selectam o adresa de fapt selectam mai multe celule de memorie, cate una pentru fiecare chip. Spre exemplu, intr-o memorie formata din 18 chip-uri, selectarea unei adrese conduce la 18 locatii de memorie simultan. Mai departe, distinctia dintre acestea se face decodificand bitii suplimentari ai adresei. Dimensiunea adresei depinde de marimea memoriei. De exemplu, o memorie SRAM de 32K x 36 va avea 15 biti de adresa (2^{15} = 32K).
- **Data Inputs and Outputs (DQs or I/Os)**: Pinii de DQ sunt folositi pentru input si output de date. Pe unele memorii pinii de data input si data output sunt separati iar pe altele ei sunt multiplexati pe aceiasi pini. Spre exemplu, o memorie SRAM de 32K x 36 va avea 36 biti de date.
    - In timpul unei operatii de scriere un semnal de date se aplica pe pinii de data input. Aceste date sunt esantionate si stocate in celula de memorie selectata prin bitii de adresa.
    - In timpul unei operatii de citire datele de la adresa de memorie selectata vor aparea pe pinii de data output odata ce accesul s-a incheiat si output-ul a fost activat.

    - In majoritatea timpului pinii DQ sunt intr-o stare de impedanta marita, adica nu trag si nu dau curent. Ei nu prezinta niciun semnal catre exterior, ca si cand ar fi deconectati din circuit.
- **Output Enable (OE or G)**: (semnal descris in continuare ca activ high) atunci cand OE este 0 output-urile (DQ) sunt intotdeauna in stare de inalta impedanta. Cand OE este 1 output-urile sunt active iar datele pot aparea pe pini atunci cand sunt disponibile. OE este un semnal asincron: poate fi modificat in orice moment de timp iar SRAM-ul va raspunde imediat schimbarii.
    - In timpul unei operatii de citire acest semnal se foloseste pentru a bloca datele de a aparea la iesire pana atunci cand sunt necesare.
    - Inaintea unei operatii de scriere acest semnal este uneori folosit pentru a evita coliziuni pe magistrala de date prin tri-state.
    - In timpul operatiei de scriere propriu-zise semnalul OE este ignorat.
- **Clock**: La SRAM-urile cu un singur semnal de ceas acesta controleaza momentul cand semnalele de input sunt esantionate de catre memorie, la inceputul unui ciclu de citire sau scriere, si cand semnalele de output devin vizibile pe pini.
- **Chip Select (CS or SS)**: (activ high) este folosit pentru a permite/bloca semnalele de input catre chip. Atunci cand CS este 0 semnalele de input aplicate pe pini sunt ignorate.
- **Write Enable (WE or RW)**: (activ high - 1 = write) semnalul este folosit pentru a alege dintre o operatie de citire sau de scriere. Atunci cand WE este 1 datele aplicate pe pinii de intrare sunt copiate in memorie. Cand WE este 0 se incepe un ciclu de citire iar datele de pe linia de date sunt ignorate.



Stabilirea nivelelor logice pe care sunt active semnalele este o chestiune ce tine strict de conventie, si este de preferat pastrarea aceleiasi conventii pe intreg parcursul unui proiect. Pentru a asigura compatibilitatea intre module cu conventii diferite de polaritate, se pot utiliza porti logice inversoare.

TL;DR

Simplificat puternic, intrebati si asistentul :)

Memoriile pot fi

- rapide sau mari [3)]
- read sau read+write
- statice sau dinamice
- cu semnal de ceas sau fara (sincron/asincron)

Ca design hardware, vrem sa legam mai multe "chipuri" de memorie si sa si mearga. Semnalele WE, OE, si CS sunt cruciale pentru asta. Desi pare complicat, este cel mai simplu sistem care **merge**. Liniile de adrese si de date au rol evident.

Avem ierarhii de memorii ca sa reducem din limitari/costuri, chit ca introducem complexitate. Ierarhia de memorie se duce pana la hard-disk[4)] [5)].

Exerciții

Task 01 (1p) Implementați o memorie ROM de dimensiune 16×8 (16 octeți).

Pentru a adresa toți cei 16 octeți este nevoie de o lățime a adreselor de 4 biți.

Conținutul memoriei este: 65,78,65,32,65,82,69,32,50,32,77,69,82,69,46,0.

```
module rom (input[4] adresa, output[8] data)
{
    always
        case(adresa)
            4'd0:    data = 8'd1
            4'd1:    data = 8'd2
            4'd2:    data = 8'd4
            4'd3:    data = 8'd8
            4'd4:    data = 8'd16
            4'd5:    data = 8'd32
            4'd6:    data = 8'd64
            4'd7:    data = 8'd128
            4'd8:    data = 8'd170
            4'd9:    data = 8'd85
            4'd10:   data = 8'd153
            default: data = 8'd0
        end case
    end always
}
```

Semnalul `data` trebuie declarat ca `reg` din cauza restricției sintactice a limbajului Verilog, ca toate semnalele atribuite în interiorul unui bloc `always` să fie de tipul `reg`. Blocul always trebuie să fie asincron față de ceas deci trebuie să fie declarat `@*`

În realitate, memoria va fi sintetizată ca un mare multiplexor, cu intrările hardcodate pentru fiecare selecție posibilă. Memoria ROM cu conținut cu adevărat nemodificabil nu este nimic altceva decât un circuit combinațional care primește ca intrare o adresă și scoate la ieșire o valoare hardcodată pentru acea adresă.

Task 02 (1p) Implementați o memorie SRAM.

Atentie! Porturile de `date` vor fi de tip wire. Asta înseamna că nu vom mai putea face atribuiri pe aceste porturi în interiorul blocurilor procedurale. Va trebui să găsiți o altă metodă de a modifica aceste porturi, care să fie validă în interiorul acestor blocuri. Utilizați pseudocodul de mai jos pentru a rezolva exercițiul și pentru a identifica modul în care a fost rezolvată problema porturilor de tip wire.

```
/*
 * clk - clock
 * oe - output enable, active high
 * cs - chip select, active high
 * we - write enable: 0 = read, 1 = write
 * adresa - adrese pentru 128 de intrari
 * data_in - intrare de date de 8 biti
```

```
 * data_out - iesire de date de 8 biti
 */
module sram (input clk, input oe, input cs, input we, input[7] adresa, input[8] data_in, output[8] data_out)
{
    registru[8] memorie
    registru[8] buffer

    always (clk)
        if cs:
            if we:
                memorie[adresa] = data_in
            else:
                buffer = memorie[adresa]
            end if
        end if
    end always

    if oe & !we:
        data_out = buffer
    else:
        data_out = 8'bz
    end if
}
```

Task 03 (2p) Magistrale de date multiplexate. Testare. Implementarea anterioară de SRAM folosește porturi separate pentru `data_in` si `data_out` deși, din punct de vedere logic, acestea nu pot fi active și valide simultan (linia `write enable` poate fi 0 sau 1, dar nu ambele în același timp). Știind că în Verilog pinii pot fi declarați atât `input`, `output`, cât și `inout` pentru un pin bidirectional, **reproiectați** memoria SRAM.

```
module sram (
        ...
        inout  wire [7:0] data     // magistrala de date de 8 biti bidirectionala
    );
endmodule
```

Tineți cont că, în starea de input, modulul `sram` ar trebui să lase linia `data` în impedanță mărită (sa nu tragă/dea curent), pentru a nu produce coliziuni cu un eventual driver care vrea să scrie o valoare pe acești pini.

Hint: Puteți folosi modulul implementat la exercițiul anterior.

Implementați modulul de testare pentru memoria SRAM proiectată.

Trebuie să stabiliți un protocol prin care cele doua module care comunică prin firul `data` (modulul SRAM și modulul de test) să nu vorbească în același timp, altfel vor apărea coliziuni.

Hint: Atunci când semnalul `we` (write enable) este activ driver-ul semnalului `data` este modulul conectat la memorie (el este cel care scrie în memorie, deci impune un set de date pe linia de date). Atunci când `we` este inactiv driver-ul este însăși memoria, fiindcă a fost solicitată o dată care trebuie pusă pe linia de date.

Codul din memorie va trebui să fie asemanator cu:

```
assign data = (we == 0) ? buffer : 8'bz;
```

Iar cel din driverul care interfațează memoria (sau din modulul de test):

```
assign data = (we == 1) ? buffer : 8'bz;
```

Nu uitați să țineți cont și de `oe` în modulul SRAM!

Atenție! În modulul de test `data` va fi acum un wire. Asta înseamna că nu vom mai putea face atriburi la `data` în interiorul blocului `initial…begin`. Va trebui să găsiți o altă metodă de a-l modifica pe `data`, care

să fie validă în interiorul acestui bloc. Identificați cum rezolvă modulul SRAM aceasta problema și folosiți o logică similară.

Task 04 (6p)

Creați un modul FSM care preia datele din memoria ROM sau SRAM.

- Dacă datele sunt găsite în SRAM, acestea sunt puse pe linia de ieșire;
- Dacă nu, acestea sunt căutate în ROM și puse pe linia de ieșire.
    - În plus, valoarea va fi preluată și în memoria SRAM.

Hint: Diagrama de stări poate fi:

```
      +----------  (STATE_SRAM_READ) <------------+
      V                                           |
>(STATE_IDLE) -----> (STATE_SRAM_READ_INIT) ------+
```

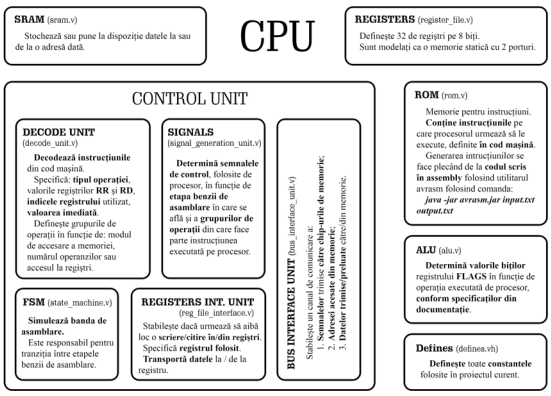
- STATE_IDLE: verifică existența unei adrese la intrare, caz în care trece la următoarea stare;
- STATE_SRAM_READ_INIT: inițiază procesul de citire din memoria SRAM;
- STATE_SRAM_READ: preia datele din SRAM
    - Dacă nu există date în SRAM, se vor prelua cele din ROM și se va iniția procesul de scriere în SRAM;
    - Altfel, se trimit datele din SRAM pe ieșire.

Semnalul `address` asigură legătura dintre module. `address` este **intrare** pentru FSM și modulele de memorie. `data` este **ieșire** pentru FSM și reprezită valorile preluate din SRAM sau ROM.

Pentru a urmări mai ușor comportamentul modulului, deschideți în simulator fișierul `.wcfg` din scheletul laboratorului.

Resurse

Understanding SRAM Operation [http://www.archive.ece.cmu.edu/~ece548/localcpy/sramop.pdf]
SRAM [http://en.wikipedia.org/wiki/Static_random-access_memory]
OpenCores coding guidelines [http://cdn.opencores.org/downloads/opencores_coding_guidelines.pdf]
SRAM lecture [http://www-inst.eecs.berkeley.edu/~cs150/sp12/agenda/lec/sram2-proj2.pdf]
Coding RAM in Verilog [http://stackoverflow.com/questions/7630797/better-way-of-coding-a-ram-in-verilog]
Single-Port RAM [http://www.altera.com/support/examples/verilog/ver-single-port-ram.html]
Single clock Synchronous RAM [http://www.altera.com/support/examples/verilog/ver-single-clock-syncram.html]
Altera HDL coding guidelines [http://www.altera.com/literature/hb/qts/qts_qii51007.pdf]
Schelet laborator
Cheatsheet:



---

1) pret vs dimensiune vs performanta, choose any two, but not three

[2] dielectricul din care sunt fabricate condensatoarele nu este izolator perfect si exista curenti de scurgere
[3] totul in viata se plateste
[4] care nu e nici static nici dinamic, asta e un moment pentru asistent
[5] se duce si mai mult de hard-disk, de ex: tapes

cn2/laboratoare/01.txt · Last modified: 2019/10/01 12:56 by tudor.visan

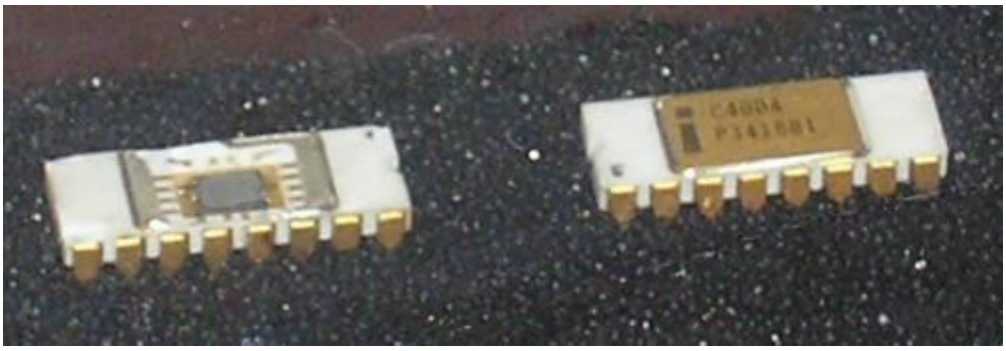
Laboratorul 02 - ISA 1: Operații simple

În acest laborator vom învăța ce este un procesor și un ISA. Vom învăța cum funcționează arhitectura AVR și vom începe implementarea procesorului nostru prin a decodifica și executa câteva operații simple.

Microprocesorul

Microprocesorul este o componentă principală a unui sistem de calcul care are rolul de a executa instrucțiunile unui program. Instrucțiunile ce pot fi rulate de către un procesor sunt de mai multe tipuri:

- Aritmetice/Logice
- De control
- De transfer de date



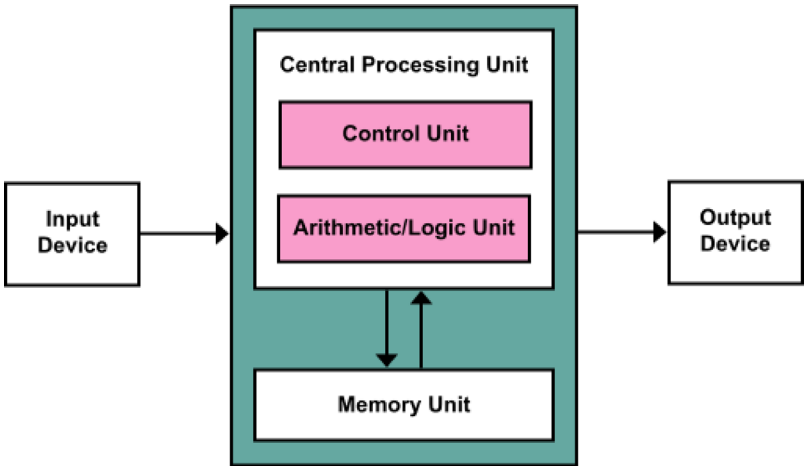
Cele mai multe procesoare, dar și cele mai puțin puternice, sunt cele făcute pentru sisteme încorporate (embedded), apoi cele pentru sisteme personale iar cele mai puține, dar și cele mai puternice, sunt procesoarele pentru sisteme specializate și servere.

Arhitecturi de microprocesoare

O caracteristică foarte importantă a unui procesor este numărul de biți folosiți pentru a reprezenta un număr întreg (lungimea unui cuvânt sau lățimea procesorului). Aceasta influențează câți biți pot fi citiți/scriși într-o singură operație de citire/scriere și, în general, câtă memorie poate fi adresată direct. Cele mai întâlnite arhitecturi din ziua de azi sunt pe 32 și pe 64 de biți dar există și procesoare pe 4, 8, 12, 16 sau 24 de biți.

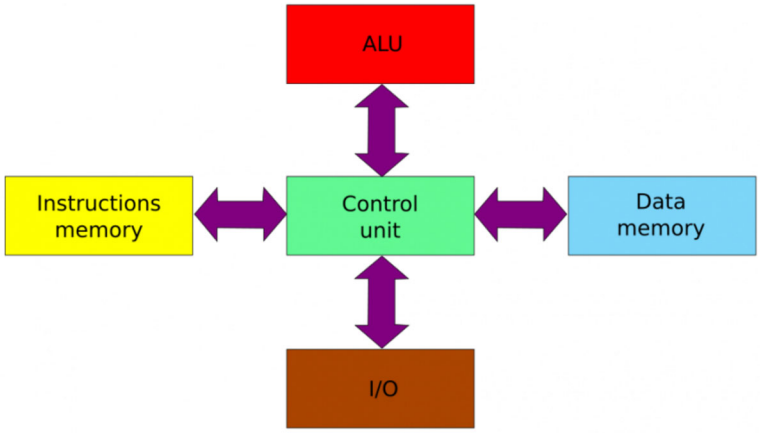
Arhitectura Von Neumann pentru sisteme de calcul

Introdusă în anul 1945 de către matematicianul american John von Neumann [http://en.wikipedia.org/wiki/John_von_Neumann], această arhitectură descrie un sistem de calcul ce conține o unitate de procesare, o memorie pentru instrucțiuni și date, și unități de intrare/ieșire. Astfel, un procesor pentru o astfel de arhitectura va avea o singura magistrală de adrese și o singură magistrală de date, pe care vor circula și date și instrucțiuni. Avantajele acestei arhitecturi sunt simplitatea și faptul că, fiindcă instrucțiunile și datele se află în aceeași memorie, codul se poate auto-modifica.



Arhitectura Harvard pentru sisteme de calcul

Spre deosebire de Von Neumann, arhitectura Harvard descrie un sistem de calcul unde memoria de date și memoria de program sunt separate. Asta înseamnă că procesoarele trebuie sa aibă două magistrale de date și de adrese: una pentru instrucțiuni și una pentru date. Avantajul este că aceste magistrale nu trebuie să aibă aceeași dimensiune, deci putem avea un procesor pe 8 biți care adresează mai mult de 256 de octeți de memorie. Un alt avantaj este ca memoria de program poate fi facută non-volatila, deci odată scris un program acesta nu mai trebuie reîncărcat la fiecare pornire.



Componentele unui microprocesor

Un procesor este format din patru componente principale:

- **Unitatea aritmetică/logică (UAL)**: execută operațiile aritmetice și logice.
- **Registre**: cea mai mică unitate de stocare ce face parte dintr-un procesor, în general de dimensiunea unui cuvânt. Registrele care pot fi folosite pentru majoritatea operațiilor se numesc *registre de uz general* (e.g. AX, BX, CX, DX pe arhitectura x86). Totuși unele registre au un scop particular. Dintre acestea cele mai importante sunt **Program Counter (PC)**, **Instruction Register (IR)**, **Stack Pointer (SP)** și **Flags Register** (registru ce indică statusul curent al procesorului).
- **Porturi de intrare/ieșire (I/O)**: linii prin intermediul cărora procesorul se interfațează cu periferice (inclusiv cu memoria). Prin intermediul acestora el poate transmite sau recepționa date.
- **Unitatea de control (UC)**: decodifică instrucțiunea curentă și pe baza acesteia și a stării interne, generează semnale de control pentru toate resursele procesorului (e.g. registrele de intrare/ieșire și operația aritmetică necesare unei instrucțiuni) și coordonează activarea acestor resurse.

În funcție de arhitectura procesorului numărul și numele registrelor poate varia.

Microcontroller

Un microcontroller este un chip care *conține un microprocesor*, memorie de date, memorie de program și dispozitive periferice. Accentul în design se pune pe consum redus de energie și costuri mici de producție. De aceea, în general, rulează la frecvente reduse (zeci, sute de MHz). De asemenea pot fi specializate pe o anumită funcționalitate.

Majoritatea microcontrollerelor sunt formate din:

- Unitatea centrală de procesare
- RAM (memorie volatilă) și/sau Flash/EEPROM (memorie non-volatila)
- Porturi de intrare/ieșire
- Timere
- Interfețe seriale și paralele
- Etc.

ATTiny20

ATTiny20 [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf] este un microcontroller produs de firma Microchip. El este construit pe arhitectura AVR [http://en.wikipedia.org/wiki/Atmel_AVR] și are următoarele caracteristici:

- Arhitectura RISC
- 112 de instrucțiuni – majoritatea executate într-un singur ciclu de ceas
- 16 registre generale de 8 biți
- 2048 de octeți de memorie programabilă Flash
- 128 de octeți de memorie SRAM
- 12 pini de intrare/ieșire programabili
- Frecventa de operare de pana la 12 MHz
- Tensiune de alimentare 1.8V - 5V

Arhitectura unui set de instrucțiuni

Orice microprocesor are definit un Instruction Set Architecture (ISA):

- O interfață între hardware și software. Așa cum în OOP rolul unei interfețe este de a oferi o imagine de ansamblu, fără a oferi detalii de implementare, așa și aici ISA definește operațiile, accesul la memorie, modurile de stocare suportate de hardware, fără a da detalii de implementare.
- Definește setul de instrucțiuni recunoscute de către procesor. Orice altă instrucțiune are efecte nedeterminate asupra procesorului.
- Definește tipurile de date care pot fi recunoscute de procesor.
- Definește contextul în care o instrucțiune operează.
- Definește cum o serie de instrucțiuni trebuie să interacționeze între ele (propagare de flaguri, executare de salturi condiționale, etc).

O instrucțiune este reprezentată de un șir de biți, o parte dintre ei fiind codul operatiei. Instrucțiunile pot fi codificate pe un număr constant de biți sau să fie de dimensiune variabilă.

În funcție de ISA numărul, numele, codificarea și funcționarea instrucțiunilor poate varia.

Etapele executării unei instrucțiuni

Orice procesor poate avea un ciclu de prelucrare a instrucțiunilor diferit, în funcție de ISA-ul pe care îl implementează, însă toate vor urma următoarea structură:

- **IF (Instruction Fetch)**: următoarea instrucțiune este adusă din memorie de la adresa către care pointează registrul *Program Counter* (PC) și este stocată în registrul *Instruction Register* (IR). PC este apoi incrementat pentru a pointa către următoarea instrucțiune de încărcat.
- **ID (Instruction Decode)**: instrucțiunea din IR este decodificată.
- **EX (Execute)**: execuția efectivă a instrucțiunii. Aceasta etapă variază în funcție de tipul instrucțiunii curente.
- **MEM (Memory Access)**: ciclu folosit în cazul în care instrucțiunea accesează memoria.
- **WB (Write Back)**: scrierea noilor valori în registre.

Instrucțiuni AVR

Instrucțiunile pentru AVR sunt pe 16 biți sau 32 de biți. Deși procesorul este pe 8 biți, memoria este organizată în rânduri de 16 biți lungime, procesorul putând citi câte un rând la fiecare ciclu de ceas din memoria de program. Astfel, memoria noastră de 2048 de octeți este definită ca 1024 * 2 octeți, adică în 1024 de rânduri, fiecare rând având 2 octeți (16 biți). Exemple de instrucțiuni AVR:

- Aritmetice/Logice
  - ADD
  - SUB
  - MUL
  - AND
  - OR
  - NEG
- De control
  - RJMP
  - BREQ
- De transfer de date
  - LDS
  - MOV
  - STS
  - IN
  - OUT

6. ADD – Add without Carry

6.1 Description

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	$0 \le d \le 31, 0 \le r \le 31$	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

Mai sus puteți vedea un extras din definiția setului de instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]. Observați numele

instrucțiunii (ADD), descrierea funcționalității ei, o descriere matematică a funcționalității, sintaxa în limbaj de asamblare (AVRASM) și codul operației pe 16 biți.

Rd și *Rr* sunt nume generice date celor două registre folosite, în practică ambele pot fi oricare dintre registrele R_0 - R_{31} (0 <= d <= 31, 0 <= r <= 31). Procesorul știe ce registre să folosească prin concatenarea biților marcați cu *d* si *r* din codul operației. Spre exemplu, codificarea operației *ADD* R_{16}, R_1 este *0000 1101 0000 0001*.

TL;DR

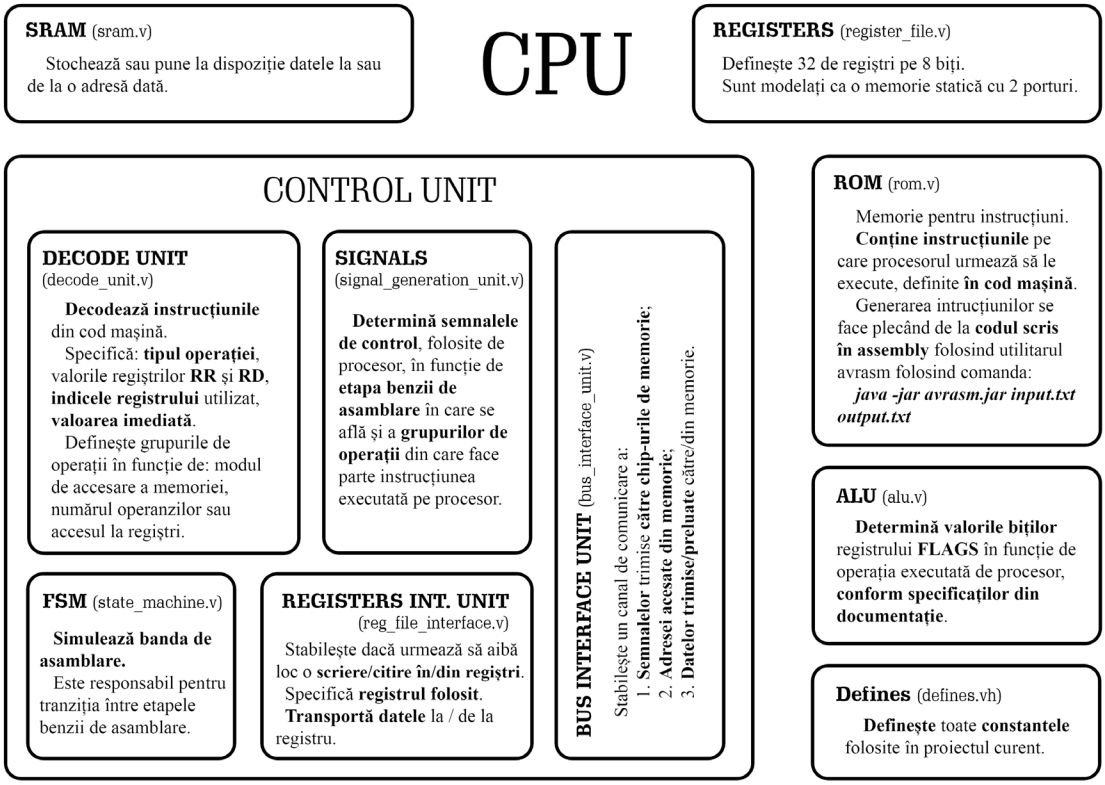
- Procesorul este unitatea principală a unui sistem de calcul.
  - Execută instrucțiunile unui program.
  - Poate avea arhitectură Von Neumann (o singura memorie ce conține instrucțiuni și date) sau Harvard (memoria de instrucțiuni este separată de memoria de date).
- Un microcontroller este un chip ce conține un microprocesor, memorie de date și/sau program și dispozitive periferice.
  - ATtiny20 este microcontrollerul pe care îl vom implementa noi pe parcursul semestrului.
- ISA definește interfața dintre hardware și software.
  - Instrucțiunile recunoscute de procesor și cum funcționează acestea.
  - ISA-ul pentru ATtiny20 este AVR (AVR Instruction Set [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]).
- Instrucțiunile au, în mod general, 5 etape în prelucrarea lor: IF, ID, EX, MEM, WB.
  - Informațiile necesare prelucrării instrucțiunii se găsesc în codul instrucțiunii care, în cazul AVR, poate fi de 16 sau de 32 biti.

Exerciții

Folosiți manualul setului de instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf] pentru a implementa codificările și decodificările comenzilor. Căutați în meniu capitolele aferente fiecărei instrucțiuni.

În scheletul de laborator sunt câteva fișiere de interes:

- **decode_unit.v** se ocupă de decodificarea instrucțiunilor. Aici trebuie să adăugăm instrucțiunile noi.
- **control_unit.v** implementează logica de control. Aici trebuie sa translatăm *type* în *opcode*.
- **alu.v** execută operații aritmetice și logice. Aici trebuie calculate rezultatele operațiilor aritmetice.
- **rom.v** conține codul ce va fi executat.



Dacă implementați complet instrucțiunile necesare, în urma simulării *checker_view.v* o să apară toate semnalele verzi. Codul se află într-o memorie ROM, așadar pentru orice schimbare în cod tot designul trebuie resimulat.

Task 01 (1p) Implementați instrucțiunea NOP.

Task 02 (1p) Implementați instrucțiunea NEG.

Task 03 (2p) Implementați instrucțiunea ADD.

Task 04 (2p) Implementați instrucțiunea SUB.

Task 05 (2p) Implementați instrucțiunea AND.

Task 06 (2p) Implementați instrucțiunea OR.

Resurse

- Schelet laborator
- AVR Instruction Set [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]

cn2/laboratoare/02.txt · Last modified: 2019/10/01 12:58 by tudor.visan

Laboratorul 03 - ISA 2: Acces la Memorie

0. Friendly reminders

Cum citim un datasheet? ATTiny20 [http://www.atmel.com/images/atmel-8235-8-bit-avr-microcontroller-attiny20_datasheet.pdf]

Mai jos avem reprezentarea binară a instrucțiunii `add R5, R3`:

Bit index	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit value	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1	1
Bit significance	opcode						Rd	Rr	Rd				Rr			

Fiindcă register file-ul procesorului nostru (ATtiny20) are doar 16 registre, putem ignora biții `Rr` si `Rd` de pe pozițiile 9 și 8 (Atmel garantează că, din motive de compatibilitate, vor fi mereu setați pe 1, sau, echivalent, vor fi folosite numai registrele `R16` → `R31`). Așadar, instrucțiunile `add R21, R19` și `add R5, R3` sunt echivalente în implementarea noastră.

Mereu filtrați de la instrucțiunile cele mai particulare (cele care au partea constanta cea mai mare) până la cele mai generale (au partea constanta mai mica). Adică? dintre opcode-ul 1) 0000_1111_1010_0101 și opcode-ul 2) 1111_0101_10rr_rddd mai particulara este 1.

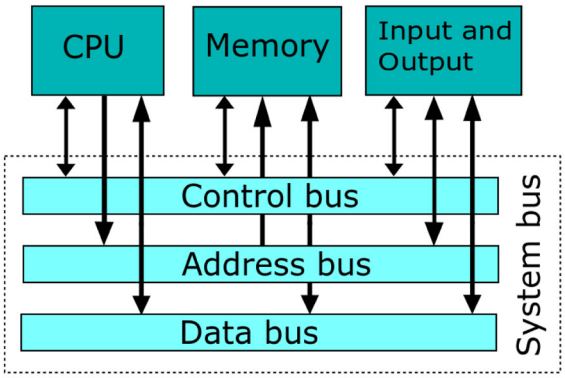
Veți observa că în datasheet veți găsi opcode-uri pe 16 și pe 32 de biți (pentru instrucțiuni). În cadrul laboratoarelor de CN veți folosi doar instrucțiuni pe **16 biți**, deoarece am ales să implementăm o versiune mai restrânsă a procesorului, care nu are acces la toate resursele sale. De exemplu: nu avem 32 de registre locale, ci doar 16.

1. Overview

1.1. Legătura memorie-microprocesor. System bus

Pentru a asigura comunicația dintre procesor, memorie și periferice le vom interconecta pe toate într-o topologie de tip **magistrala**. Aceasta este un mediu partajat de comunicație, în sensul că există un singur set de semnale (de control, date și adrese) la care au acces toate modulele conectate (spre deosebire de o topologie point-to-point unde există o legătură separată între fiecare modul care dorește să comunice cu altul).

Magistrala de sistem [http://en.wikipedia.org/wiki/System_bus] este, pe scurt, o magistrală care conectează procesorul de memoria principala și de periferice. Magistrala de sistem trebuie sa aibă, în general, 3 componente:



- linii de control
- linii de adrese
- linii de date

În general, pentru comunicația mai multor module pe o magistrala, trebuie stabilit un protocol de comunicație (trebuie evitată situația în care există mai mulți vorbitori concomitenți pe aceeași linie, fiindcă acest gen de conflicte [http://wiki.nesdev.com/w/index.php/Bus_conflict] pot duce la scurtcircuite și alte probleme de natură electrică. În practică, există un *bus master* și mai mulți *bus slaves*. Nu întâmplător, în cazul nostru bus master va fi CPU-ul iar, momentan, singurul slave de pe system bus este memoria.

Ierarhii de magistrale

În sistemele desktop, system bus, adică o singură magistrala general-purpose, a fost demult înlocuita cu o ierarhie de magistrale (unele locale, altele externe) de viteze si lățimi de bandă diferite. Astfel, pentru asigurarea conectivității la memorie, un CPU x86 folosește un *memory controller hub*, care se ocupă de interfațarea propriu-zisă (generarea semnalelor de control) pentru memoria SDRAM conectată pe placa de bază.

Istoric, acest controller de memorie s-a aflat pe un chip extern procesorului, dar situat pe placa de baza în apropierea acestuia, numit Northbridge [http://en.wikipedia.org/wiki/Northbridge_%28computing%29]. Magistrala de mare viteza dintre CPU și Northbridge se numește Front-side bus [http://en.wikipedia.org/wiki/Front-side_bus] (sau FSB, pe scurt) pentru sistemele Intel, sau EV6 [http://en.wikipedia.org/wiki/Athlon#General_architecture], pentru implementarea cu același rol de la AMD.

1.2. Moduri de adresare

Modurile de adresare a memoriei descriu metoda prin care este calculată adresa efectivă a datelor accesate, folosind informații constante și/sau conținute în registre. Pentru a fi mai flexibil pentru programatori un ISA va implementa mai multe moduri de adresare a memoriei, fiecare cu avantaje și dezavantaje. Aceste moduri pot fi folosite prin diferite instrucțiuni. Mai departe ne vom uita peste modurile de adresare a memoriei de date disponibile pe arhitectura AVR.

1.2.1. Adresare directa

Adresa de memorie dorită se află chiar în corpul instrucțiunii. Un exemplu de instrucțiune ce folosește adresare directa este LDS (16-bit).

75. LDS (16-bit) – Load Direct from Data Space

Operation:

(i) Rd ← (k)

	Syntax:	Operands:	Program Counter:
(i)	LDS Rd,k	$16 \le d \le 31$, $0 \le k \le 127$	PC ← PC + 1

16-bit Opcode:

1010	0kkk	dddd	kkkk

Biții marcați cu *k* formează o constantă ce reprezintă adresa de memorie de unde sunt citite datele, iar aceste date sunt scrise în registrul Rd.

1010	0kkk	dddd	kkkk
1010	0111	0110	0101

d = 0110 = 6

k = 1110101 = 117

1.2.2. Adresare indirectă

Adresa de memorie dorită se află în registrul X, Y sau Z. O instrucțiune ce folosește adresare indirectă cu deplasament este LDD (i).

Conform datasheetului când ne referim la registrul X, Y sau Z ne referim de fapt la grupări de câte doua registre:

- Registrul X este grupul R27:R26
- Registrul Y este grupul R29:R28
- Registrul Z este grupul R31:R30

Există desigur și variații **Adresare indirectă cu offset:** Adresa de memorie dorită se află în registrul Y sau Z, la care se adaugă un deplasament găsit în corpul instrucțiunii. O instrucțiune ce folosește adresare indirectă cu deplasament este LDD (iv).

71. LD (LDD) – Load Indirect from Data Space to Register using Index Y

Using the Y-pointer:

	Operation:		Comment:
(i)	Rd ← (Y)		Y: Unchanged
(ii)	Rd ← (Y)	Y ← Y + 1	Y: Post incremented
(iii)	Y ← Y - 1	Rd ← (Y)	Y: Pre decremented
(iv)	Rd ← (Y+q)		Y: Unchanged, q: Displacement

	Syntax:	Operands:	Program Counter:
(i)	LD Rd, Y	0 ≤ d ≤ 31	PC ← PC + 1
(ii)	LD Rd, Y+	0 ≤ d ≤ 31	PC ← PC + 1
(iii)	LD Rd, -Y	0 ≤ d ≤ 31	PC ← PC + 1
(iv)	LDD Rd, Y+q	0 ≤ d ≤ 31, 0 ≤ q ≤ 63	PC ← PC + 1

16-bit Opcode:

(i)	1000	000d	dddd	1000
(ii)	1001	000d	dddd	1001
(iii)	1001	000d	dddd	1010
(iv)	10q0	qq0d	dddd	1qqq

Biții marcați cu *q* formează o constantă ce reprezintă deplasamentul ce trebuie adăugat adresei de memorie găsită în registrul Y sau Z. Datele sunt scrie în registrul Rd.

Adresare indirectă cu pre-decrementare

Adresa de memorie dorită se afla în registrul X, Y sau Z, care este decrementată înainte de a fi folosită. O instrucțiune ce folosește adresare indirectă cu deplasament este LDD (iii).

Adresare indirectă cu post-incrementare

Adresa de memorie dorita se afla în registrul X, Y sau Z, care este incrementata după ce este folosită. O instrucțiune ce folosește adresare indirectă cu deplasament este LDD (ii).

2.1 Memoria procesorului

Procesorul pe care îl implementăm la laborator are o arhitectura de tip Harvard [https://en.wikipedia.org/wiki/Harvard_architecture]. Asta înseamnă că vom avea 2 memorii separate pentru instrucțiuni și date.

- **Memoria de instrucțiuni** este de tip ROM (2048 bytes) și se găsește în *rom.v*
  - **adresele sunt pe 8 biți**
  - **datele au 16 biți** (adică lungimea unei instrucțiuni)
- **Memoria de date** este un SRAM sincron de tip pipeline (128 bytes) și se găsește în *sram.v*
  - **adresele sunt pe 7 biți**
  - **datele au 8 biți** (= dimensiunea unui registru)

Procesorul lucrează cu 16 registre "general purpose" de 8 biți. Teoretic există 32 registre, însă implementarea noastră nu va folosi niciodată primele 16. Astfel când scriem cod AVR vom putea folosi numai registrele din intervalul R16 → R31.

Totuși, după cum ați văzut în laboratorul trecut, nu vom lucra direct cu aceste registre, ci vom folosi registrul sursa **Rr** și registrul sursă/destinație **Rd**. Aceștia în spate adresează cele 16 registre. Puteți vedea acest lucru în sursa *register_file.v* în care este implementat un Dual Port SRAM [https://en.wikipedia.org/wiki/Dual-ported_RAM]. Acesta garantează accesul concomitent (într-un ciclu de ceas) atât la Rd, cât și la Rr.

Driver-ul de memorie pentru modulul dual_port_sram este reg_file_interface_unit definit în fișierul *reg_file_interface_unit.v*. Aici vom seta biții de *cs*, *we* si *oe*, precum și liniile de date și adrese ce sunt necesare pentru lucrul cu memoria.

Accesul la o memorie SRAM cu magistrala de date multiplexată se face prin setarea următorilor biți:

- cs : *chip select* stabilește dacă chip-ul este activ sau nu;
- we : *write enable* când este activ, se scriu date în RAM;
- oe : *output enable* când este activ și nu se scrie în RAM, se citesc date din RAM;

În cazul memoriei Dual Port SRAM din laborator, vom avea acești biți atât pentru Rd, cât și pentru Rr (*rr_cs, rd_cs*, etc). Similar, vor exista linii de date și adrese distincte pentru fiecare dintre Rd si Rr.

Reluat: Veți observa în datasheet menționate o serie de registre index: X, Y și Z. Acestea corespund unor perechi de registre generale astfel: XH:XL (R27:R26), YH:YL (R29:R28) și ZH:ZL (R31:R30). În scheletul de cod, aceste registre index sunt definite ca macro-uri în fișierul *defines.vh*.

2.2 Semnale importante în scheletul de cod

Fata de laboratorul anterior, veți găsi o serie de semnale noi ce au fost adăugate unor modulelor procesorului nostru pentru a putea permite implementarea instrucțiunilor de lucru cu memoria. În această secțiune va fi explicat pe scurt rolul acestor semnale noi și utilitatea lor.

1. ***decode.v***
   - *opcode_imd* - asemănător lui *opcode_rr* și *opcode_rd*, acest semnal va reține valoarea decodificata din instrucțiune a valorilor imediate folosite de unele instrucțiuni.
2. ***reg_file_interface_unit.v***
   - *rr_addr* & *rd_addr* - ieșirile modulului care spun ce adresă folosim pentru RD si RR pentru operația curenta. Valorile acestor adrese trebuie stabilite în funcție de tipul operației (scriere sau citire) dar și de modul în care aceasta codifica adresa (indirect, printr-un alt registru sau direct prin OPCODE). O astfel de ieșire este activă doar dacă bit-ul CS corespunzător uneia dintre cele doua este activ.
   - *internal_rr_addr* & *internal_rd_addr* - fire de "lucru" în cadrul modului folosite pentru a determina ce adresă se asignează pe ieșirile *rr_addr* respectiv *rd_addr*.
   - *signals* - un registru ai cărui biți reprezintă ce fel de operație (READ sau WRITE) se execută asupra memoriei sau registrelor (RD, RR sau MEM). Semnificația fiecărui bit din acest registru se poate găsi în *defines.vh*.
3. ***control_unit.v***
   - *writeback_value* - valoarea ce trebuie salvată în memorie la finalul ultimului stagiu de pipeline. În funcție de tipul operației, trebuie asignat semnalul corespunzător. Spre exemplu, operațiile UAL atribuie ALU_OUT_BUFFER.
   - *bus_data* - valoarea aflata pe magistrala de data în urma ultimei operații de citire din memorie.
   - *alu_rr* - valoarea curenta stocata în registrul RR.
   - *alu_rd* - valoarea curenta stocata în registrul RD.

2.3 Instrucțiunile ATTiny pentru acces la memorie

Instrucțiunile pe care le veți implementa în acest laborator diferă putin de cele UAL pe care le-ați întâlnit pana acum. În aceasta secțiune va fi explicat pe scurt cum trebuie citit si interpretat datasheet-ul în cazul acestor instrucțiuni.

1. **LDI** - Fata de instrucțiunile obișnuite, LDI introduce un nou operator: *K*. *K* reprezinta o valoare constanta pe 8 biți ce este salvata în registrul RD.
2. **LDD** - Aceasta instrucțiune este un load indirect și folosește registrul special Y pentru a specifica adresa de memorie de la care se face citirea. În datasheet, veți găsi 4 codificări distincte pentru aceasta

operație. Acest tip de LOAD poate modifica valoarea adresei înainte sau după accesarea efectiva a registrului. Cele 4 variante descrise in datasheet sunt:
    - *LDD Rd, Y* - in aceasta varianta, adresa din registrul Y este nemodificata în urma execuției
    - *LDD Rd, Y+* - Post increment; adresa este incrementata după ce se face accesul
    - *LDD Rd, -Y* - Pre increment; adresa este decrementată înainte de a se face accesul
    - *LDD Rd, (Y+q)* - Displacement; adresa de la care se citește este specificata de registrul Y plus un deplasament (o constanta pe 6 biți codificata în instrucțiune)
3. **LDS** - Instrucțiune pentru load direct din spațiul de date în registru. Asemănător cu LDI, aceasta instrucțiune codifica o constanta *k*, însă aceasta are o semnificație cu totul diferita. Aici, *k* codifica adresa din spațiul de date de la care se va face operația de load. Formula pentru calculul acestei adrese o găsim în datasheet: ***ADDR[7:0] = (~INST[8], INST[8], INST[10], INST[9], INST[3], INST[2], INST[1], INST[0])***. biții 10, 9, 8, 3, 2, 1, 0 din instrucțiune reprezinta biții ce îl codifica pe *k*.
4. **STS** - Scrie dintr-un registru la o adresa în memorie. Calculul adresei se face la fel ca cel pentru LDS.
5. **MOV** - Copiază valoarea unui registru într-un alt registru. Codificarea acestei instrucțiuni este similara cu cea a instrucțiunilor UAL.

Instrucțiunile prezentate pot fi grupate și după tipurile de memorie intre care transfera date, astfel:

- REG-REG: **MOV**
- MEM-REG: **LDD**, **LDS**, **STS**
- REG-IMMEDIATE: **LDI**

TL;DR

Modurile de adresare ale memoriei sunt urmatoarele:

- Adresare directa - adresa de memorie dorita se afla in corpul instructiunii
- Adresare indirecta - adresa de memorie dorita se afla in registrul X,Y sau Z
- Adresare indirecta cu pre-decrementare - ca mai sus, doar ca adresa este decrementata inainte de a fi folosita
- Adresare indirecta cu post-incrementare - la fel ca mai sus, adresa de memorie este incrementata dupa ce este folosita

Semnale pentru implementarea instructiunilor de lucru cu memoria:

```
decode.v
  * opcode_imd - retine valoarea decodificata din instructiune a valorilor imediate
```

```
reg_file_interface_unit.v
  * rr_addr si rd_addr - iesirile modulului care spun ce adresa RD si RR folosim pentru operatia curenta
  * internal_rr_addr si internal_rd_addr - determina ce adresa se asigneaza pe iesirile rr_addr si rd_addr
  * signals - un registru ai carui biti reprezinta ce operatie(READ sau WRITE) se executa asupra memoriei sau registrelor (RD,RR)
```

```
control_unit.v
  * writeback_value - valoarea ce trebuie salvata in memorie la finalul ultimului stagiu de pipeline
  * bus_data - valoarea aflata pe magistrala de date in urma ultimei operatii de citire din memorie
  * alu_rr - valoarea curenta stocata in registrul RR
  * alu_rd - valoarea curenta stocata in registrul RD
```

Instructiunile ATTiny pentru acces la memorie:

- LDI - foloseste un operator, k, ce reprezinta o valoare constanta pe 8 biti salvata in registrul RD
- LDD - load indirect, foloseste registrul special Y pentru a specifica adresa de memorie de la care se face citirea.
- LDS - load direct din spatiul de date in registru. Codifica adresa din spatiul de date cu o constanta k
- STS - scrie dintr-un registru la o adresa de memorie

- MOV - copiaza valoarea dintr-un registru intr-un alt registru

Exerciții

Hinturi generice: Incercati sa folositi semnale cat mai generice. Preferati sa folositi grupuri in loc de tipuri de instructiuni. Checkerul este destul de hard-codat in aceasta faza. Cuvantul asistentului ramane final la corectare.

Task 1 (0.5p). Modificați rom.v

Generați codul mașină corespunzător acestor instrucțiuni folosind tool-ul AVR.

- Comandă rulare: **java -jar avrasm.jar input.txt output.txt**
- avrasm.jar îl găsiți în secțiunea de **Resurse**

```
ldi     r29, 10
sts     10, r29
ldi     r29, 0
ldi     r28, 138
ld      r27, y
lds     r28, 10
mov     r27, r28
```

Task 2 (0.5p). Simulați unitTestCpu.v

Unele zone sunt roșii, pentru a investiga ce mai trebuie implementat:

- De la instanțe și procese, expandați până ajungeți la cpu.v. Adaugați semnalele în fereastră (rclick → add to wave window);
- Re-lansați (Re-launch).

Task 3 (2p). Pentru LDI. Modificati decode_unit.v

- Pentru a scrie în registrul Rd, trebuie să controlam 2 semnale: **writeback_value** (control_unit.v) și **signals[CONTROL_REG_RD_WRITE]** (signal_generation_unit.v)

Task 4 (2p). Pentru STS. Modificați decode_unit.v

- Atenție la calcularea adresei - adică ce valoare atribuim în opcode_imd. Memoria de date e mapată între 0x0040 și 0x00BF.
- Pentru a scrie pe bus, trebuie să controlăm 2 semnale: **signals[`CONTROL_REG_RR_READ]** și **signals[`CONTROL_MEM_WRITE]** (signal_generation_unit.v).
- Inspectați bus_interface_unit.v
- Valoarea scrisă pe bus, este transmisă prin data_to_store din control_unit.v. În acest laborator va fi preluată din variabila de buffer pipeline alu_rr.

Task 5 (2p). Pentru LD. Modificati decode_unit.v

- Trebuie generate mai multe semnale pentru o adresare *indirectă*:
  - CONTROL_REG_RR_READ și CONTROL_REG_RD_READ în etapa de decode (ID)
  - CONTROL_MEM_READ în etapa memory access (MEM)
  - CONTROL_REG_RD_WRITE în etapa register write back (WB)

- Ultimele două semnale sunt active și pentru o adresare directă a memoriei? Modificați semnalele corespunzatoare în signal_generation_unit.v
- Adresa *indirectă* este transmisă la bus_interface_unit prin indirect_address (o cuplare a alu_rr și alu_rd), inspectați control_unit.v
- Valoarea scrisă pe bus este transmisă prin writeback_value din control_unit.v.

Task 6 (2p). Pentru LDS. Modificati decode_unit.v (Atenție la calcularea adresei)

- Pentru a citi de pe bus, trebuie să controlam 2 semnale: **signals[`CONTROL_REG_RD_WRITE]** și **signals[`CONTROL_MEM_READ]**
- Inspectați bus_interface_unit.v.
- Valoarea scrisă pe bus este transmisă prin writeback_value din control_unit.v.

Task 7 (1p). Pentru MOV. Modificati decode_unit.v

- Modificați scheletul astfel încât MOV să fie executată corect.

Greseli comune intalnite in laborator:

- Ati setat opcode_rr in loc de opcode_rd.
- Ati setat opcode_rd in loc de opcode_rr.
- Ati calculat adresa directa considerand 7 biti, in loc de adresa specificata in datasheet. La LDS si STS cautati forma {~8, 8, 10, 9, 3, 2, 1, 0}.
- Ati uitat sa setati grupurile in decode_unit.v.
- Nu ati setat CONTROL_REG_RR_READ si CONTROL_REG_RD_READ pentru toate instructiunile care au nevoie de ele (de ce are nevoie LD Y sa citeasca 2 registre?).
- LDI, STS(16-bit) și LDS(16-bit) folosesc doar 4 biti in loc de 5 biti pentru codificarea opcode_rr/opcode_rd. Totusi, aceste instructiuni pot opera doar pe registri R16-R31, asadar o decodificare corectă seteaza bitul cel mai semnificativ al lui opcode_rr/opcode_rd la valoarea 1.
- Ati setat o valoare gresita in decode_unit.v
- Nu ati adaugat toate semnalele la wave window. Nu ati inspectat unit testele sa vedeti exact care pas returneaza 1'bx, si ce semnale nu sunt pe valorile asteptate.

Resurse

Schelet de cod

[0] We <3 The Datasheet [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]

[1] Datasheet++ [http://ww1.microchip.com/downloads/en/devicedoc/atmel-8235-8-bit-avr-microcontroller-attiny20_datasheet.pdf]

[2] Atmel AVR instruction set [https://en.wikipedia.org/wiki/Atmel_AVR_instruction_set]

Tool generare cod masina AVR

cn2/laboratoare/03.txt · Last modified: 2019/10/18 13:26 by serban_ioan.ciofu

Laboratorul 04 - ISA 3: Instrucțiuni de control și stiva

0. Introducere

În acest laborator vom studia **2 tipuri noi** de instrucțiuni din cadrul ISA: instrucțiuni de control și instrucțiuni care lucrează cu stiva.

1. Instrucțiunile de control sunt importante deoarece determină ordinea în care sunt executate celelalte instrucțiuni din cadrul unui program. Ordinea de execuție este dată de salturi la diferite adrese în memoria ROM unde se află secvențele de program.
2. Instrucțiunile care lucrează cu stiva sunt importante pentru **apelurile de funcții**.

În funcție de modul în care se realizează saltul, **instrucțiunile de control** pot fi împărțite în mai multe categorii:

- **Salturi necondiționate** (`jump`): cele care determină continuarea rulării de la o anumită adresă (instrucțiune)
- **Salturi condiționate** (`branch`/ alegeri): continuarea execuției de la o anumită instrucțiune doar dacă se îndeplinește o anumită condiție
- **Salturi relative**: adresa la care se sare este calculată pe baza valorii curente a PC-ului
- **Salturi absolute**: adresa la care se sare este cea specificată în instrucțiune, indiferent de valoarea PC-ului
- **Apel de funcție/rutină** (`call`, `ret`): un salt special la un anumit grup de instrucțiuni "îndepărtat". După terminarea execuției acelor instrucțiuni, se revine la execuția normală a codului, continuând de la adresa unde s-a facut apelul de funcție. Diferența față de tipurile precedente de salt este că acelea erau "one-way", pe când acesta este și cu revenire.
- **Oprirea rulării**: oprirea necondiționată a execuției (`halt`) este considerată tot o instrucțiune de control.

În continuare, ne vom ocupa de instrucțiunile de salt necondiționat și salt condiționat, dar și de instrucțiunile push și pop.

1. Program Counter (PC)

Pentru a înțelege efectul salturilor și modul în care acestea funcționează, trebuie mai întâi să înțelegem ce este un program counter (**PC**) și care este rolul său.

Secvența de cod pe care o are de executat un procesor se află de cele mai multe ori stocată într-o **memorie de tip ROM** conectată la acesta. Pentru a decodifica și executa corect instrucțiunile din acea memorie, procesorul trebuie să știe tot timpul care este adresa de la care trebuie adusă o instrucțiune în etapa de fetch. Această sarcină este îndeplinită de registrul PC (program counter). Rolul acestui registru este de a reține adresa de la care trebuie adusă următoarea instrucțiune din memorie. Mai este numit și Instruction Pointer deoarece funcționează exact ca un pointer: reține o adresă și ajută procesorul să obțină datele stocate la acea adresă.

În mod normal, de fiecare dată când procesorul se află în stagiul `ID` al pipeline-ului, după ce instrucțiunea a fost adusă pentru decodificare, program counter-ul va fi incrementat. Totuși, există și alte situații în care valoarea program counter-ului poate fi modificată. De cele mai multe ori, instrucțiunile de salt sunt responsabile pentru modificarea PC-ului în afara stagiului de `Instruction Decode`.

2.1 Salturi necondiționate

Salturile necondiționate sunt instrucțiuni ce modifică valoarea Program Counter-ului cu o valoare fixă. Acest tip de salturi nu ține cont de evenimente produse în timpul rulării sau de anumite registre ce țin informații legate de operațiile executate recent de către procesor.

În general, instrucțiunile de salt necondiționate sunt codificate ca instrucțiuni cu un singur operand, operandul fiind o constanta care determină cu cât se modifică Program Counter-ul (peste câte adrese va sări).

Acest tip de salturi este în general folosit atunci când dorim să sărim peste o zonă de memorie deoarece conține implementarea unei rutine pe care nu dorim să o executăm sau dacă dorim să evităm o serie de instrucțiuni ce prezintă un posibil hazard. În limbajele high level, saltul necondiționat este cel mai des reprezentat prin tag-uri `GOTO`.

2.2 Salturi condiționate. Registrul SREG

La polul opus față de salturile necondiționate se află cele condiționate. Acestea au același efect, acela de a modifica valoarea Program Counter-ului, însă nu îl vor modifica de fiecare dată. Atunci când se execută o instrucțiune de salt condiționat, se verifică anumite criterii, în funcție de tipul instrucțiunii. Dacă acele criterii sunt îndeplinite, se va realiza saltul. Altfel, Program Counter-ul continuă să fie incrementat în mod normal, în stagiul `ID`. Salturile condiționale sunt cel mai ușor asemănate cu instrucțiunile de tip `if` din limbajele de programare de nivel înalt.

Deoarece aceste instrucțiuni verifică îndeplinirea unor condiții pentru a executa salturi, este nevoie de un mod de a reține starea parametrilor verificați. De cele mai multe ori, acești parametri reprezintă evenimente ce au avut loc în urma executării unor operații de către UAL:

- rezultatul a fost egal sau nu cu zero
- a avut loc un overflow
- rezultatul este negativ sau pozitiv etc.

Pentru a reține toate aceste evenimente se folosesc `flag-uri` grupate în ceea ce se numește `Processor Status Register` sau `SREG` (este oarecum echivalentul registrului `flags` din arhitectura x86).

Printre flag-urile des folosite din SREG se află:

- `Z` (Zero) - indică dacă rezultatul unei operații aritmetice este zero
- `C` (Carry) - indică faptul că s-a realizat un transport la nivelul bitului cel mai semnificativ in cazul unei operații aritmetice. Altfel spus, a avut loc o depășire în aritmetica modulo N considerată. În procesorul nostru pe 8 biți, `255 + 1`, deși ar trebui să aibă rezultatul 256, de fapt acesta este 0 din cauza aritmeticii modulo 256 (2^8). Pentru a diferenția dintre un 0 apărut "pe bune" și unul cauzat de o depășire, se utilizează acest semnal de carry.
- `V` (Overflow) - arată că, în cazul unei operații aritmetice cu semn, complementul față de doi al rezultatului nu ar încăpea în numărul de biți folosiți pentru a reprezenta operanzii. Cu alte cuvinte, se poate întampla ca adunând două numere pozitive, să obținem unul negativ ($127_{signed} + 1_{signed} = -128_{signed}$), dar și adunând două numere negative să obținem unul pozitiv ($-128_{signed} + (-1)_{signed} = +127_{signed}$). Evident, rezultatul nu este corect în aceste situații, și semnalarea se face prin flag-ul de overflow.
- `N` (Negative) - semnul rezultatului unei operații aritmetice (este pur și simplu valoarea bitului de semn al rezultatului)
- `S` (Sign) - este un flag unic AVR, calculat după formula `S = N xor V`, ce reprezintă "care ar fi trebuit să fie semnul corect al rezultatului". Cu alte cuvinte, dacă `N == 1`, dar și `V == 1`, înseamnă că rezultatul este negativ, dar din cauza unei depășiri. `S` este setat în acest caz pe 0, semnalând că semnul "corect" al operației ar fi trebuit să fie pozitiv.

Scrierea biților cu valorile corespunzătoare din `SREG` revine UAL-ului. La execuția unei operații, se calculeaza și valorile fiecărui flag ce poate fi afectat de acel tip de operație. Practic, ca și la arhitectura x86, putem considera că `SREG` este un registru global, a cărui valoare este setată de ultima instrucțiune aritmetico-logică executată de procesor.

Pentru a vedea specificațiile Atmel asupra `SREG`, consultați datasheet-ul ISA-ului. Modul în care acesta este modificat de fiecare instrucțiune este de asemenea descris în pagina corespunzatoare acesteia.

2.3 Instructiuni de salt in AVR

Există două tipuri de salturi, în funcție de adresa destinație:

- la adrese *absolute*, în care adresa destinație a saltului este conținută în valoarea imediată din instrucțiune
- salturi *autorelative*, în care adresa destinație este codificată ca fiind diferența dintre Program Counter-ul curent (adresa instrucțiunii de salt) și adresa destinației. Avantajul este că, pentru salturi apropiate, se salveaza biți pentru codificarea adresei, dar dezavantajul este că nu se poate sări prea departe.

Pentru AVR, salturile condiționate (branch-urile) sunt întotdeauna autorelative, iar cele necondiționate (jump-urile) vin în două "arome": unele relative (`RJMP` pe 16 biți), și altele absolute (`JMP` pe 32 de biți, pe care este dificil să o

implementăm pe procesorul nostru cu instrucțiuni de 16 biți). În cazul instrucțiunii JMP, adresa destinație este o valoare imediată pe 22 de biți, permițând saltul la orice adresă în spațiul de 4M (de fapt, 8 megabytes, ținând cont că fiecare instrucțiune are 16 biți). Pe de alta parte, RJMP cu cei 12 biți de valoare imediată, poate executa un salt la "doar" +/- 2K instrucțiuni relativ la cea curentă. Încă și mai ciudat, există și CALL/RCALL, prin care se poate specifica adresa unei funcții atât ca valoare absolută, cât și relativă. Considerentele sunt aceleași: economisirea memoriei de instrucțiuni (32 de biți pentru CALL vs 16 pentru RCALL), dar și simplificarea logicii de Instruction Fetch și Instruction Decode.

Din alt considerent, există jump-uri și branch-uri la adrese:

- *imediate*, conținute în instrucțiune
- *indirecte*, conținute într-un registru, care trebuie citit în prealabil.

În acest laborator vom implementa doar salturi la adrese **imediate**.

3. PUSH, POP

Această pereche de instrucțiuni este folosită pentru a pune și a extrage date de pe stivă.

- **PUSH** pune la poziția curentă de pe stivă octetul aflat în registrul **Rr** și apoi decrementează stack pointer-ul cu 1 (**stack[sp- -] = Rr**).
- **POP** incrementează stack pointer-ul cu 1 și apoi pune în **Rd** valoarea de la poziția curentă din stiva (**Rd = stack[++sp]**).

Așa cum este implementată stiva în AVR, index-ul cel mai mare este la baza stivei, iar cel mai mic în vârful acesteia.

> **Stack Pointer**
>
> The stack is mainly used for storing temporary data, local variables and return addresses after interrupts and subroutine calls. The Stack Pointer registers (SPH and SPL) always point to the top of the stack. Note that the stack grows from higher memory locations to lower memory locations. This means that the PUSH instructions decreases and the POP instruction increases the stack pointer value.

- Inițial, stack pointer-ul se află la **baza stivei** (valoarea cea mai mare - în schelet: **define STACK_START 8'hBF**).
- Atunci când se pune un octet pe stivă, stack pointer-ul este mutat către vârf, adică se decrementează.
- Dacă realizăm operația inversa, stack pointer-ul se mută către bază, deci este incrementat.

- Stack pointer-ul se va afla tot timpul pe poziția liberă cea mai apropiată de baza stivei.
- În acest laborator stiva va fi mapată peste spațiul de adresă **0x40 - 0xBF**.

TL;DR

1. PC

Registrul PC sau **program counter** (PC) reține adresa de la care trebuie adusă urmatoarea instrucțiune din memorie. Mai este numit și Instruction Pointer (IP) deoarece funcționeaza exact ca un pointer: reține o adresă și ajuta procesorul să obțină datele stocate la acea adresă.

2. Control flow

În funcție de modul în care se realizează saltul, instrucțiunile de control pot fi împărțite în mai multe categorii:

- Salturi necondiționate (jump): cele care determină continuarea rulării de la o anumita adresă (instrucțiune)
- Salturi condiționate (branch/alegeri): continuarea execuției de la o anumită instrucțiune doar dacă se îndeplinește o anumită condiție
- Salturi relative: adresa la care se sare este calculată pe baza valorii curente a PC-ului

- Salturi absolute: adresa la care se sare este cea specificată în instrucțiune, indiferent de valoarea PC-ului
- Apel de funcție/rutină (`call`, `ret`)
- Oprirea rulării (halt)

3. Stack

SP sau stack pointer-ul se va afla tot timpul pe **poziția liberă** cea mai apropiată de baza stivei.

PUSH, POP - această pereche de instrucțiuni este folosită pentru a pune și a extrage date de pe stivă.

- PUSH pune la poziția curentă de pe stivă octetul aflat în registrul Rr și apoi decrementează stack pointer-ul cu 1 (**stack[sp−] = Rr**).
- POP incrementează stack pointer-ul cu 1 și apoi pune în Rd valoarea de la poziția curentă din stiva (**Rd = stack[++sp]**).

În acest laborator stiva va fi mapată peste spațiul de adresă **0x40 - 0xBF**. Inițial, stack pointer-ul se află la baza stivei (valoarea cea mai mare - în schelet: define STACK_START 8'hBF).

Exerciții

În laboratorul de astăzi vom implementa câteva dintre instrucțiunile de jump și branching menționate. Pentru fiecare instrucțiune va trebui să aduceți modificări următoarelor fișiere:

- `decode_unit.v` - aici se implementează logica pentru decodificarea instrucțiunilor
- `control_unit.v` - controller-ul este cel care se ocupă de modificarea program counter-ului. Pentru instrucțiunile de branch și jump, considerăm că această modificare se face în starea de EXECUTE
- `alu.v` - implementarea logicii de calcul a flag-urilor verificate în cadrul operațiilor de branch.
- `signal_generation_unit.v` - Analizați cum sunt setate semnalele ``CONTROL_STACK_POSTDEC`` și ``CONTROL_STACK_PREINC``.

- Folositi **datasheet-ul** pentru a identifica opcode-ul instructiunilor si bitii corespunzatori operanzilor.
- ATENȚIE! Pentru a beneficia de mesajele de debug puse în checker va trebui să lăsați ROM.v neschimbat! (P.S. **Puteți adăuga** la sfârșit, **începând cu adresa 16**.)
- Implementați doar instrucțiunile menționate în enunțurile din acest laborator. Instrucțiunile din laboratoarele anterioare sunt **deja** implementate, acesta fiind motivul pentru care checkerul arată deja **OK** în dreptul acestora.

- Tool java pentru a genera cod mașină AVR
- java -jar avrasm.jar input.txt output.txt

- **Pentru verificarea** corectitudinii exercițiilor **consultați log-urile simulării**. Semnalul nu va fi complet verde, însă checker-ul va arăta **mesajul OK peste tot** dacă exercițiile au fost rezolvate corect.

Task 1 (2 x 2p). Implementați instrucțiunile **PUSH** și **POP**.

- Considerăm că stiva începe la **STACK_START** (0xBF - definit în schelet) și crește spre adresa 0.
- Dacă ați implementat **corect PUSH**, atunci checker-ul o să vă dea OK pentru instrucțiunile PC == 2 și PC == 3.
- Dacă ați implementat **corect POP**, atunci checker-ul o să vă dea **OK aproape peste tot**. Analizați codul din rom.v și **găsiți o explicație pentru acest comportament**.
    -

Task 2 (2p). Implementați instrucțiunea `RJMP` - salt necondiționat. Această operație are un singur operand: numărul de instrucțiuni peste care trebuie să sară program counter-ul.

- Pentru testare implementați și BRBS și BRBC. Apoi folosiți checker-ul și verificați-le pe toate.

Task 3 (2p). Implementați instrucțiunea BRBS (**BR**anch if **B**it in SREG is **S**et). BRBS este un exemplu de instrucțiune de control cu 2 operanzi. Al doilea operand este bit-ul din SREG corespunzător flag-ului pe care dorim să îl verificăm.

Această instrucțiune va face un salt **doar dacă bit-ul selectat este 1**.

- Pentru testare implementați și BRBC. Apoi folosiți checker-ul și verificați-le pe toate.

- Dacă alegeți să folosiți BRMI pentru codul de testare, **NU trebuie** să implementați această instrucțiune.
- **BRMI** este un caz special de BRBS (**BRMI – Branch if Minus**). Căutați în datasheet diferența.
- avrasm.jar va interpreta BRMI ca pe un BRBS cu anumiți biți setați corect. Va genera **o instrucțiune BRBS**.

Task 4 (2p). Implementați instrucțiunea BRBC (**BR**anch if **B**it in SREG is **C**leared). BRBC este un alt exemplu de instrucțiune de control cu 2 operanzi. Operandul suplimentar este bit-ul din SREG corespunzător flag-ului pe care dorim să îl verificăm. Această instrucțiune va face un salt **doar dacă bit-ul selectat este 0**.

- Pentru testare folosiți exercițiul 5.

- Dacă alegeți să folosiți BRPL pentru codul de testare, **NU trebuie** să implementați această instrucțiune.
- **BRPL** este un caz special de BRBC (**BRPL – Branch if Plus**). Căutați în datasheet diferența.
- avrasm.jar va interpreta BRPL ca pe un BRBS cu anumiți biți setați corect. Va genera **o instrucțiune brbc**.

Task 5 (BONUS 2p). Scrieți codul ASM corespunzător unei bucle FOR ce realizează 3-4 iterații în care să utilizați instrucțiunile **implementate** în cadrul exercițiilor aterioare (**BRBS/BRBC, RJMP, PUSH, POP**). **Generați codul mașină** și copiați-l în memoria ROM. Ce trebuie să mai modificați în rom.v?

Exemplu:

```
    ldi r16, 5
    ldi r17, 15
    push r16
    push r17
    main_loop:
        mov r30, r16
        sub r30, r17
        breq gigel_is_done
        brmi r17_is_greater
    r16_is_greater:
        sub r16, r17
        rjmp main_loop
    r17_is_greater:
        sub r17, r16
        rjmp main_loop
    gigel_is_done:
        push r16
        pop r20
        pop r21
        pop r22
```

Task 6 (BONUS 2p). În exemplul oferit în cadrul exercițiului 5 a fost folosit un registru temporar r30 pentru stocarea variabilei din care scădem. Scăderea este necesară pentru a decide care dintre numere este mai mare. Găsiți o soluție să scăpați de acest registru temporar.

- **Hint: CP**

Resurse

[0] Schelet de cod

[1] We <3 The Datasheet [http://www.atmel.com/images/Atmel-0856-AVR-Instruction-Set-Manual.pdf]

[2] avrasm - tool java pentru a genera cod mașină AVR.

cn2/laboratoare/04.txt · Last modified: 2019/10/01 12:59 by tudor.visan

Laboratorul 05 - ISA 4: Apelul funcțiilor

În laboratorul 4 [https://elf.cs.pub.ro/cn/wiki/lab/cn2/lab04] am menționat că din categoria instrucțiunilor de salt fac parte instrucțiunile ce apelează rutine și instrucțiuni care lucrează cu stiva. În continuare vom studia aceste instrucțiuni, dar și alte instrucțiuni și noțiuni necesare atunci când lucrăm cu funcții.

1. Funcții

Funcțiile sunt un element important în orice limbaj de programare deoarece oferă suport pentru abstractizarea și refolosirea codului. Pentru a folosi o funcție, avem nevoie de prototipul ei - numele, numărul și tipul argumentelor, respectiv tipul valorii întoarse. Pentru a vedea cum se implementează funcțiile și apelul lor, trebuie întâi să stabilim ce se întâmplă atunci când apelăm o funcție. Vom lua un exemplu simplu: o funcție recursivă scrisă în C care calculează factorialul unui număr:

```
int fact(int n)
{
    if (n == 0)
        return 1;
    else
        return fact(n - 1) * n;
}
```

Se declară parametrul **n**, de tip întreg. Această variabilă și oricare altă variabilă pe care am folosi-o trebuie să fie locală funcției și să nu existe conflicte cu alte variabile din alte funcții, chiar dacă acolo se vor folosi aceleași nume ca aici. De fapt, mai mult decât atât, pentru aceste variabile nu trebuie să existe conflicte nici măcar cu alte copii ale lor (exact cum se întâmplă la un apel recursiv).

Pentru a putea implementa așa ceva, pentru fiecare apel al funcției **fact**, se realizează o copie într-o zonă de memorie diferită pentru variabila n, la fel ca și pentru tot corpul funcției. Vom avea nevoie de o zonă continuă de memorie, separată de restul programului, în care vom realiza aceste operații cu funcții și pe care o vom numi stivă, deoarece implementarea ei este asemănătoare cu a structurii de date cu același nume.

2. Conventii de apelare

Diferite arhitecturi oferă diferite moduri in care trebuie implementat apelul unei funcții. Mențiuni importante sunt:

- Unde sunt stocați parametri, valorile de return, și adresa de return.
- Ordinea in care argumentele sunt transferate funcției
  - Pe stivă
  - Prin regiștri
  - Prin memorie
  - Un amestec din metodele de mai sus
- Cum se alocă și cum se dezalocă contextul unei funcții si ce asistență trebuie sa ofere funcția apelatoare și funcția apelată.
  - Dacă se stochează valoarea de pointer la stiva pentru a usura reconstructia valorii de stiva la intrarea in funcție.

Exemple de convenții:

X86:

- Toți parametrii funcției apelate sunt pe stivă
- Adresa de return este pe stivă

- Valoarea de return se face tipic prin eax. Dacă valoarea de return este prea mare, funcția apelatoare trebuie să aloce spațiul necesar.
- Se folosește tipic registrul ebp pentru a salva starea registrului esp și pentru a il reface la sfârșitul apelului.
- Funcția apelată poate, dar nu e obligată să facă clean la parametrii de pe stivă (eg: ret 12 va șterge 3 parametri de pe stivă)

ARM(32):

- Registrii r0-r3 vor conține primii patru parametrii, restul se vor transfera pe stivă. De asemenea valoarea de return se face tot prin r0-r3. Dacă valoarea de return este prea mare, funcția apelatoare trebuie să aloce spațiul necesar.
- Regiștrii r4-r11 vor conține variabile temporare, dacă avem nevoie de mai multe, le vom stoca pe stivă.
    - Funcția apelată trebuie să salveze și să repare starea regiștrilor r4-r11
- r13 este folosit ca sp
- r14 este folosit să salvăm adresa de return. Dacă o funcție poate apela alte funcții, valoarea lui r14 trebuie salvată și restaurată
- r15 este folosit ca PC.

3. Stiva/Convenția de apelare din laborator

Stiva este o zonă rezervată din memorie, continuă, cu ajutorul căreia este implementată ideea de funcție și apel de funcție.

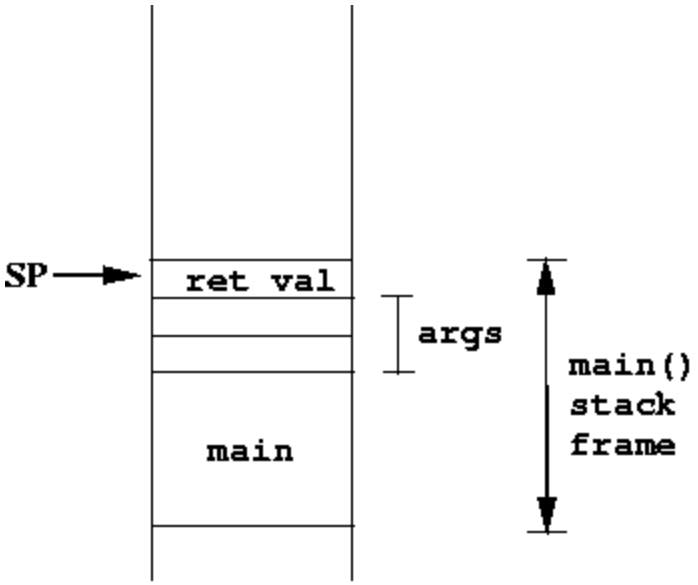
Implementarea efectivă a unui apel

Se alege prima oară o zonă de memorie arbitrară care va fi stivă. Pentru fiecare apel de funcție, va exista o secțiune pe stiva rezervată pentru acea funcție, numită “stack frame”. Să luăm următorul exemplu de cod în C:

```
int main(void)
{
    int a = 5, b = 7, c;
    c = foo(a, b);
    return 0;
}

int foo(int a, int b)
{
    return a + b;
}
```

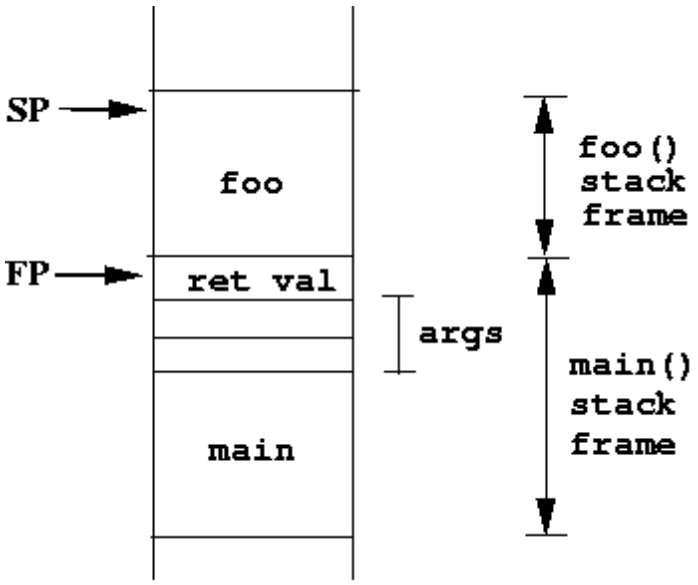
La intrarea în main(), stack-ul va conține doar frame-ul funcției main. Apoi, se ajunge la apelul funcției foo(), care ia 2 argumente. Metoda folosită de a trimite argumentele din main în funcția foo este să le încarci pe stivă, lăsând spațiu totodată și pentru valoarea de return a funcției foo(), spațiu din memorie care va fi încărcat odată ce foo() se termină. Stiva arată acum așa:



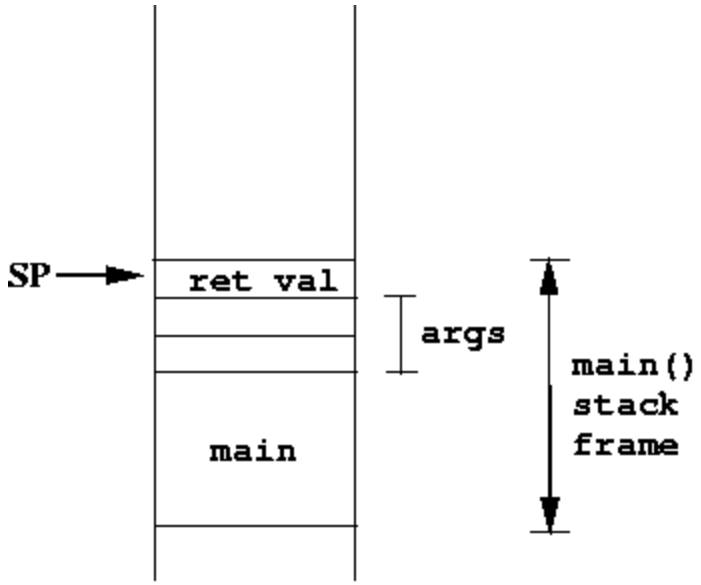
După cum se vede, după ce am încărcat pe stivă argumentele și valoarea de return pentru funcția pe care o vom apela, "stack frame-ul" pentru main a crescut . Practic ce s-a întâmplat a fost să micșorăm valoarea lui SP (stack-pointer - care de obicei arată ori spre prima zonă de memorie neinițializată/nefolosită din stivă, ori spre ultima zonă inițializată/folosită)

Argumentele și valoarea de return a unei funcții nu fac parte din stack frame-ul acesteia!

Când ajungem în corpul funcției foo(), aceasta e posibil să aibă nevoie de spațiu pentru variabile locale (în cazul de față nu), caz în care se modifică SP-ul pentru a crește "stack-frame-ul" lui foo() (operație echivalentă cu un PUSH). Stiva arată acum așa:



Se observă că a apărut un nou pointer, FP - frame pointer, care arată spre zona de memorie unde arată SP-ul înainte ca foo() să îl mute pentru a face loc variabilelor sale locale. Acest nou pointer ne va ajuta sa calculam locația în memorie pentru argumentele sau variabilele locale funcției (calculele de adresă pentru diferite variabile se vor face relativ la acest FP). Când funcția foo() s-a terminat, tot ce trebuie făcut este ca SP-ul să fie pus să arate spre locație de memorie indicată de FP (echivalent cu un POP). Astfel, după ce ieșim din foo(), stiva arată la fel cum arata înainte de a apela foo(), doar ca acum zona de memorie care a fost rezervată pentru valoarea de return o conține pe aceasta:



După ce main() salvează valoarea de return, crește SP-ul astfel încât după ce se face POP la aceasta și la argumente, stiva va arăta exact ca la început.

4. Instrucțiuni AVR

Instrucțiunile ce ne interesează în mod special sunt RCALL și RET.

RCALL

RCALL este un salt relativ de la adresa curentă la adresa specificată ca parametru. Atunci când se realizează un apel de funcție folosind RCALL, adresa instrucțiunii ce urmează după RCALL este salvată pe stivă. Această adresă se numește `return address`. Este important să memorăm această adresă deoarece, la ieșirea din funcție, dorim să continuăm execuția normală a programului. În plus, la realizarea unui apel de funcție, pe stivă se rezervă 2 octeți - (`return address` are 16 biți).

Motivul pentru care RCALL realizează un salt relativ și reține adresa de revenire pe stivă este următorul: codul unei funcții/proceduri se poate afla oriunde în memorie. Aceasta poate fi situată la începutul sau la sfârșitul memoriei, într-o zonă în care există deja și o parte din codul unei alte funcții (spre exemplu `main()`) sau într-o zonă aleatoare izolată.

Datasheet-ul AVR oferă două implementări pentru RCALL: cu program counter pe 16 (I) și 22 (II) de biți. Implementarea folosită de noi este cea cu program counter pe 16 biți!

RET

Instrucțiunea RET realizează revenirea dintr-o funcție la codul de unde a fost apelată aceasta. Practic, marchează sfârșitul zonei de memorie de unde se citește și execută codul funcției.

RET este o instrucțiune cu 0 operanzi. Operanzii nu sunt necesari deoarece adresa la care trebuie să revină PC-ul pentru a continua execuția normală a programului se află salvată deja pe stivă. În urma apelului RET, se vor elibera 2 octeți de pe stivă (pentru varianta cu program counter pe 16 biți).

Atenție! Dacă cineva modifică adresa de revenire salvată pe stivă la apelul unei funcții, programul va continua de la acea adresă modificată dacă ea este validă. Astfel, o funcție poate fi forțată să apeleze alte funcții la revenire.

TL;DR

- Funcțiile oferă suport pentru: abstractizare și refolosire cod.
- Stiva e o zonă continuă de memorie cu ajutorul căreia se implementează apelurile de funcții.
- Fiecare apel de funcție folosește o zonă de pe stivă numită stack frame.
- Argumentele și valoarea de return a unei funcții nu fac parte din stack frame-ul acesteia!

În acest laborator vom implementa următoarele instrucțiuni (pe 16 biți):

- **RCALL** Salt relativ la o adresă (apelarea unei funcții) și salvarea (pe stivă) a următoarei instrucțiuni (return address)
  - Sintaxă: `rcall k`
  - Program Counter: PC ← PC + k + 1 ; -2048 < k < 2047
  - Stack Pointer: SP ← SP - 2
- **RET** Revenirea dintr-o funcție
  - Sintaxă: `ret`
  - Program Counter: PC ← return address (de pe stivă)
  - Stack Pointer: SP ← SP + 2

Exerciții

Scheletul laboratorului conține un checker pentru verificarea Taskului 1 și 2.

Punctajul maxim pentru Task1 și Task2 se obține doar dacă **nu** apar în consola ISim teste picate (FAILED) **și** dacă ordinea de execuție a instrucțiunilor de mai jos (deja scrise in `rom.v`) este:

0,1,4,5,2,3,6

```
0:ldi   r16, 5
1:rjmp  main_function
first_function:
2:ldi   r17, 15
3:ret
main_function:
4:ldi   r17, 10
5:rcall first_function
6:ldi   r18, 20
```

Analizați fișierul `state_machine.v`. Cum s-a modificat stagiul MEM al benzii de asamblare pentru instrucțiunile RCALL și RET?

Task 1 (2P) Implementați instrucțiunea RCALL. Folosiți varianta (I) a instrucțiunii (pentru dispozitive cu PC pe 16 biți).

RCALL este o combinație între PUSH și RJMP. Problema este că registrul PC, care trebuie salvat pe stivă, are, în implementarea noastră particulară, 10 biți, iar PUSH salvează numai 8. Pentru a putea salva toți biții necesari RCALL trebuie să facă două accese la memorie, vedeți `TWO_CYCLE_MEM` și `cycle_count`.

Urmăriți comentariile marcate `TODO 1` din fișierele `decode_unit.v`, `control_unit.v` și `signal_generation_unit.v`.

Task 2 (2P) Implementați instrucțiunea RET. Folosiți varianta (I) a instrucțiunii (pentru dispozitive cu PC pe 16 biți).

RET este o combinație între POP și un salt absolut. Urmăriți comentariile marcate `TODO 2` din fișierele `decode_unit.v`, `control_unit.v` și `signal_generation_unit.v`.

Task 3 (2P) Creați un program pentru testarea instrucțiunilor implementate mai devreme. Programul trebuie să apeleze o funcție care încarcă constanta 66 în registrul R20). Scrieți programul în fisierul `rom.v` folosind tool-ul avrasm.jar și executați programul.

java -jar avrasm.jar input.txt output.txt

Task 4 (2P) Creați un program care calculează diferența a două numere folosind o funcție `dif` ce primește doi parametrii. Înainte de apel se vor pune pe stivă cei doi parametrii ai funcției. Funcția `dif` va lua parametrii de pe stivă **fără a modifica** valoarea return_address (salvată tot pe stivă). Funcția `dif` va pune rezultatul în registrul R16.

Exemplu cod C:

```
int diff (int a, int b) {
  return a-b;
}

int main () {
  int a, b, d;
  a = 10;
  b = 2;
  d = diff (a, b);
}
```

Task 5 (1P) Creați un program care calculează suma primelor `n` numere naturale și o stochează în registrul R20. Programul va citi parametrul `n` de la o adresa fixa de memorie (pe care o stabiliți când creați programul). Trebuie sa calculați suma folosind o funcție iterativa (`sum(n) = n + (n-1) + (n-2) + … + 1`). Începeți prin scrie codul pe hârtie și rulați-l manual (tot pe hârtie) pentru un număr mic.

Task 6 (1P) Refaceți exercițiul anterior folosind o variantă recursivă de calcul (`sum(n) = n + sum(n - 1)`).

Resurse

- Datasheet ATTiny20 [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf]
- Setul de Instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]
- avrasm - tool java pentru a genera cod mașină AVR.
- Scheletul laboratorului

cn2/laboratoare/05.txt · Last modified: 2019/11/04 17:13 by tudor.visan

Laboratorul 06 - GPIO 1: Ieșiri

În acest laborator vom învăța ce este GPIO, cum este implementat în microcontroller-ul ATTiny20 și îl vom adăuga implementării noastre a acestui controller.

1. Ce este GPIO?

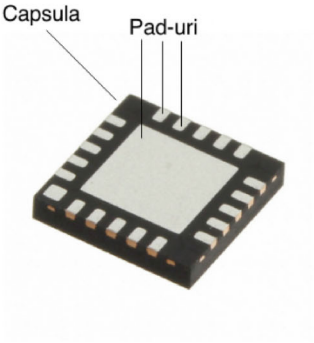
GPIO (General-Purpose Input/Output) este calitatea unui pin al unui circuit integrat de a-i putea fi controlat comportamentul, adică direcția de trecere a curentului electric prin el și valoarea acestuia, într-un mod programatic. Astfel, noi putem să configurăm un pin pentru a '**scrie**' o valoare, caz în care pinul devine o sursă de '0' sau de '1' logic, controlată prin registre (practic legăm pinul la VCC sau GND cu ajutorul unor tranzistori). În acest mod putem, spre exemplu, să aprindem un led folosind instrucțiuni.

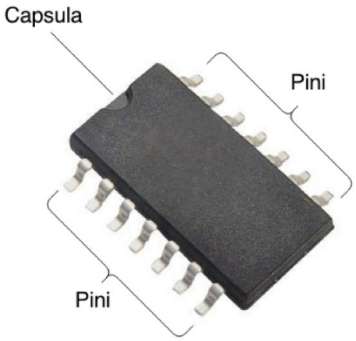
Un pin GPIO poate fi folosit, de asemenea, și pentru '**citire**'. Astfel, putem afla dacă pinul este comandat din exterior către '0' sau '1' logic.

Atenție! Un pin configurat pentru citire care nu este comandat din exterior (este lăsat 'în aer'), nu va avea o valoare logică concretă de '0' sau '1', iar citirea lui ne va da **oricare** dintre aceste două valori.

1.1. Ce este un pin?

Un pin este o bucată de metal ce permite crearea unei legături între o componentă electronică și alta. În cazul circuitelor integrate pinii pot ieși din capsula integratului sau pot face parte dintr-una din fețele sale, caz în care ne referim la ei ca *pad-uri*.





2. GPIO si ATTiny20

În datasheetul ATTiny20 [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf], la pagina 2, sunt descrise configurațiile pinilor. Observați ca exista mai multe configurații, deoarece acest microcontroller vine în diferite capsule ce diferă ca mărime, cost și metodă de lipire.

Să ne concentrăm pe prima varianta, descrisă în capitolul 1.1, SOIC (Small Outline Integrated Circuit) & TSSOP (Thin-Shrink Small Outline Package). Primul lucru pe care îl observăm este că această capsulă are 14 pini. Dintre aceștia, doi sunt necesari pentru alimentarea microcontroller-ului (VCC - pinul 1, si GND - pinul 14). Evident, acești doi pini **nu** pot fi folosiți ca GPIO. Însă restul, da!

Al doilea lucru pe care îl observăm este că pinii, pentru a fi mai ușor de folosit, au câte o denumire. Această denumire ne indică și ce funcții poate avea acel pin. În cazul pinilor 1 și 14 este simplu: VCC - alimentare de curent continuu, și GND - masă (ground). Denumirea celorlalți pini este de forma PXn, unde X este în {A, B} și n este în {0, 1, 2, 3, 4, 5, 6, 7} (spre exemplu: PB0, PA5, PB4, PA7). Aceasta este denumirea principală a acestor pini. Între paranteze găsim și denumiri auxiliare, care indică funcții alternative ale pinilor (spre exemplu: pinul PA0 mai este numit și ADC0, ceea ce înseamnă că el poate servi ca și intrare pentru convertorul analog-digital).

În microcontrollerele Atmel, cum este și ATTiny20, pinii ce suportă GPIO sunt grupați în porturi. Un port are întotdeauna 8 pini logici (ce pot fi folosiți într-un program), dar poate avea mai puțini pini fizici (ce pot fi folosiți pentru a realiza legături fizice). Aceste porturi sunt denumite în ordine alfabetică: A, B, etc. De aici reies și

numele pinilor: PA0 înseamnă că pinul face parte din portul A, și este pinul cu numărul 0 în acel port. Observați că portul B nu are decât 4 pini fizici (PB0, PB1, PB2 si PB3). Totuși nu este o eroare să atribuim într-un program o valoare pinului PB4, doar că această atribuire nu va avea niciun efect în exteriorul controller-ului.

2.1. Registrele de control al perifericelor și ale porturilor I/O

În cadrul microprocesoarelor ca ATTiny20, avem de-a face cu două tipuri de registre de control:

- Registre de control a modulelor periferice (e.g. timer, convertor analog-digital, controller USB, placă de rețea, placă video, tastatura, etc.)
- Registre de control a pinilor de intrare/ieșire (pinii GPIO - ținta noastră în acest laborator)

Pentru modulele periferice, registrele de control sunt folosite pentru a efectua operații precum: pornește/oprește modulul, schimbă anumite setări (e.g. BAUD rate pentru portul serial), etc. Pe lângă registre, pentru a interacționa cu aceste periferice, mai folosim registre de stare (pentru a afla care este starea perifericului), și registre de date (pentru a transmite/recepționa date către/de la el).

Dacă un pin are funcția de GPIO, atunci vom putea să îi controlăm direcția și valoarea în mod programatic. Vom realiza acestea prin scrieri/citiri în/din registre speciale, ce controlează exact acest comportament.

Registrele care controlează comportamentul unui port sunt:

- DDRx: Data Direction Register (i.e.: DDRA și DDRB),
- PORTx: Data Register (i.e.: PORTA și PORTB),
- PINx: Data Input Register (i.e.: PINA și PINB).

Fiecare dintre aceste registre are 8 biți, câte unul pentru fiecare pin logic al portului. De exemplu, DDRA (pentru portul A) are biții DDA0, DDA1, …, DDA7, PORTA are biții PORTA0, …, PORTA7 și PINA are biții PINA0, …, PINA7. În continuare ne vom limita doar la portul A al microncontrollerului.

Schimbând un bit din 1 în 0 sau invers putem schimba comportamentul pinului. Astfel, registrele menționate anterior au următoarea funcționalitate:

- DDRAn: un bit n setat pe 0 indică faptul ca va fi folosit pentru '**citire**', iar 1 indică '**scriere**'
- PORTAn: dacă bitul n este 0 atunci pinul respectiv va lua valoarea LOW (adică GND, care este în general 0V), iar dacă bitul este 1 atunci pinul va lua valoarea HIGH (adică VCC, care poate fi 5V, 3.3V, etc.). Vedeți tabelul 10-1 (pag. 44) din datasheet [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf] pentru detalii despre ce se întamplă când DDAn e setat ca port de intrare.
- PINAn: Dacă prin PORTAn stabilim cum vrem sa fie pin-ul, PINAn ne indică starea lui reală.

2.2. Spațiul de adrese

I/O SPACE	0x0000 ... 0x003F
SRAM DATA MEMORY	0x0040 ... 0x00BF
(reserved)	0x00C0 ... 0x3EFF
NVM LOCK BITS	0x3F00 ... 0x3F01
(reserved)	0x3F02 ... 0x3F3F
CONFIGURATION BITS	0x3F40 ... 0x3F41
(reserved)	0x3F42 ... 0x3F7F
CALIBRATION BITS	0x3F80 ... 0x3F81
(reserved)	0x3F82 ... 0x3FBF
DEVICE ID BITS	0x3FC0 ... 0x3FC3
(reserved)	0x3FC4 ... 0x3FFF
FLASH PROGRAM MEMORY	0x4000 ... 0x47FF
(reserved)	0x4800 ... 0xFFFF

Figura precedentă descrie organizarea spațiului de adresă pentru ATTiny20. Observați că memoria RAM nu ocupă decât o mică parte din acest spațiu și aceasta începe de la adresa 0x40. În rest avem diferite zone ce servesc altor scopuri. Ce se întâmplă de fapt este că toate perifericele, inclusiv memoria RAM si ROM (FLASH) sunt legate la aceeași magistrală de adrese, pe 16 biți (noi vom considera că are doar 8 biți). Fiecare dispozitiv este responsabil să răspundă doar la adresele ce îi aparțin. Deci, pentru orice adresă în intervalul [0x40:0xBF] memoria RAM este responsabilă de tratarea citirii/scrierii, iar celelalte periferice trebuie să elibereze magistrala de date. La fel, pentru adrese în intervalul [0x3FC0:0x3FC3] putem citi dintr-o mică memorie non-volatilă unde este reținut Device IDul.

De la 0x00 la 0x3F gasim *I/O Space*. Aici se află toate registrele de lucru cu periferice și, desigur, PORTA. În datasheet [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf], la capitolul 22. Register Summary (pagina 203) putem vedea adresele tuturor registrelor I/O din controller (care nu sunt de uz general). Registrul PORTA are adresa 0x02 deci pentru a scrie/citi date în/din el trebuie să executăm o instrucțiune STS/LDS cu adresa de scriere/citire 0x02.

```
LDS R16, 0x40    ; se încarcă date din RAM de la adresa 0x40 în registrul R16
STS 0x02, R16    ; se stochează datele din registrul R16 la adresa 0x02, adică în registrul PORTA
```

Pentru a lucra cu registrele din I/O Space putem folosi două instrucțiuni speciale: IN și OUT. IN este echivalent cu LDS, iar OUT este echivalent cu STS. Diferența este că, în vreme ce LDS/STS nu pot lucra decât cu registrele R16:R31, IN/OUT pot lucra cu toate cele 32 de registre. De asemenea, pe un core AVR adevărat, LDS/STS se execută în 2 cicli de ceas, iar IN/OUT într-unul singur. Astfel, codul precedent poate fi rescris:

```
LDS R16, 0x40
OUT 0x02, R16
```

2.2.1. Alte instrucțiuni ce operează cu I/O space

Pe lângă instrucțiunile dedicate de load și store (denumite `in` și `out`), mai sunt și alte instrucțiuni ce accesează această zonă de memorie. Noi vom lucra cu instrucțiunile `sbi` și `cbi` (set/clear bit în I/O register), a căror funcționalitate o vom vedea în cele ce urmează. Să presupunem că vrem sa facem toggle la pinul cu indicele 3 de pe portul A:

```
while (1) {
    PORTA |=  (1 << 3);
    PORTA &= ~(1 << 3);
}
```

Am putea traduce secvența de mai sus în următorul cod assembly:

```
0x00:    ldi     R17, 0b00001000
0x01:    ldi     R18, 0b11110111
         ; PORTA |= (1 << 3)
```

```
0x02:    in      R16, PORTA
0x03:    or      R16, R17
0x04:    out     PORTA, R16
         ; PORTA &= ~(1 << 3)
0x05:    in      R16, PORTA
0x06:    and     R16, R18
0x07:    out     PORTA, R16
         # Jump back to 0x00
         # PC = PC + k + 1 => 0x00 = 0x08 + k + 1 => k = -0x09
0x08:    rjmp    -9
```

Așa cum se poate vedea, comportamentul de mai sus este unul de tip *read-modify-write*: pentru a modifica un singur bit dintr-un registru, el trebuie citit și scris înapoi prin instrucțiuni `in` și `out` separate. Pentru a economisi cicli de ceas, putem folosi `sbi` (set bit in I/O register) și `cbi` (clear bit in I/O register). Să vedem cum ar arăta programul de mai sus utilizându-le:

```
         ; PORTA |= (1 << 3)
0x00:    sbi     PORTA, 3
         ; PORTA &= ~(1 << 3)
0x01:    cbi     PORTA, 3
         ; jump back to 0x00
0x02:    rjmp    -3
```

3. Hinturi pentru laborator

3.1. Signals

Sunt o serie de semnale de control (mănunchiul de fire denumit `signals`). Aceste semnale sunt de tipul one-hot [http://en.wikipedia.org/wiki/One-hot] și cuprind:

- `CONTROL_MEM_READ`, `CONTROL_MEM_WRITE`: semnale către `bus_interface_unit` care se pun pe 1 în momentul decodificării unui load/store, în funcție de starea curentă (în acest caz, în `STATE_MEM` și, eventual, `STATE_WB` în cazul în care este o citire, care durează 2 cicli de ceas)
- `CONTROL_REG_RD_READ`, `CONTROL_REG_RD_WRITE`, `CONTROL_REG_RR_READ`: semnale către `register_file_interface_unit` care îi semnalează, pe stările respective (citire pe `STATE_ID` și `STATE_EX`, respectiv scriere pe `STATE_WB`), faptul că ar trebui să acceseze register file-ul.
- `CONTROL_IO_READ`, `CONTROL_IO_WRITE`: semnale de asemenea către `bus_interface_unit` care sunt active în momentul în care a fost decodificată o instrucțiune de acces explicit la I/O (cum ar fi `in/out`, `sbi/cbi`, `push/pop` etc, dar în mod notabil **nu** `lds/sts/ldd/std`). În cazul unei instrucțiuni load/store, este datoria lui `bus_interface_unit` să decidă dacă destinația comunicațiilor sale trebuie să fie memoria RAM de date sau memoria I/O.

3.2. Instrucțiunea in

Să exemplificăm comportamentul pe stări al instrucțiunii `in Rd, A`:

- `STATE_IF`: procesorul trimite `program_counter` pe linia de adrese a ROM-ului, așteaptă răspuns
- `STATE_ID`: primește răspuns de la ROM pe linia `instruction`, îl bufferează în registrul `instr_buffer` pe care îl forwardează către modulul `decode_unit`. Aceasta ar trebui sa deducă următoarele:
  - `opcode_type` va fi `TYPE_IN`
  - `opcode_rd` va fi decodificat din instrucțiune
  - `opcode_imd` va fi decodificat din instrucțiune să ia valoarea A
  - `opcode_group` va fi `GROUP_IO_READ`
  - `signals` va avea următorii biți activați: `CONTROL_IO_READ`, `CONTROL_REG_RD_WRITE`
- Tot în acest ciclu, `bus_interface_unit`, cu semnalul `CONTROL_IO_READ` primit de la unitatea de decodificare de instrucțiuni, sesizează că trebuie să citească date de la memoria I/O, așa că setează semnalele de control `io_cs`, `io_we` și `io_oe`, iar pe linia `addr` plasează valoarea `opcode_imd`, așa cum este ea primită de la unitatea de decodificare (este fix A-ul din instrucțiune).

- `STATE_EX`: operațiunea de citire din I/O memory se finalizează, iar pe linia `data`, exportata și către `control_unit`, se va afla valoarea din I/O memory, de la adresa A.
- `STATE_MEM`: nu se întâmplă nimic notabil în această stare
- `STATE_WB`: `register_file_interface_unit` depistează că instrucțiunea curentă necesită writeback în Rd, și poziționează semnalele de `cs`, `we` și `oe` pentru o scriere, la adresa `opcode_rd` conținută în instrucțiune, a datelor `data` citite din memorie, similar cu instrucțiunea `load`.

Așa cum am văzut mai sus, în timp ce instrucțiunile load/store accesează memoria pe ciclul `STATE_MEM` (cu prelungire în `STATE_WB` pentru o citire), instrucțiunile de acces la I/O o accesează în același mod ca și pe registre: în `STATE_ID` (cu prelungire în `STATE_EX`) pentru o citire, respectiv în `STATE_WB` pentru o scriere. Doar astfel ne putem permite un **read-modify-write într-o singură instrucțiune** pe registre din I/O space, precum modificarea SP-ului la operațiile cu stiva, sau a SREG-ului la operațiile aritmetice.

TL;DR

Folosind GPIO (General-Purpose Input/Output) putem controla un pin al unui circuit integrat. De exemplu, putem aprinde un led folosind instrucțiuni AVR sau putem citi starea unui pin - LOW sau HIGH. Pinii ce suportă GPIO sunt grupați în porturi (grupuri de 8 pini logici numite A sau B) controlate de registre speciale (pe câte 8 biți):

1. DDRx: Data Direction Register
    - un bit setat pe 0 indică faptul ca va fi folosit pentru 'citire'
    - un bit setat pe 1 indică 'scriere'
2. PORTx: Data Register: (Stabilesc cum vreau să fie un pin)
    a. Dacă DDRx e setat ca ieșire ('scriere')
        - dacă un bit n este 0 atunci pinul respectiv va fi legat la LOW (GND)
        - dacă un bit n este 1 atunci pinul respectiv va fi legat la HIGH (VCC)
    b. Dacă DDRx e setat ca intrare ('citire')
3. PINx: Data Input Register (Indică starea reală a unui pin)
    - daca un bit este 0 atunci pinul respectiv are valoarea LOW
    - daca un bit este 1 atunci pinul are valoare HIGH

Obs: x poate fi A sau B

În memoria microcontrollerului de la 0x00 la 0x3F gasim I/O Space. Aici se află toate registrele de lucru cu periferice și, desigur, DDRx, PORTx și PINx. De exemplu registrul PORTA are adresa 0x02.

Pentru a scrie citi date din I/O Space putem folosi instrucțiunile LDS/STS, însă recomandat este să folosim instrucțiunile speciale pentru lucru de memoria I/O Space ce se execută mai rapid:

- `OUT A, Rr`: Scrie (Store) în I/O Space la adresa A valoarea din registrul Rr
- `IN Rd, A` : Citește (Load) din I/O Space de la adresa A și pune în registrul Rd

Exerciții

Ce trebuie modificat?
Pentru rezolvarea exercițiilor 1 și 2 trebuie să modificați următoarele fișiere:

- `bus_interface_unit.v`
- `decode_unit.v`
- `signal_generation_unit.v`
- `control_unit.v`

Checker
Scheletul conține un checker minimal doar pentru exercițiile 1 și 2. Simulați fișierul `unitTestCpu` pentru a-l utiliza.

Punctajul maxim pentru exercițiile 1 și 2 se obține doar dacă **nu** apar teste picate (FAILED) în consola ISim **și** în registrul R17 se află valoarea 3 la sfârșitul programului din `rom.v`.

Task 01(3p) Implementați logica de selecție a modulului de RAM și a celui de GPIO. Urmăriți comentariile marcate cu TODO1 din modulul `bus_interface_unit.v` .

Task 02(3p) Adăugați logica necesară decodificării și executării instrucțiunilor IN și OUT. Urmăriți comentariile marcate cu TODO2 din `decode_unit.v`, `signal_generation_unit.v` și `control_unit.v` . Citiți secțiunea 3.2 a laboratorului pentru a înțelege cum funcționează instrucțiunea `in`.

Task 03(2p) Scrieți un program care încarcă valoarea 42 în registrul R19, apoi o scrie în registrul PORTA, apoi încarcă valoarea din PORTA în R20. Folosiți tool-ul avrasm pentru a-l scrie în fișierul `rom.v`. Simulați fișierul `unitTestCpu` și verificați valoarea registrului R20 la finalul programului.

Din perspectiva programului, PORTA este doar o adresă de memorie din spațiul I/O. Identificați ce adresă corespunde lui PORTA folosind secțiunea "Register summary" din datasheet [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf].

java -jar avrasm.jar input.txt output.txt

Task 04(2p) Scrieți un program care pune pinii portului A in următoarea stare: PA0 - aprins, PA1 - stins, PA2 - aprins, PA3 - stins, PA4 - stins, PA5 - aprins, PA6 - aprins, PA7 - stins. Simulați funcționarea programului.

Task 05(BONUS - 2p) Scrieți un program care să aprindă led-urile în următoarea secvență, apoi simulați funcționarea programului:

```
t0   *-------
t1   -*------
t2   --*-----
t3   ---*----
t4   ----*---
t5   -----*--
t6   ------*-
t7   -------*
```

Pentru a vedea efectul programului va trebui să încetiniți semnalul de ceas. Puteți realiza asta modificând modulul cpu astfel încât semnalul de ceas care intră, `clk`, să comute un semnal **`_clk`** o dată la fiecare 1000000 de fronturi pozitive. Toate modulele instanțiate în cpu vor trebui să primească semnalul **`_clk`** în loc de semnalul `clk`.

Resurse

7. Resurse

- Datasheet ATTiny20 [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf]
- Setul de Instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]
- Datasheet Digilent Nexys [https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys3/documentation/Nexys3_rm.pdf]
- avrasm - tool java pentru a genera cod mașină AVR.
- Scheletul laboratorului

cn2/laboratoare/06.txt · Last modified: 2019/10/01 13:00 by tudor.visan

Laboratorul 07 - GPIO 2: Intrări

În acest laborator vom îmbunătăți implementarea noastră a modulului de GPIO adăugând posibilitatea pinilor de a funcționa și ca intrări.

Registrele de control ale perifericelor și ale porturilor I/O

În laboratorul trecut am învățat că, dacă un pin are funcția de GPIO, atunci vom putea să îi controlăm direcția și valoarea în mod programatic prin scrieri/ citiri în/din anumite registre.

Registrele care controlează comportamentul unui port sunt:

- DDRx: Port x Data Direction Register (ie: DDRA, DDRB)
- PORTx: Port x Data Output Register (ie: PORTA, PORTB)
- PINx: Port x Data Input Register (ie: PINA, PINB)

Aceste registre sunt toate pe 8 biți iar fiecare bit controlează comportamentul unui pin al portului respectiv.

DDRx

Registrul DDRx controleaza direcția pinilor din portul x:

- DDRxn == 0: dacă bitul n este 0 atunci pinul respectiv va funcționa ca și intrare (microcontroller-ul nu va impune nicio tensiune pe linie, ci îl va lăsa în stare de înaltă impedanță, Z)
- DDRxn == 1: dacă bitul n este 1 atunci pinul va funcționa ca și ieșire (microcontroller-ul va putea impune o tensiune electrică pe acel pin)

PORTx

Registrul PORTx controlează valoarea pinilor din portul x care au fost configurați ca ieșiri:

- PORTxn == 0: dacă bitul n este 0 atunci pinul respectiv va lua valoarea LOW
- PORTxn == 1: dacă bitul n este 1 atunci pinul respectiv va lua valoarea HIGH

PINx

Din registrul PINx putem citi valoarea valoarea pinilor din portul x care au fost configurați ca intrări:

- PINxn == 0: dacă bitul n este 0 atunci pinul respectiv are valoarea LOW
- PINxn == 1: dacă bitul n este 1 atunci pinul respectiv are valoare HIGH

La pagina 203 din Datasheet ATTiny20 [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf] gasim adresele acestor registrii.

Instrucțiunile SBI și CBI

Până acum am învațat că dacă schimbăm un bit din DDRx sau PORTx putem să influențăm comportamentul unui pin. Însă instrucțiunile `in` și `out` nu pot scrie decât valori pe 8 biți. Spre exemplu, dacă dorim să schimbăm doar bitul 5 din DDRA în 1 atunci pașii pe care trebuie sa îi urmăm sunt:

```
ldi     R17, 0b00100000 ; încărcăm în R17 o mască ce are 1 pe bitul 5
in      R16, DDRA       ; încărcăm în R16 valoarea din DDRA
or      R16, R17        ; în urma operației sau între R16 și R17 în R16 bitul 5 va fi pus pe 1
out     DDRA, R16       ; stocăm valoarea din R16 înapoi în DDRA
```

Pentru a pune **bitul n pe 1** trebuie să folosim o mască de biți ce are **1 pe poziția n și 0 în rest** (eg: pentru a pune bitul 5 pe 1 folosim masca 00100000), apoi să folosim operațiu **sau pe biți**.

Pentru a pune **bitul n pe 0** trebuie să folosim o mască de biți ce are **0 pe poziția n și 1 în rest** (eg: pentru a pune bitul 5 pe 0 folosim masca 11011111), apoi să folosim operațiu **și pe biți**.

Fiindcă acțiunea de a schimba un bit în 1 (set) sau în 0 (clear) este una întâlnită des în programele ce rulează pe arhitectura AVR aceasta include două instrucțiuni ce permit schimbarea unui bit dintr-un registru ce se află în primele 32 de adrese din I/O Space în 0 sau în 1: `cbi` (Clear Bit in Register) și `sbi` (Set Bit in Register). Ele primesc ca și argumente adresa registrului (în intervalul [0:31]) și numărul bitului (în intervalul [0:7]). Dacă rescriem codul folosind aceste înstrucțiuni el devine:

```
sbi DDRA, 5 ; mai scurt, nu?
```

Instrucțiunile `cbi` și `sbi` primesc ca și argument **numărul** bitului pe care dorim să-l schimbăm, nu o mască de biți.

TL;DR

- Registre de control pentru porturile I/O
  - DDRx: fiecare bit controlează dacă pinul respectiv este intrare (0) sau ieșire (1)
  - PORTx: fiecare bit controlează daca pinul respectiv ia valoarea HIGH sau LOW, daca acel pin este configurat ca ieșire (vezi DDRx)
  - PINx: fiecare bit reprezintă valoarea pe care o are pinul respectiv, daca acel pin este configurat ca intrare (vezi DDRx)
- Modificarea unui bit dintr-un registru I/O
  - Folosind instrucțiunile IN și OUT cu o mască de biți:

```
LDI R16 0x00  ; R16 = 00000000
OUT 0x01, R16 ; DDRA = R16 = 00000000
...
LDI R16, 0x01 ; R16 = 00000001
IN  R17, 0x01 ; R17 = DDRA = 00000000
OR  R17, R16  ; R17 = R17 | R16 = 00000001
OUT 0x01, R16 ; DDRA = R17 = 00000001
...
LDI R16 0xFF  ; R16 = 11111111
OUT 0x02, R16 ; PORTA = R16 = 11111111
...
LDI R16, 0xFE ; R16 = 11111110
IN  R17, 0x02 ; R17 = PORTA = 11111111
AND  R17, R16 ; R17 = R17 & R16 = 11111110
OUT 0x02, R16 ; PORTA = R17 = 11111110
```

  - Folosind instrucțiunile SBI și CBI

```
LDI R16 0x00  ; R16 = 00000000
OUT 0x01, R16 ; DDRA = R16 = 00000000
...
SBI 0x01, 0   ; DDRA = 00000001
...
LDI R16 0xFF  ; R16 = 11111111
OUT 0x02, R16 ; PORTA = R16 = 11111111
...
CBI 0x02, 0   ; PORTA = R17 = 11111110
```

Exerciții

Task 01 (4p) Extindeți modulul gpio astfel încât să suporte și intrări și ieșiri. Urmăriți comentariile marcate cu `TODO 1` din `gpio.v`.

Task 02 (2p) Decodificați instrucțiunile `sbi` și `cbi` și completați logica necesară execuției lor. Urmăriți comentariile marcate cu `TODO 2` din `signal_generation_unit.v`, `decode_unit.v` și `control_unit.v`.

Task 03 (1p) Scrieți un program care setează portul A ca ieșire, portul B ca intrare, apoi, într-o buclă, citește valoarea de pe portul B și o scrie pe portul A.

Task 04 (3p) Scrieți un program care să dispuna următoarea secvență folosind instructiunle SBI si CBI. Vom verifica din simulator daca este corect.

```
t0   *------*
t1   -*----*-
t2   --*--*--
t3   ---**---
t4   ---**---
t5   --*--*--
t6   -*----*-
t7   *------*
```

Task 05 (Bonus 3p) Modificați programul de la punctul 4 astfel incat afisarea secventei să fie controlată prin intermediul portului B. Fiecare PINBx va controla afisarea uneia dintre linii. Exemple:

- Daca PINB0 si PINB1 sunt pe 1 si restul pinilor pe 0, atunci programul va cicla doar printre linile 0 si 1
- Daca PINB4, PINB5 si PINB7 sunt pe 1 si restul pinilor pe 0, atunci programul va cicla doar printre liniile 4, 5 si 7

Vedeti TODO-ul din unitTest.v pentru un mod de a modifica valoarea pinilor portului B.

Resurse

- Scheletul laboratorului
- Datasheet ATTiny20 [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf]
- Setul de Instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]
- Datasheet Digilent Nexys [https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys3/documentation/Nexys3_rm.pdf]
- avrasm - tool java pentru a genera cod mașină AVR.

cn2/laboratoare/07.txt · Last modified: 2019/11/18 07:10 by tudor.visan

Laboratorul 08 - Timer

În acest laborator vom învăța ce este și cum funcționează un timer/counter și vom adăuga un astfel de modul implementării noastre a microcontroller-ului ATtiny20.

Timer/Counter

În esență, un timer este un simplu numărător implementat în hardware:

```
reg [7:0] counter;

always @(posedge clk)
    if (reset)
        counter <= 0;
    else
        counter <= counter + 1;
```

Datorită faptului că un timer e implementat în hardware, incrementarea lui este complet transparentă pentru programator, permițându-i acestuia să execute alte task-uri.

Un bun exemplu de utilizare a unui timer este păstrarea evidenței timpului într-o aplicație. Putem configura un "stopwatch" care să declanșeze un eveniment ce va trebui tratat de software. Astfel, putem deduce că este foarte util ca un timer să poată genera o *întrerupere*, adică o funcție care se execută atunci când un anumit eveniment hardware are loc.

În laboratorul de astăzi vom aprofunda conceptul de timere, fără a ne atinge de întreruperi, însă.

În principiu, orice timer ar trebui să aibă următoarele caracteristici:

- O **sursă de ceas** în baza căreia numără (în general, la fiecare tranziție pozitiva a acesteia)
    - Opțional, această sursă de ceas poate fi divizată în prealabil (aplicat un prescaler) pentru a număra la intervale mai rare de timp
- Un registru **contor** în care este stocată valoarea curentă a numărătorii
    - În funcție de lățimea (numărul de biți ai) acestui registru, timer-ul poate număra până la o valoare mai mică sau mai mare
    - În general valoarea acestui registru este incrementată, adica timerele în general numără crescător
    - Stabilim două valori de referință:
        - **BOTTOM** (cea mai mică valoare de la/până la care poate număra timer-ul)
        - **MAX** (cea mai mare valoare până la/de la care poate număra timer-ul)
        - BOTTOM și MAX sunt două valori intrinseci implementării hardware a timer-ului, adică noi nu le putem schimba din software
    - Opțional timer-ul poate implementa și o valoare configurabilă **TOP**, care poate lua orice valoare între BOTTOM și MAX
        - Atunci când contorul întâlnește această valoare, se pot întâmpla mai multe lucruri, depinzând de configurare:
            - Contorul se resetează la BOTTOM
            - Timer-ul începe sa numere descrescător
            - Generează un eveniment și se oprește din numărat (se mai numește și *one-shot timer*)

Denumirile BOTTOM, MAX și TOP sunt în concordanță cu terminologia Microchip, în datasheet-ul ATtiny20 [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf], pagina 60.

PWM

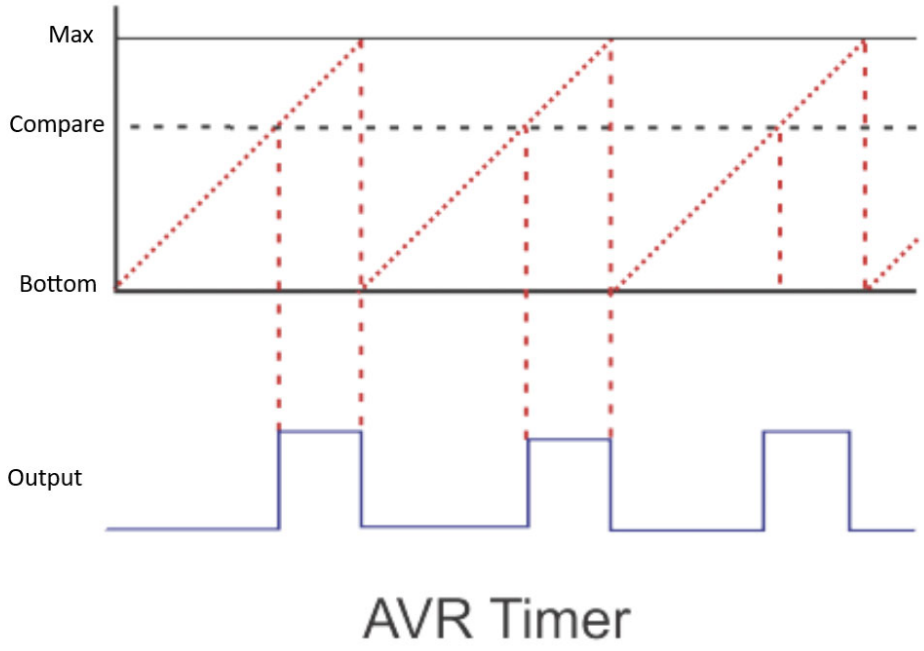
Pulse-width modulation [http://en.wikipedia.org/wiki/Pulse-width_modulation] este o tehnică de generare a unui semnal analogic pornind de la unul digital (adică de a înlocui un DAC [http://en.wikipedia.org/wiki/Digital-to-analog_converter]). Fiindcă logica digitală nu poate genera la ieșire decât tensiuni de Vcc sau de GND (ideal vorbind), PWM își propune să sintetizeze tensiuni cuprinse între aceste valori prin comutarea foarte rapidă între cele două.

Spre exemplu, presupunând că avem o tensiune Vcc de 5V și GND de 0V, dacă am comuta *foarte* repede între 0V și 5V ieșirea, menținând procentele egale de timp între cele două nivele logice (adică la fel de mult pe 1 ca și pe 0), atunci am putea spune că, **în medie**, tensiunea noastră de ieșire este de 2.5V. Asemănător, dacă am păstra aceeași frecvență de comutație, însă am tine ieșirea 75% din timp pe 1 și 25% din timp pe 0, atunci am putea spune că avem o ieșire cu o tensiune medie de 0.75 * 5V + 0.25 * 0V = 3.75V.

Putem spune, așadar, că PWM se traduce prin **modulare în factor de umplere** (factor de umplere înseamnă raportul dintre perioada cât stă semnalul pe 1 și perioada totală). Prin urmare tensiunea rezultată dintr-un semnal PWM este direct proporționala cu factorul de umplere. Astfel putem, plecând de la semnale digitale, să generăm semnale analogice.

Ce legătura are PWM cu timerele?

Un timer poate continua să numere după ce generează un eveniment. Grafic, este această imagine (în desen, valoarea la care se generează evenimentul este denumită “Compare”):



Asociat numărătorului se afla un output:

```
    assign out = (counter >= compare);
```

Putem observa că factorul de umplere al acestui semnal este direct proporțional cu valoarea lui `Compare`. Ca atare, putem genera un semnal PWM.

Timere și ATtiny20

Informațiile particulare despre timerele mictrocontrollerului ATtiny20 le putem găsi în datasheet [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf],

capitolele 11 și 12. Noi vom implementa doar `Timer/Counter0`.

Timerele ATtiny20 au 3 registre foarte importante:

TCNT0 (Timer/Counter Register)

Este registrul cu rolul de **contor**.

Incrementarea acestui registru se face automat de către hardware!

OCR0A/OCR0B (Output Compare Register A/B)

Aceste două registre conțin fiecare câte o valoare de **comparație** între BOTTOM și MAX. Atunci când `TCNT0 == OCR0A` sau `TCNT0 == OCR0B` se pot întâmpla mai multe lucruri, în funcție de configurare:

- Nimic
- Timer-ul se resetează (TOP este configurat să ia valoarea OCR0A/OCR0B)
- Timer-ul generează un eveniment și continuă să numere
- Timer-ul generează un eveniment și se resetează (TOP este configurat să ia valoarea OCR0A/OCR0B)

Evenimentul generat atunci când `TCNT0 == OCR0A` sau `TCNT0 == OCR0B` poate fi:

- O întrerupere
- Schimbarea stării pinului OC0A/OC0B

Registrele OCR0A/OCR0B pot fi folosite și pentru a genera un semnal PWM dacă timer-ul este configurat să genereze un eveniment de comparație și să continue să numere, și acel eveniment este schimbarea stării pinului OC0A/OC0B. În acest caz putem varia factorul de umplere modificând valoarea registrului OCR0A/OCR0B (în imaginea din capitolul precedent, valoarea “Compare”), iar semnalul PWM va fi dispoibil pe pinul OC0A/OC0B (în imaginea din capitolul precedent, valoarea “Output”).

Fiindcă sunt două la număr înseamnă că putem genera două semnale PWM folosind un singur timer.

TCCR0A/TCCR0B (Timer/Counter Control Register)

Acestea sunt registrele de **control** pentru modul de funcționare al timer-ului. În realitate, cele două alcătuiesc un singur registru de control, TCCR0, pe 16 biți. Registrul respectiv conține multe informații de configurare codificate În biții săi. O lista completă a acestora o puteți găsi în datasheet, capitolul 11.9.1 și 11.9.2 paginile 69-73, însă un rezumat scurt vom face aici.

Bit	7	6	5	4	3	2	1	0	
0x19	COM0A1	COM0A0	COM0B1	COM0B0	–	–	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Remarcăm faptul că registrul TCCR0A conține 3 sub-valori:

- Biții 7:6 conțin valoarea **COM0A**, pe 2 biți, ce controlează funcția pinului de I/O OC0A
- Biții 5:4 conțin valoarea **COM0B**, pe 2 biți, ce controlează funcția pinului de I/O OC0B
- Biții 1:0 conțin (o parte din) valoarea **WGM0** (Waveform Generation Mode), un număr pe 3 biți (al cărui ultim bit se găsește În registrul TCCR0B, pentru că motive) care setează modul de funcționare al timer-ului: normal, fast PWM, phase-correct PWM, CTC, etc. Vom vedea imediat ce face fiecare mod de funcționare

Bit	7	6	5	4	3	2	1	0	
0x18	FOC0A	FOC0B	–	–	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Remarcăm faptul că registrul TCCR0B conține 2 sub-valori:

- Bitul 3 conține valoarea WGM02 din **WGM0**, nu are semnificatie de sine stătătoare
- Biții 2:0 conțin valoarea **CS0** (Clock Source), pe 3 biți, care dictează valoarea prescalerului (dacă este cazul), respectiv dacă timer-ul este pornit sau oprit

Literele A și B nu au nici o legătură cu canalele A și B de la registrele OCR0A/OCR0B.

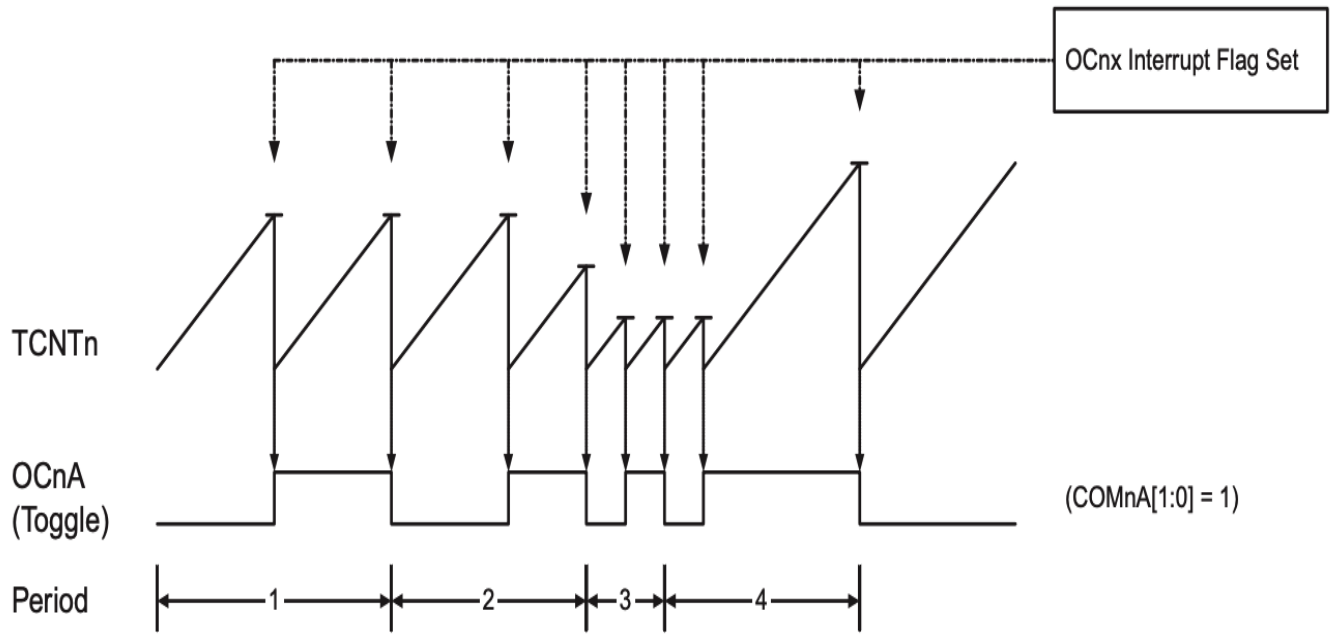
Moduri de funcționare

După cum spuneam, timer-ul poate avea comportament diferit în funcție de configurare. Modurile de funcționare controlează direcția de numărare (crescătoare, descrescătoare sau ambele), valoarea TOP și parțial comportamentul pinilor OC0A/OC0B.

Modul de funcționare poate fi setat modificând biții WGM0 din registrele TCCR0A și TCCR0B. Ele pot fi:

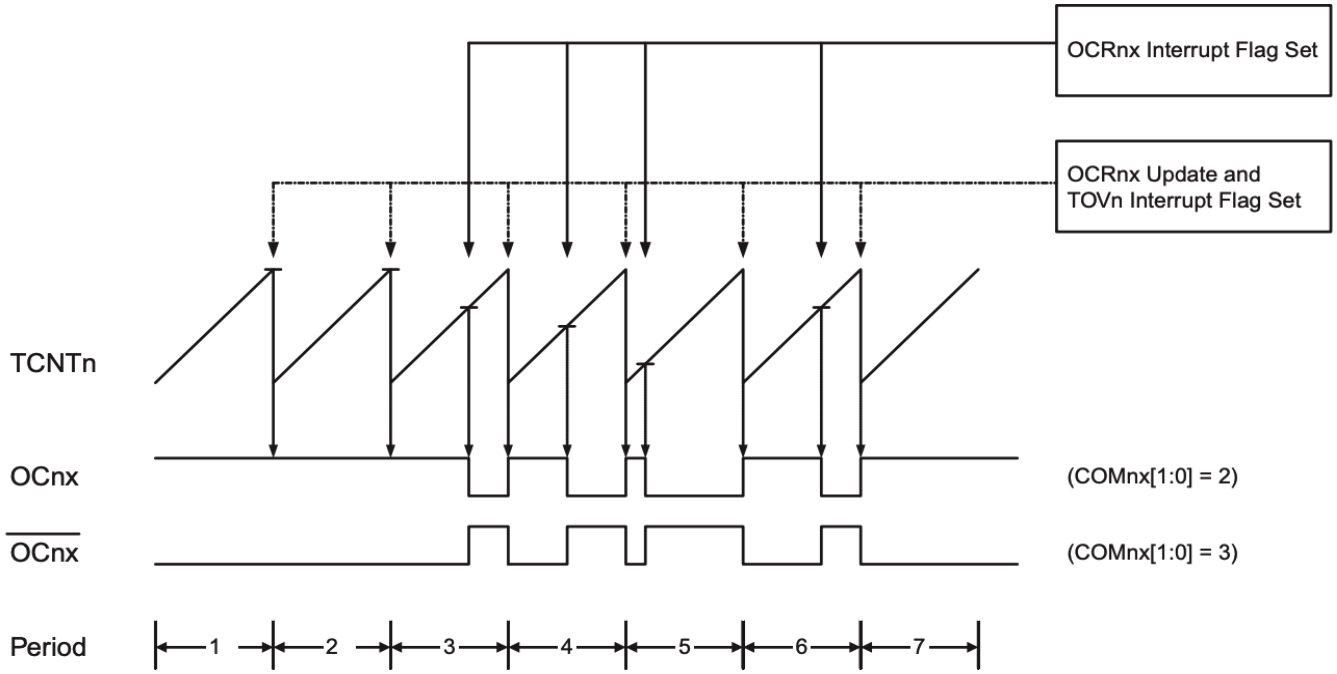
- **Normal** (`WGM0 == 0`)
  - Timer-ul numără de la 0 (deci de la `BOTTOM == 0`) crescător până la 255 (deci până la `TOP == MAX == 255`), apoi se resetează înapoi la 0
  - Starea pinilor OC0A/OC0B poate fi schimbată atunci când se atinge pragul de comparație (OCR0A/OCR0B)

- **CTC** (Clear Timer on Compare Match - `WGM0 == 2`)
  - Timer-ul numără de la 0 (deci de la `BOTTOM == 0`) crescător până la OCR0A (deci până la `TOP == OCR0A`), apoi se resetează înapoi la 0
  - Starea pinilor OC0A/OC0B poate fi schimbată atunci când se atinge pragul de comparație (OCR0A/OCR0B)



- **Fast PWM** (`WGM0 == 3 sau WGM0 == 7`)

- Timer-ul numără de la 0 (deci de la `BOTTOM == 0`) crescător până la 255, sau până la OCR0A (deci până la `TOP == MAX == 255`, dacă `WGM0 == 3`, sau `TOP == OCR0A`, dacă `WGM0 == 7`), apoi se resetează înapoi la 0
- Starea pinilor OC0A/OC0B poate fi schimbată atunci când se atinge pragul de comparație (OCR0A/OCR0B) și atunci când se atinge valoarea BOTTOM



Exemplu de configurare

Timer-ul 0 al unui microcontroller ATtiny20 se poate configura în modul normal astfel:

```
ldi     R16, 0b00000000    ; COM0A = 0 (normal port operation, OC0A disconnected)
                           ; COM0B = 0 (normal port operation, OC0B disconnected)
                           ; WGM0[1:0] = 0 (normal mode operation)
out     TCCR0A, R16

; Pornim timer-ul scriind o valoare diferită de 0 pe biții CS0
ldi     R16, 0b00000101    ; WGM0[2] = 0 (normal mode operation)
                           ; CS0 = 5 (clkT = clkIO/1024 from prescaler)
out     TCCR0B, R16
```

Din punctul în care am scris ceva nenul pe biții 2:0 din TCCR0B (biții CS0), timer-ul a pornit. Este important de realizat că timer-ul nu poate fi cu adevărat pornit sau oprit ci, pur și simplu, dacă CS0 este 0, atunci modulul nu primește semnal de ceas, deci contorul nu va fi incrementat. Așadar, din momentul ultimei instrucțiuni din exemplul de cod, timer-ul începe să își incrementeze contorul cu câte o unitate la fiecare 1024 de cicli de ceas.

TL;DR

- Un timer este un simplu numărător implementat în hardware
- Orice timer trebuie să aibă:
  - O sursă de ceas
  - Un registru contor
    - Trei valori de referință: BOTTOM, MAX și TOP
- PWM este o tehnică de generare a unui semnal analogic pornind de la unul digital
  - Semnalul analogic obtinut depinde de factorul de umplere al semnalului digital

- Timerele de pe ATtiny20 conțin 3 registre importante: TCNT0, OCR0A/OCR0B, TCCR0A, TCCR0B
  - TCNT0 este registrul cu contor
  - OCR0A/OCR0B definesc valori de comparație iar OCR0A poate fi folosit la TOP
  - TCCR0A și TCCR0B sunt registre de control pentru modul de funcționare
- Moduri de funcționare:
  - Normal
    - Timer-ul numără crescător de la 0 la 255, apoi se resetează la 0
  - CTC
    - Timer-ul numără crescător de la 0 la OCR0A, apoi se resetează la 0
  - Fast PWM
    - Timer-ul numără crescător de la 0 la 255 (sau OCR0A), apoi se resetează la 0
    - Starea pinilor OC0A/OC0B poate fi schimbată la comparației și BOTTOM

Exerciții

Task 1 (2p) Construiți logica de prescaling pentru ceasul timer-ului 0.

Task 2 (3p) Decodificați biții din registrele de control `TCCR0A` și `TCCR0B`. Implementați logica de selecție a modului de operare și a valorii maxime de numărare (TOP). Implementați logica de incrementare a contorului.

Task 3 (3p) Implementați logica ieșirilor `OC0A` și `OC0B`.

Task 4 (4p) Scrieți și simulați un program în avrasm [https://elf.cs.pub.ro/cn/wiki/_media/lab/cn2/avrasm.zip] care:

- Configurează timer-ul 0 in modul `fast PWM, TOP == 0xFF`
  - HINT: Ce biți controlează modul de operare al timer-ului? Ce valoare trebuie să aibă acei biți? În ce registre se află acei biți?
- Setează ceasul timer-ului 0 la `clkI/O (No prescaling)`
  - HINT: Ce biți controlează sursa de ceas a timer-ului? Ce valoare trebuie să aibă acei biți? În ce registru se află acei biți?
- Configurează timer-ul 0 să schimbe starea pinului OC0A astfel: `Clear OC0A on Compare Match, set OC0A at BOTTOM (non-inverting mode)`
  - HINT: Ce biți controlează pinul OC0A? Ce valoare trebuie să aibă acei biți? În ce registru se află acei biți?
- Setează valoarea lui OCR0A la 63

Puteți consulta din nou sectiunea 3 și datasheet-ul la capitolul 11 pentru mai multe detalii. Adresele registrelor I/O noi (`TCCR0A`, `TCCR0B`, `TCNT0`, `OCR0A`, `OCR0B`) au fost deja trecute În `defines.vh`, însa pot fi găsite și în datasheet [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf] la pagina 203 dacă este nevoie de ele.

Pentru a usura lucrul cu adresele din IO (pentru a evita repetarea adreselor in cod), avrasm poate lucra cu aliasuri. Putem sa ne definim:

```
TCCR0A equ 0x19
TCCR0B equ 0x18
TCNT0  equ 0x17
OCR0A  equ 0x16
OCR0B  equ 0x15
PORTA  equ 0x02
DDRA   equ 0x01
```

Apoi in fisierul de input pentru avrasm putem folosi DDRA, inloc de 0x01.

Exemplu de fisier de intrare pentru avrasm:

```
TCCR0A equ 0x19
TCCR0B equ 0x18
TCNT0  equ 0x17
OCR0A  equ 0x16
OCR0B  equ 0x15
PORTA  equ 0x02
DDRA   equ 0x01


ldi r16, 0b01111111
out DDRA, r16       ; DDRA = 0b01111111
```

Resurse

Schelet de cod

avrasm - tool java pentru a genera cod mașină AVR.

[0] Datasheet ATTiny20 [http://www.atmel.com/Images/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf]

[1] Setul de Instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]

[2] Datasheet Digilent Nexys 3 Spartan6 [https://www.xilinx.com/support/documentation/university/XUP%20Boards/XUPNexys3/documentation/Nexys3_rm.pdf]

More Info:

- Atmel AVR130: Setup and Use the AVR Timers [http://ww1.microchip.com/downloads/en/AppNotes/Atmel-2505-Setup-and-Use-of-AVR-Timers_ApplicationNote_AVR130.pdf]
- AVR and Arduino timer interrupts [http://www.engblaze.com/microcontroller-tutorial-avr-and-arduino-timer-interrupts/]
- Timer tutorial on AVRBeginners [http://www.avrbeginners.net/architecture/timers/timers.html]

cn2/laboratoare/08.txt · Last modified: 2019/11/25 16:50 by tudor.visan

Laboratorul 09 - Întreruperi

În acest laborator vom înțelege cum funcționează întreruperile și vom implementa un sistem minimal de întreruperi pentru procesorul nostru AVR. Vom demonstra funcționalitatea lor folosind o rutină de tratare a întreruperilor de timer.

Descriere generală

Întreruperile sunt evenimente **generate asincron** față de codul nostru care trebuie să capteze atenția procesorului pentru a fi rezolvate imediat. Altfel spus, în timpul execuției unui program, dacă este generată o întrerupere, procesorul va **întrerupe firul normal de execuție după terminarea instrucțiunii curente** și va trata întreruperea.

Întreruperile sunt folosite pentru a evita polling-ul și reprezintă fundamentul multitasking-ului preemptiv [https://en.wikipedia.org/wiki/Preemption_(computing)#Preemptive_multitasking] pe un sistem cu un singur procesor și un singur set de resurse hardware (unitate de execuție, registre, program counter, spațiu de adrese). În lipsa lor, singurul tip de execuție concurențială posibilă pe un astfel de sistem este multitasking-ul cooperativ [https://en.wikipedia.org/wiki/Cooperative_multitasking].

Tratarea întreruperilor

Pentru a determina dacă trebuie să trateze o întrerupere, procesorul verifică, după fiecare instrucțiune, linia de cereri de întreruperi (interrupt request sau **IRQ**). Dacă această linie este activă atunci, după terminarea execuției instrucțiunii curente, starea curentă a procesorului este salvată pe stivă și se transferă execuția către o rutină de tratare a întreruperii respective, după un mecanism foarte asemănător instrucțiunii `CALL`.

Tratarea întreruperilor se face de către funcții dedicate, numite rutine de tratare a întreruperilor (Interrupt Service Routine - **ISR**). Acestea sunt rutinele care sunt apelate la apariția unei întreruperi. Ceea ce caracterizează o rutină de tratare a întreruperilor față de o rutină normală este că ea se încheie întotdeauna cu instrucțiunea `RETI` (echivalentul `IRET` de pe x86).

Prin **vector de întrerupere** ne referim la adresa de memorie la care se afla începutul unei astfel de rutine. Rutinele de tratare pentru toate întreruperile pe care știe să le trateze un procesor sunt înscrise într-o tabelă a vectorilor de întrerupere [http://en.wikipedia.org/wiki/Interrupt_vector_table], o zonă dedicată a memoriei de instrucțiuni. În această zonă procesorul este programat să caute rutine de tratare, creând astfel asocierea dintre un request hardware și o rutină software de tratare a lui. Numărul unei întreruperi (**IRQNO**) reprezintă indexul în tabela vectorilor de întrerupere a rutinei sale de tratare.

Pentru a evita tratarea aceleiași întreruperi imediat dupa terminarea rutinei sale, rutina trebuie să informeze sursa că întreruperea a fost tratată. Acest proces se numește acknowledge - **ACK**. Este responsabilitatea programului să facă ACK unei întreruperi astfel încât sursa să își retragă cererea.

Să luăm un exemplu. La sfârșitul execuției unei instrucțiuni, procesorul verifică linia de cereri de întreruperi (IRQ) și vede că ea este activă. Asta înseamnă că există o întrerupere ce trebuie tratată. Mai departe, se uită la numărul întreruperii (IRQNO), pentru a determina cum să trateze întreruperea. Odată ce știe numărul, în tabela vectorilor de întrerupere la indexul respectiv găsește rutina de tratare a întreruperii respective, și o execută.

În sistemele cu procesor x86 convenția este ca întreruperea cu numărul 2 să fie asociată cu evenimente de la tastatură. Asta înseamnă că toate tastaturile, atunci când vor să trimită informații către sistem, vor genera o cerere de întrerupere (IRQ), și vor transmite numărul întreruperii (IRQNO) ca fiind 2. Programatorii, știind această convenție, vor înregistra în tabela vectorilor de întrerupere, la intrarea numărul 2, o rutină care va citi caracterele trimise de către tastatură și le va prelucra.

Mascarea întreruperilor

Nu tot timpul este de dorit ca un program să poată fi întrerupt. De aceea, întreruperile pot fi deactivate sau mascate. Chiar mai mult, în general procesoarele își încep execuția după reset cu întreruperile dezactivate.

A activa/dezactiva întreruperile pe un procesor înseamnă a permite/nu permite, la nivel global, tratarea lor. Activarea/Dezactivarea întreruperilor are loc doar la nivel global procesorului, deci fie toate întreruperile sunt active, fie niciuna nu este.

A demasca/masca întreruperi pe un procesor înseamnă a permite/nu permite, la nivel individual, tratarea unei întreruperi. Demascarea/Mascarea întreruperilor are loc doar la nivel individual, deci mascarea unei întreruperi nu afectează nicio altă întrerupere.

Există o categorie de întreruperi ce nu pot fi nici mascate, nici dezactivate. Acestea se numește întreruperi non-mascabile (Non-Mascable Interrupts - **NMI**).

Întreruperi pe AVR

În arhitectura AVR sunt mai multe registre I/O dedicate întreruperilor. Ele se grupează, în principiu, în:

- Registre de status - Conțin **flag-uri** de întrerupere, biți care indică dacă o întrerupere a fost sau nu generată
- Registre de control - Conțin **măști** de întrerupere, biți care indică dacă o întrerupere a fost sau nu demascată

Pentru ca un procesor AVR sa trateze o cerere de întrerupere (non-NMI) trebuie să fie adevărate 3 condiții:

- Întreruperile sunt activate la nivel global
  - Bitul `I` din registrul `SREG` este 1
- Întreruperea respectivă nu este mascată
  - Bitul corespunzător ei din registrul ei de control este 1, deci masca ei este 1
- Întreruperea respectivă este activă
  - Bitul corespunzător ei din registrul ei de stare este 1, deci flag-ul ei este 1

Tratarea întreruperilor pe AVR

În arhitectura AVR, tabela vectorilor de întrerupere se află întotdeauna la adresa 0. Fiecare intrare în tabelă reprezintă prima instrucțiune din rutina de tratare a acelei întreruperi.

În cazul în care nu vrem să facem nimic la întâlnirea unei întreruperi, acea instrucțiune va fi `RETI`, pentru a încheia rutina. Acesta este cazul comun.

În cazul în care vrem să executăm rutina `foo`, atunci acea instrucțiune va fi `RJMP foo`. Acesta ar trebui să fie mereu cazul pentru prima rutină, cea a întreruperii de reset.

În datasheetul ATtiny20 [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf], în capitolul *9. Interrupts*, pagina 36, este prezentata tabela vectorilor de întrerupere a acestui procesor:

Vector No.	Program Address	Label	Interrupt Source
1	0x0000	RESET	External Pin, Power-on Reset, Brown-Out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	PCINT1	Pin Change Interrupt Request 1
5	0x0004	WDT	Watchdog Time-out
6	0x0005	TIM1_CAPT	Timer/Counter1 Input Capture
7	0x0006	TIM1_COMPA	Timer/Counter1 Compare Match A
8	0x0007	TIM1_COMPB	Timer/Counter1 Compare Match B

9	0x0008	TIM1_OVF	Timer/Counter1 Overflow
10	0x0009	TIM0_COMPA	Timer/Counter0 Compare Match A
11	0x000A	TIM0_COMPB	Timer/Counter0 Compare Match B
12	0x000B	TIM0_OVF	Timer/Counter0 Overflow
13	0x000C	ANA_COMP	Analog Comparator
14	0x000D	ADC	ADC Conversion Complete
15	0x000E	TWI_SLAVE	Two-Wire Interface
16	0x000F	SPI	Serial Peripheral Interface
17	0x0010	QTRIP	Touch Sensing

În cod, dacă vrem să folosim întreruperi, înseamnă că programul nostru trebuie să urmeze următoarea structură:

```
    rjmp        main            ; Adresa 0x0000
    rjmp        INT0_ISR        ; Adresa 0x0001
    rjmp        PCINT0_ISR      ; Adresa 0x0002
    rjmp        PCINT1_ISR      ; Adresa 0x0003
    rjmp        WDT_ISR         ; Adresa 0x0004
    rjmp        TIM1_CAPT_ISR   ; Adresa 0x0005
    rjmp        TIM1_COMPA_ISR  ; Adresa 0x0006
    rjmp        TIM1_COMPB_ISR  ; Adresa 0x0007
    rjmp        TIM1_OVF_ISR    ; Adresa 0x0008
    rjmp        TIM0_COMPA_ISR  ; Adresa 0x0009
    rjmp        TIM0_COMPB_ISR  ; Adresa 0x000A
    rjmp        TIM0_OVF_ISR    ; Adresa 0x000B
    rjmp        ANA_COMP_ISR    ; Adresa 0x000C
    rjmp        ADC_ISR         ; Adresa 0x000D
    rjmp        TWI_SLAVE_ISR   ; Adresa 0x000E
    rjmp        SPI_ISR         ; Adresa 0x000F
    rjmp        QTRIP_ISR       ; Adresa 0x0010

main:
    <instr>                     ; Adresa 0x0011, prima instrucțiune a programului
    ...
```

Pentru un anumit tip de întrerupere, procesorul primește de la controller-ul de întreruperi un anumit vector (e.g. pentru întreruperea de overflow a Timer/Counter0 - TOV0 controllerul trimite procesorului vectorul TIM0_OVF_ISR, care este definit ca 0x000B). Procesorul execută un CALL virtual către vectorul primit de la controller-ul de întreruperi, ajungând astfel să execute codul de la adresa 0x000B. La această adresă procesorul găsește o valoare. Printre lucrurile plauzibile pe care le poate găsi aici sunt:

- O instrucțiune RJMP către rutina efectivă de tratare, dacă aceasta este implementată
- O instrucțiune RETI dacă aceasta nu este implementată
- Altceva - dacă tabela vectorilor de întrerupere conține orice altă valoarea, ea va fi pur și simplu executată ca o instrucțiune

Faptul că procesorul este programat ca, la pornire, să înceapă să execute cod de la adresa 0x0000 coincide cu faptul că la acea adresa se afla vectorul de întrerupere pentru reset. Deci funcția noastră main nu este, așadar, nimic altceva decât rutina de tratare a întreruperii de reset.

Activarea întreruperilor pe AVR

Bitul I din SREG indică faptul că întreruperile sunt activate/dezactivate la nivel global. Dacă bitul este 1, înseamnă că întreruperile sunt activate la nivel global. Dacă bitul este 0, înseamnă că sunt dezactivate la nivel global. Întreruperile pot fi activate la nivel global folosind instrucțiunea SEI (Set Interrupt). Ele pot fi dezactivate folosind instrucțiunea CLI (Clear Interrupt).

La începutul tratării unei întreruperi (deci la execuția instrucțiunii virtuale CALL_ISR) întreruperile sunt dezactivate. La sfârșitul tratării unei întreruperi (deci la execuția instrucțiunii RETI) întreruperile sunt activate.

Mascarea/demascarea întreruperilor se face în registre specifice fiecărui periferic. Pentru perifericul de timer, acel registru este TIMSK.

Exemplu de utilizare a întreruperilor

Vom configura Timer/Counter0 în modul de funcționare normal, cu prescaler 1. Vom configura timer-ul să genereze întreruperea de overflow. Vom configura rutina TIM0_OVF_ISR astfel încât să fie executată atunci când apare o astfel de întrerupere. Această rutină va încărca valoarea 42 în R31. La final, vom activa global întreruperile.

timer_interrupt_demo.asm

```
    TCCR0A      equ 0x19
    TCCR0B      equ 0x18
    TIMSK       equ 0x26

    rjmp        main            ; Adresa 0x0000
    reti                        ; Adresa 0x0001
    reti                        ; Adresa 0x0002
    reti                        ; Adresa 0x0003
    reti                        ; Adresa 0x0004
    reti                        ; Adresa 0x0005
    reti                        ; Adresa 0x0006
    reti                        ; Adresa 0x0007
    reti                        ; Adresa 0x0008
    reti                        ; Adresa 0x0009
    reti                        ; Adresa 0x000A
    rjmp        TIM0_OVF_ISR    ; Adresa 0x000B
    reti                        ; Adresa 0x000C
    reti                        ; Adresa 0x000D
    reti                        ; Adresa 0x000E
    reti                        ; Adresa 0x000F
    reti                        ; Adresa 0x0010

TIM0_OVF_ISR:
    ; Rutina doar încarcă valoarea 42 în R31.
    ldi         R31, 0x2A
    reti

main:
    ; Pornim Timer/Counter0.
    ldi         R16, 0b00000000 ; COM0A = 0 (normal port operation, OC0A disconnected)
                                ; COM0B = 0 (normal port operation, OC0B disconnected)
                                ; WGM0[1:0] = 0 (normal mode operation)
    out         TCCR0A, R16

    ldi         R16, 0b00000001 ; WGM0[2] = 0 (normal mode operation)
                                ; CS0 = 1 (clkT = clkIO/1, no prescaling)
    out         TCCR0B, R16

    ; Activăm întreruperea de overflow pentru Timer/Counter0.
    ldi         R16, 0b00000001 ; TOIE0 = 1 (Timer/Counter0 overflow interrupt enabled)
    out         TIMSK, R16

    ; Activăm întreruperile global.
    sei

    loop:
        rjmp loop
```

TL;DR

- Întreruperile sunt eveniment asincrone execuției programului
- Dacă apare o întrerupere, procesorul oprește execuția normală a programului după instrucțiunea curentă și execută o rutină de tratare

        - Tabela vectorilor de întreruperi conține câte o intrare pentru fiecare întrerupere care duce la rutina ei de tratare
        - În cadrul rutinei de tratare trebuie să dăm acknowledge la întrerupere
    - Întreruperile pot fi activate/dezactivate la nivel global și demascate/mascate la nivel individual
        - Excepție fac NMI
    - Pe AVR
        - Activarea/Dezactivarea la nivel global se face cu instrucțiunile SEI/CLI
        - Demascarea/Mascarea se face prin registrele de control
            - E.g. TIMSK
        - Tabela vectorilor de întreruperi se află la adresa 0
            - La intrarea într-o rutină de tratare întreruperile sunt dezactivate la nivel global
            - La ieșirea dintr-o rutină de tratare întreruperile sunt activate la nivel global
        - O întrerupere este tratată dacă
            - Întreruperile sunt activate la nivel global (bitul `I` din `SREG` este 1)
            - Întreruperea este demascată la nivel individual (masca ei este 1)
            - Întreruperea este activă (flag-ul ei este activ)

Exerciții

Task 01 (3p) Implementați logica controller-ului de întreruperi. Urmăriți comentariile marcate `TODO 1` din `interrupt_controller.v`.

Task 02 (2p) Implementați instrucțiunile `SEI` și `CLI`. Urmăriți comentariile marcate `TODO 2` din `decode_unit.v` și `control_unit.v`.

Task 03 (3p) Implementați instrucțiunea virtuală `CALL_ISR` și instrucțiunea `RETI`. Urmăriți comentariile marcate `TODO 3` din `decode_unit.v`, `signal_generation_unit.v` și `control_unit.v`.

Task 04 (4p) Plecând de la următorul schelet de cod, creați un program care schimbă starea pinului PA0 folosindu-se de o întrerupere, cât timp bucla principală este într-o buclă infinită. Simulați programul.

task04_skel.asm

```
    DDRA        equ 0x01
    PORTA       equ 0x02
    TCCR0A      equ 0x19
    TCCR0B      equ 0x18
    TIMSK       equ 0x26
    OCR0A       equ 0x16

    rjmp        main            ; Adresa 0x0000
    reti                        ; Adresa 0x0001
    reti                        ; Adresa 0x0002
    reti                        ; Adresa 0x0003
    reti                        ; Adresa 0x0004
    reti                        ; Adresa 0x0005
    reti                        ; Adresa 0x0006
    reti                        ; Adresa 0x0007
    reti                        ; Adresa 0x0008
    rjmp        TIM0_COMPA_ISR  ; Adresa 0x0009
    reti                        ; Adresa 0x000A
    reti                        ; Adresa 0x000B
    reti                        ; Adresa 0x000C
    reti                        ; Adresa 0x000D
    reti                        ; Adresa 0x000E
    reti                        ; Adresa 0x000F
    reti                        ; Adresa 0x0010

TIM0_COMPA_ISR:
    ; TODO 4: Schimbați (toggle) starea pinului PA0.
```

```
main:
    ; TODO 4: Configurați pinul PA0 ca și ieșire.

    ; TODO 4: Porniți un timer care să genereze o întrerupere de comparație pe canalul A. Puneți
    ; valoarea 42 ca valoare de comparație pentru canalul A.

    ; TODO 4: Activați întreruperile global.

    ; Deși programul pare să stea într-o buclă infinită, ar trebui să vedem că totuși starea
    ; pinului PA0 se schimbă.
    loop:
        rjmp loop
```

Resurse

- Schelet laborator
- AVRASM
- Datasheet ATTiny20 [http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8235-8-bit-AVR-Microcontroller-ATtiny20_Datasheet.pdf]
- Manual set de instrucțiuni AVR [http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf]

cn2/laboratoare/09.txt · Last modified: 2019/10/01 13:02 by tudor.visan