Pràctica de curs: SimulAVR
Un simulador de l'MCU AVR

Tecnologia de la Programació — iTIC

Sebastià Vila-Marta

Departament de Disseny i
Programació de Sistemes Electrònics

6 de maig de 2020

Índex

1 Introducció

Aquest és l'enunciat del treball de curs de TECPRO. L'objectiu d'aquest treball és essencialment implementar un simulador d'un microcontrolador (*Micro Controler Unit*) real. Un microcontrolador és un computador molt simple integrat en un sol xip que s'usa essencialment en aplicacions de control, [Wik11g].

Aquest treball simula un microcontrolador de la família AVR d'Atmel, [Wik11b; Atm11b]. La simulació no és completa ni exacta. Només se'n simula una part i certes peculiaritats de l'arquitectura dels AVR s'han simplificat per disminuir la quantitat de treball necessària.

La simulació ho és del nivell llenguatge màquina i es fa des d'un punt de vista estrictament funcional: no es volen simular els circuits digitals que implementen la MCU sinó la seva funcionalitat. La compleció del treball comporta tenir una comanda que permet simular un programa escrit en assemblador de l'AVR si aquest es restringeix al joc d'instruccions implementades.

Aquesta pràctica complementa la pràctica de *Mini AVR* de TECPRO, [Pal12], i és un exercici que avança alguns elements que després seran centrals a l'assignatura de DP.

2 Condicions

2.1 Equip de treball

El projecte està dissenyat per ser treballat en equip. Cal fer-lo en el marc del vostre equip de pràctiques habitual.

2.2 Entorn de desenvolupament

El projecte s'ha de desenvolupar obligatòriament usant les eines de subversion que hem posat a la vostra disposició. Es valorarà l'ús correcte d'aquestes eines atès que el seu correcte aprofitament és un dels objectius del projecte.

2.3 Lliuraments

A la fi del projecte caldrà lliurar:

- El codi font del projecte convenientment documentat.
- Els doctests corresponents.
- Una documentació Sphinx correctament organitzada.
- Un joc d'exemples que avalin el seu funcionament.

- La taula de dedicacions.

El lliurament final del projecte serà presencial. Caldrà que hi siguin presents tots els membre de l'equip. Les dates s'anunciaran oportunament.

2.4 La taula de dedicacions

En aquest projecte caldrà treballar més a consciència la taula de dedicacions. Ja sabeu que l'objectiu d'aquesta feina és anar prenent consciència del cost de les tasques de desenvolupament. El factor cost és essencial en enginyeria.

Cada persona de l'equip haurà de classificar el temps que va invertint en el projecte en els següents ítem:

1. *Estudi.* Inclou entre d'altres la dedicació a estudiar l'enunciat, els coneixements previs, a fer proves per entendre noves eines o conceptes, etc.

2. *Desenvolupament.* El temps dedicat a escriure nou codi.

3. *Disseny de tests.* El temps dedicat a escriure tests.

4. *Correcció d'errors.* El temps invertit corregint errors.

5. *Documentació.* El temps invertit documentant l'aplicació i generant els lliurables.

6. *Altres.* El temps dedicat a altres activitats: desplaçaments, reunions, consultes, etc.

En el cas de les activitats de desenvolupament, disseny de tests i correcció d'errors, caldrà acumular com s'han distribuït aquestes dedicacions en funció del mòdul en que es treballava.

Haureu de presentar una matriu en la que s'indica el temps dedicat per cada individu a cada activitat i comentar breument quines conclusions en podeu treure. També caldrà resumir la dedicació conjunta de tots els individus de l'equip, tabular-la convenientment i indicar el percentatge de dedicació a cada tasca que ha tingut l'equip en el seu conjunt.

A més, caldrà resumir convenientment les dedicacions del grup per activitat i mòdul, de forma que es pugui saber quin esforç s'ha endut cadascun dels mòduls del projecte.

2.5 Consells per afrontar el projecte

1. Invertiu el temps necessari a estudiar el problema i l'enunciat abans d'escriure cap línia de codi.

2. Invertiu un temps a pensar quina estratègia usareu per repartir-vos la feina de forma que l'equip rendeixi al màxim.

3. Aquest projecte és molt perepunyetes. Si no s'implementen les exactament com cal la simulació no funcionarà. Cal respectar al màxim l'especificació que us donem.

4. La correcció serà especialment precisa. Per corregir passarem tests específics, que hauran de funcionar perfectament.

5. Mireu que la mateixa persona que codifica les solucions no faci els doctests: és molt més fiable si els doctests els fan persones diferents a les que escriuen la classe o mètode.

6. És convenient fer doctests exhaustius. Quan cal doctest complicats no és interessant que estiguin escrits conjuntament amb la documentació (docstring) de cada mòdul, classe o mètode. En aquests casos es poden escriure doctests en fitxers separats, que també formaran part del projecte. És *molt* convenient escriure doctests separats en aquest projecte.

7. L'eina `nose` us pot facilitar molt la tasca de fer tests. Sap buscar els tests automàticament i passar tots els tests de tots els mòduls. També sap trobar els tests escrits separadament i executar-los si s'invoca convenientment (mireu-vos la seva man page). Aquests fitxers de test es poden escriure usant format ReST i així explicar els tests que es van fent. Encara més, es poden integrar a la documentació feta amb Sphinx a través del plugin d'Sphinx sphinx.ext.doctest.

3 Coneixements previs

Per abordar amb tranquil·litat el projecte és convenient tenir certes habilitats i coneixements previs a més dels que ja heu adquirit durant la vostra formació. Adquirir aquests coneixements i habilitats forma part també d'aquest treball i cal fer-ho abans de començar a implementar.

3.1 Coneixements generals

1. *Representació de naturals.*

   Cal entendre en la seva totalitat el concepte de representació dels naturals en diferents bases, saber escriure i llegir en base 2, 10 i 16 amb naturalitat i tenir un cert hàbit en la manipulació mental d'enters escrits en aquestes bases. Per aprofundir en aquest tema, podeu consultar qualsevol llibre de matemàtiques elementals.

2. *Representació d'enters en complement a 2*

   El complement a dos és una forma interessant de *representar* un número enter sobre un natural. És una de les formes corrents que usen els computadors per representar els enters (positius i negatius) usant un natural (un byte, p. ex.). És convenient entendre el sistema i com es treballa amb ell. Entre altres referències podeu consultar [Wik11j].

3. *Treball amb els formats de Python*

   Representar els tipus de dades com cadenes és quelcom habitual en molts treballs de programació. Sovint aquesta representació està envoltada de petits detalls: que si es volen dos dígits decimals, que si es vol el text alineat a la dreta d'una columna, etc. Facilitar això és la tasca dels formatadors. Python disposa de dues implementacions de formatador: l'antiga, que es basa en l'operador tant-per-cent i la moderna.

   En aquest projecte donar format a dades serà molt corrent. És convenient que us familiaritzeu amb la versió moderna del formatador de Python. Ho podeu fer amb una mica d'estudi del manual i algunes proves sobre l'intèrpret. La referència principal és el manual de la llibreria de Python, [Pyt11], apartat *7.1.3 Format String Syntax*

4. *Treball amb les operacions bitwise de Python*

   El projecte també comporta un treball intensiu amb les operacions bitwise de Python. Aquestes operacions ja les heu usat en la implementació de la darrera pràctica, la del simulador. És convenient que les tingueu per mà i no us causin confusió. Si cal, jugueu-hi una mica fins que les tingueu per ma.

És convenient que estudieu i practiqueu una mica aquests temes per abordar amb tranquil·litat el projecte. Segurament podeu compartir materials i notes usant el wiki de que disposeu.

3.2 L'arquitectura dels AVR

Les MCU's de la família AVR són màquines RISC d'arquitectura Hardvard, [Wik11c; Wik11i]. Per tant, tenen dues memòries diferenciades: una per encabir-hi el programa i una altra per encabir-hi les dades.

La memòria de programa (ProgramMemory) té una amplada de paraula de 16 bits. En ser una maquina de tipus RISC l'amplada de les instruccions de llenguatge màquina és constant i coincideix amb la de la memòria de programa: 16 bits.

La memòria de dades (DataMemory) té una amplada de 8 bits i les 32 primeres cel·les, amb adreces que van de 0x0000 a 0x001F, es coneixen com a registres i tenen un ús privilegiat. És a dir, hi ha moltes instruccions del repertori que operen específicament sobre aquests registres.

A banda dels dos bloc principals de memòria, també es disposa de:

- Registre comptador de programa (PC), que té 16 bits d'amplada i emmagatzema l'adreça de la següent instrucció a executar. El PC serveix per tant per indexar la memòria de programa.
- Registre de flags (FLAG), que te un byte d'amplada i en el qual cadascun dels bits té un significat concret i diferenciat. S'usa per emmagatzemar algunes característiques referents a com ha acabat l'execució de la darrera instrucció. Per exemple per saber si el resultat ha estat 0.

Naturalment també comptem amb una ALU, que és capaç de fer operacions amb dades de 8 bits i d'una unitats de control que gestiona el comportament de tota la màquina.

En aquest simulador no tindrem en compte molts altres detalls referents als AVR com ara les unitats d'entrada/sortida o certs aspectes peculiars de la seva arquitectura. A la figura 1 teniu un esquema de blocs d'aquesta arquitectura.

3.3 Repertori d'instruccions

Com ja sabeu, els computadors que es dediquen a executar instruccions del *llenguatge màquina*, [Wik11f], emmagatzemades en la memòria. Una instrucció, que té una longitud concreta mesurada en bits, està formada per diversos camps, cadascun dels quals té el seu paper. Sempre hi ha un camp que indica el que ha de fer la instrucció. És el que es coneix amb el nom d'*opcode*, [Wik11h]. A més, poden haver-hi altres camps que indiquin sobre quines dades cal fer l'operació indicada per l'opcode. Les dades es poden indicar seguint diverses estratègies. Per exemple, es pot donar el valor de la dada directament o bé indicar en quina adreça de memòria s'ha d'anar a buscar la dada. Aquesta estratègia rep el nom de *mode d'adreçament*, [Wik11a]. La figura 2 mostra una instrucció ADD de l'AVR i la seva interpretació. El conjunt de tipus d'operació (d'opcodes) que un computador sap executar s'anomena el *repertori d'instruccions*, [Wik11d]. Si voleu entendre millor aquests conceptes, el llibre de Tanenbaum, [Tan05], és una referència escaient.

En aquest projecte treballarem amb un repertori d'instruccions reduït però exactament compatible amb el dels microcontroladors AVR. D'aquesta forma es podran utilitzar les mateixes eines de treball que usaríem si estiguéssim desenvolupant sobre un microcontrolador real.

Les instruccions de llenguatge màquina de l'AVR són de 2 bytes d'amplada i el seu format varia segons el mode d'adreçament. La comprensió precisa del format de cada instrucció és essencial per a implementar el simulador atès que una de les tasques principals que fa és "trencar" cadascuna

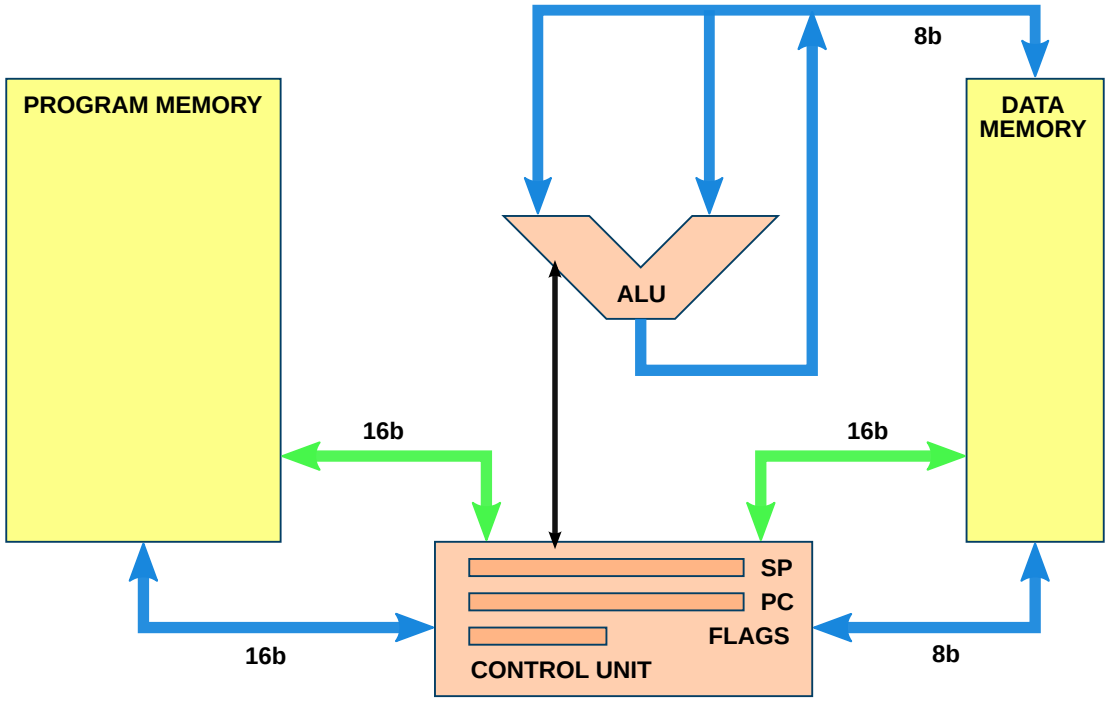

Figura 1: Arquitectura simplificada de l'AVR.

de les instruccions del programa en els camps oportuns per poder-la executar. La família AVR implementa un elevat nombre de modes d'adreçament. El simulador només implementarà els més bàsics:

- Adreçament directe d'un registre. Correspon a les instruccions que es refereixen directament a un sol registre amb el que operen.
- Adreçament directe de dos registres. Correspon a les instruccions que es refereixen a dos registres. En general són operacions binàries que obtenen les dades de dos registres i deixen el resultat en un d'aquests.
- Adreçament immediat. Quan una de les dades està codificada directament en la instrucció.

Les instruccions que implementarà el simulador són essencialment les aritmètiques bàsiques, les de moviment registre-memòria, les de salt i tractament de la pila. De forma resumida són les següents:

ADC Suma registre-registre amb carry.

SUB Resta registre-registre sense carry.

SUBI Resta registre-constant sense carry.

AND And registre-registre.

OR Or registre-registre.

EOR Or exclusiva registre-registre.

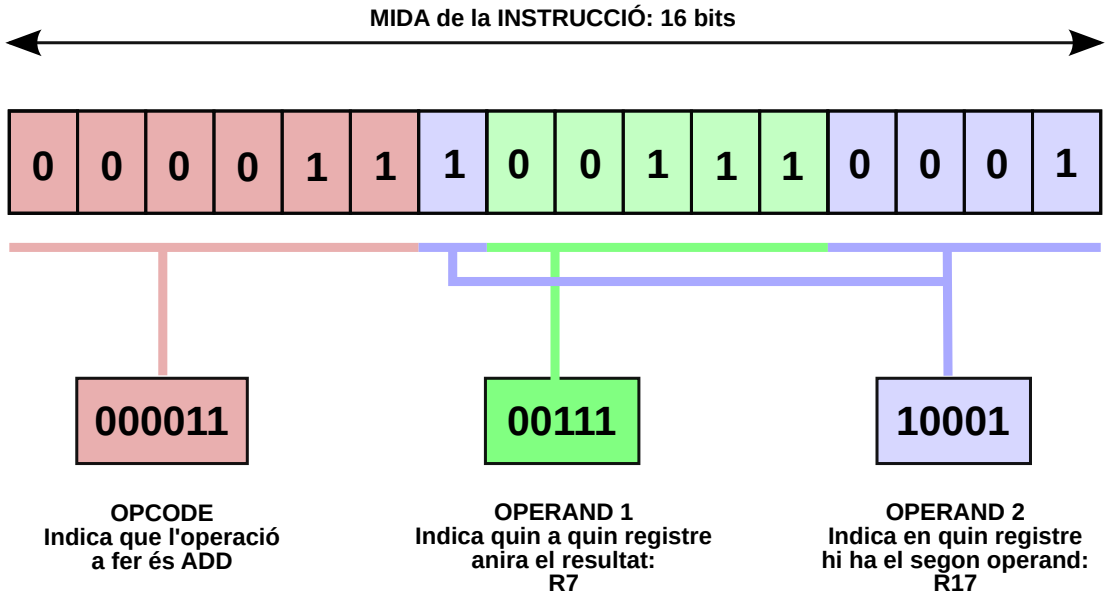

Figura 2: Instrucció ADD R7, R17 de l'AVR.

LSR Desplaçament dreta registre.

MOV Còpia registre-registre.

LDI Assigna valor a registre.

STS Còpia registre a memòria.

LDS Còpia memòria a registre.

RJMP Salt relatiu a una nova instrucció.

BRBS Salta a adreça de programa si cert bit de FLAGS és 1.

BRBC Salta a adreça de programa si cert bit de FLAGS és 0.

NOP No fa res. És la instrucció nul·la.

Aquestes instruccions es comportaran de forma idèntica que en els microcontroladors AVR. Per tant, la seva especificació precisa, tant pel que fan com pel format que tenen s'ha de consultar en la documentació tècnica del repertori d'instruccions de la família AVR, [Atm11a].

A banda de les instruccions anteriors també incorporarem les següents instruccions mantenint el mateix format però variant-ne lleugerament el significat per tal de fer més operatiu el simulador

BREAK Atura la simulació i acaba l'execució del programa simulador.

IN Quan s'aplica al port 0x0, llegeix un caràcter del teclat.

OUT S'usa per escriure per la pantalla. Quan el port és:

- **0x0** Escriu el valor del registre corresponent pel terminal en base 10.
- **0x1** Escriu el valor del registre corresponent pel terminal en base 16.

0x2 Escriu el valor del registre assumint que correspon a la codificació UTF d'un caràcter.

Noteu que les instruccions `IN` i `OUT` normalment s'usen per comunicar-se amb els perifèrics del microcontrolador a través dels ports de comunicacions. En aquest cas, com no simulem els ports de comunicacions, definim alguns *ports virtuals* que associem al teclat i la pantalla i usem les instruccions per a l'entrada/sortida dels programes assemblador.

4 L'estructura del projecte

4.1 Estructura de mòduls

El projecte s'estructura al voltant d'una sèrie de mòduls cadascun dels quals conté una o més classes que representen parts de l'arquitectura o conceptes relacionats. La llista dels mòduls i una petita descripció de cadascun és la que segueix:

bitvec Conté diverses classes que tenen com a objectiu representar les paraules de diferent longitud que intervenen en la simulació.

memory Conté diverses classes que representen les diverses tipologies de memòria de l'arquitectura de l'AVR.

state Conté una classe que representa l'estat (inclosa la memòria) del microcontrolador.

instruction Conté les classes que implementen el significat de totes i cadascuna de les operacions de llenguatge màquina que admet el simulador.

repertoir Conté una classe que agrupa el repertori d'instruccions del simulador.

avrmcu Conté una classe que implementa el control general del microcontrolador. És en certa manera la classe que aglutina la resta de components.

intelhex És un modul que no escriurem nosaltres sinó que, escrit per un tercer, l'incorporarem al nostre projecte. L'usarem per poder llegir amb facilitat programes en llenguatge màquina continguts en fitxers de format *Intel HEX*, [Wik11e].

avrexcep És un mòdul que defineix diverses classes d'excepcions usades en el simulador.

simavr És el mòdul principal del simulador. Els usuaris finals del simulador invoquen aquest mòdul per simular programes.

4.2 L'estructura de classes

El programa respon a l'estructura de classes que s'il·lustra en el diagrama de la figura 3.

La classe BitVector és la classe més bàsica de tot el disseny. Representa un paraula binària de mida "petita", és a dir, que, com a molt, s'allarga 32 bits. D'aquí se'n deriven les classes Byte i Word, que representen respectivament paraules de 8 bits i de 16 bits. Aquestes classes són usades àmpliament pel simulador.

La classe Memory representa una taula adreçable de paraules, és a dir un banc de memòria. D'ella se'n derives dues subclasses, DataMemory i ProgramMemory, que es diferencien essencialment per la mida de la paraula emmagatzemada. En el cas de DataMemory és d'1 byte i en el cas de ProgramMemory de 2. La capacitat, és a dir el nombre màxim de paraules que emmagatzemem, de les instàncies d'aquestes classes es configura en l'instant d'inicialitzar la instància.

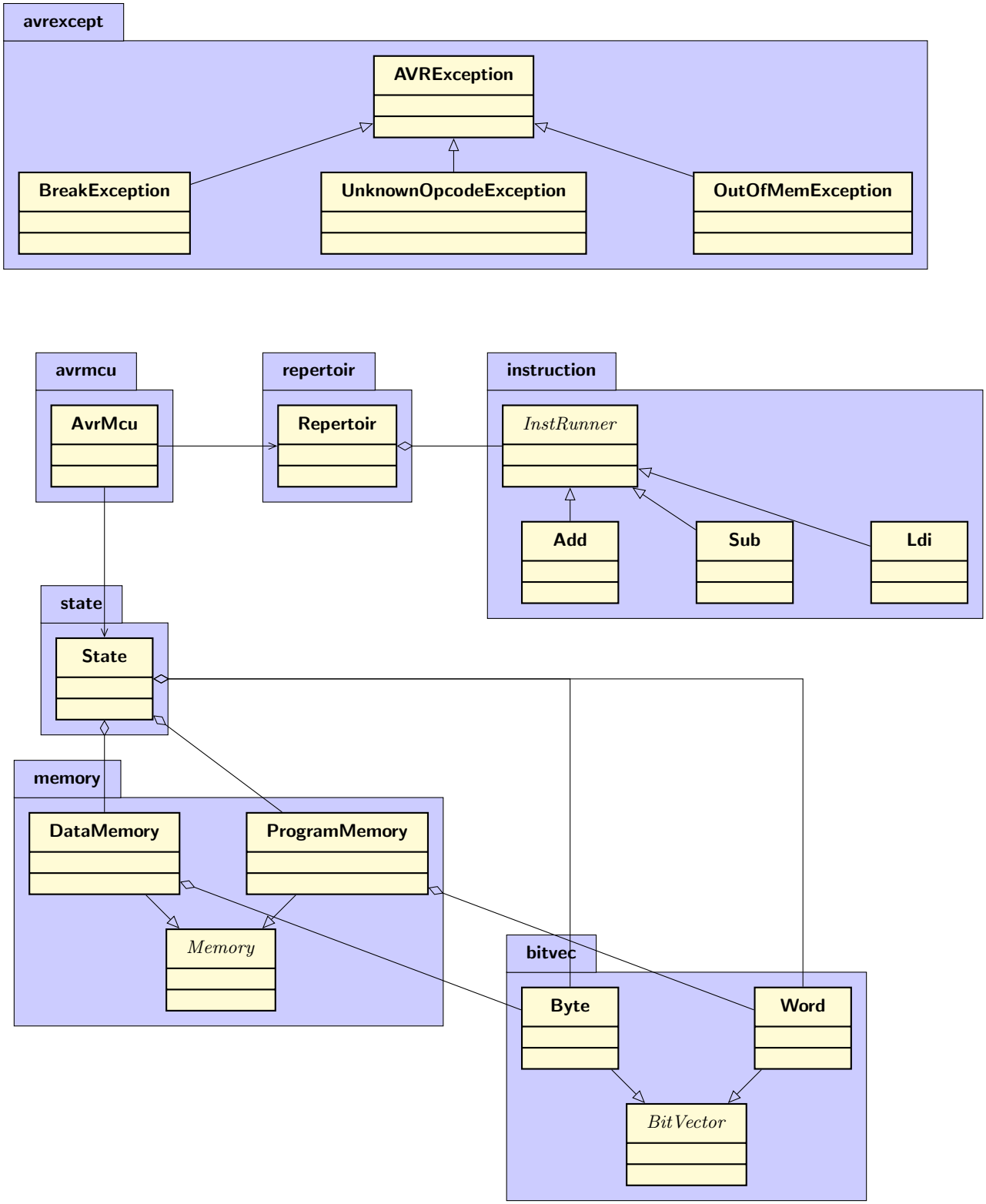

Figura 3: Diagrama de classes del simulador. El mòdul `instruction` nom mostra totes les classes per tal de fer el diagrama més clar.

La classe InstRunner és la superclasse d'uns objectes que concentren dues capacitats fonamentals. La primera, poder reconèixer instruccions de codi màquina que saben executar i la segona, saber executar aquestes instruccions. En aquesta classe i les seves subclasses (una per a cada tipus d'instrucció) es concentra el mecanisme que interpreta cada una de les instruccions del llenguatge maquina. Les classes com RegReg o Immediate són estrictament tècniques i la seva funció és simplificar l'escriptura de les seves subclasses.

Repertoir és un contenidor especialitzat en emmagatzemar conjunts d'instàncies de InstRunner's. La seva funció bàsica és aglutinar el repertori d'instruccions del simulador.

Finalment, AvrSim és la classe que representa el simulador del microcontrolador AVR. La seva principal funcionalitat és executar un programa en llenguatge màquina de l'AVR.

5 Especificació de les classes i mòduls

5.1 La classe ≪BitVector≫

Aquesta classe representa una paraula binària de mida arbitrària però menor o igual a 16 bits que s'interpreta sempre sense signe. És una classe abstracta i per tant no s'usa directament sinó que s'usa únicament com a super-classe. La representació d'aquesta paraula es fa sobre un **int** de Python, que sempre té pel cap baix 32 bits. Algunes particularitats es deleguen a les subclasses Byte i Word, que són paraules binàries de mida fixada en 8 i 16 bits respectivament. Entre aquestes la implementació del mètode especial __len__ que indica la mida de la paraula. Cal anar amb compte amb alguns dels mètodes ja que tornen sempre com a resultat un objecte de la mateixa classe que self i aquesta classe no és mai BitVector sinó Byte o Word segons convingui.

mòdul Aquesta classe cal implementar-la en el mòdul bitvec.

atributs Els atributs d'aquesta classe són:

1. _w, privat de tipus **int**

   Codifica el valor del BitVector

mètodes Els mètodes d'aquesta classe són els següents:

1. __init__(self, w=0)

   Inicialitza l'objecte amb un valor inicial de w. Cal assegurar-se que només s'usen els k bits de menys pes de w, on k és la longitud en bits de self. Això s'aconsegueix aplicant la màscara corresponent. Noteu que el valor de k dependrà de la subclasse concreta que s'estigui instanciant. Per exemple, si la subclasse és Byte, $k = 8$.

2. extract_field_u(self, mask)

   mask és un enter que s'interpreta com una màscara binària. Retorna sempre un enter positiu. L'objectiu d'aquest mètode és facilitar l'extracció de camps d'una paraula com ara el número de registre en el cas de la instrucció ADD de la figura 2. Donada una màscara mask que indica un subconjunt de bits de la paraula self, aquest mètode torna l'enter sense signe representat per aquest subconjunt de bits. Per exemple, si self=0b10100110 i mask=0b00110011 el resultat hauria de ser el enter 0b00001010, és a dir el 10 en decimal.

3. extract_field_s(self, mask)

Fa exactament el mateix que el mètode anterior però interpreta el resultat com un enter amb signe. Pot retornar doncs en un enter positiu o negatiu. S'assumeix que la codificació és complement a dos.

4. __int__(self)

   Torna el valor enter corresponent a self interpretat sempre com a un valor sense signe.

5. __index__(self)

   Torna el mateix que en el mètode anterior. Consulteu el manual de Python per entendre quin paper juga aquest mètode.

6. __repr__(self)

   Torna la representació en hexadecimal del valor del BitVector. Cal tenir en compte la llargada de la paraula i escriure sempre el nombre de dígits corresponents. Per exemple, si la llargada de la paraula és d'un byte i el valor és 10, __repr__ hauria d'escriure 0A i no pas A.

7. __add__(self, o)

   __sub__(self, o)

   Sumen i resten paraules. Retornen sempre un objecte de la mateixa classe que self. També admeten que el segon operand, o, sigui un **int** o un BitVector indistintament.

8. __and__(self, o)

   __or__(self, o)

   __xor__(self, o)

   __invert__(self)

   Implementen les corresponents operacions booleanes bit a bit. Els mètodes retornen un objecte de la mateixa classe que self.

9. __lshift__(self,i)

   __rshift__(self,i)

   Implementen les operacions de rotació esquerra i dreta. Retornen un objecte de la mateixa classe que self. En cas que i estigui fora de rang, el mètode aixeca l'excepció **IndexError**.

10. __eq__(self, o)

    __eq__(self, o)

    Implementen la igualtat. Cal tenir en compte que per decidir si dos BitVector són iguals només s'han de considerar els bits inclosos dins la mida de la paraula.

11. __getitem__(self, i)

    __setitem__(self, i, v)

    Permeten implementar l'accés als bits d'una paraula individualment. Si w és un BitVector, llavors es pot escriure w[3] per accedir al quart bit de la paraula. w[3] retorna un **bool** ja que els bits individuals els representem com a valors booleans. En cas que l'índex emprat superi la mida de la paraula, el llença l'excepció **KeyError**.

5.2 La classe «Byte»

Representa una paraula de 8 bits, és a dir un byte.

mòdul Aquesta classe cal implementar-la en el mòdul bitvec.

superclasses BitVector

atributs No té atributs propis.

mètodes Els mètodes d'aquesta classe són els següents:

1. __len__(self)

   Sempre torna l'enter 8 atès que un byte són 8 bits.

2. concat(self, b)

   Concatena self amb el Byte b i retorna un objecte de classe Word. Noteu que self és el MSB i b el LSB.

5.3 La classe «Word»

Representa una paraula de 16 bits.

mòdul Aquesta classe cal implementar-la en el mòdul bitvec.

superclasses BitVector

atributs No té atributs propis.

mètodes Els mètodes d'aquesta classe són els següents:

1. __len__(self)

   Sempre torna l'enter 16 atès que un word són 16 bits.

2. lsb(self)

   msb(self)

   Retornen el Byte de menys i més pes respectivament.

5.4 La classe «AVRException»

És una excepció que denota un problema en el simulador d'AVR. En general s'usa a través de subclasses.

mòdul Aquesta classe cal implementar-la en el mòdul avrexcep.

superclasses Exception

5.5 La classe «OutOfMemError»

És una excepció que denota un accés a un banc de memòria amb una adreça inexistent.

mòdul Aquesta classe cal implementar-la en el mòdul memory.

superclasses AVRException

5.6 La classe «Memory»

Aquesta classe representa un banc de memòria. La classe és abstracta i, per tant, no poden haver-hi objectes instanciats de la classe sinó que sempre ho són de les seves subclasses DataMemory i ProgramMemory.

atributs Els atributs de la classe són:

1. _m

   Una llista de Word o Byte que representa el banc de memòria.

2. _trace

   Un **bool** que indica si la funcionalitat de traça està activada.

mètodes Els mètodes d'aquesta classe són els següents:

1. __init__(self)

   Assigna **False** a _trace.-

2. trace_on(self)

   trace_off(self

   Activa i desactiva la funcionalitat de trace. Aquesta funcionalitat permet traçar els accessos al banc de memòria.

3. __len__(self)

   Retorna el nombre de cel·les de la memòria.

4. __repr__(self)

   Retorna un **str** que conté un bolcat del banc de memòria en un format exactament com el que segueix (en el cas que les cel·les siguin Byte):

   ```
   0000: 00
   0001: 01
   ```

5. dump(self,f=0,t=5)

   Retorna un **str** que conté un bolcat del banc de memòria exactament com en el cas de __repr__ però únicament de les cel·les que estan en l'interval d'adreces $[f, t)$.

6. __getitem__(self, addr)

   __setitem__(self, addr, val)

   Implementen les operacions d'accés a la memòria. addr és un **int** o qualsevol altre objecte que implementi el mètode __index__, en particular Word, i es correspon amb l'adreça de memòria que a la que vol accedir. val és un BitVector de la mateixa mida que la cel·la de la memòria, ja sigui Byte o Word.

   __getitem__ torna una paraula de la mida de la cel·la de la memòria.

   Si el valor de l'atribut _trace és **True** o, dit d'altra manera, si la funció de trace està activada, cada vegada que es fa un accés a la memòria s'escriu un missatge que indica:

   a) Quina operació s'ha fet: read o write.

   b) A quina adreça de memòria s'ha accedit.

   c) Quin valor s'ha llegit/escrit.

Per exemple, els missatges poden ser similars a:

```
Write C2 to   00A3
Read  1D from 010F
```

En cas que addr estigui fora de rang, aquestes operacions aixequen l'excepció OutOfMemError tot indicant com a missatge quelcom similar a `'Read from 0005 out of range'` o bé `'Write to 0005 out of range'` segons escaigui.

5.7 La classe «ProgramMemory»

Aquesta classe representa un banc de memòria per emmagatzemar programes. Per tant les dades emmagatzemades són de mida Word.

mòdul Aquesta classe cal implementar-la en el mòdul memory.

superclasses Memory

atributs No té atributs propis.

mètodes Els mètodes d'aquesta classe són els següents:

1. __init__(self,ncells=1024)

   Inicialitza un banc de memòria d'amplada Word i ncells cel·les.

5.8 La classe «DataMemory»

Aquesta classe representa un banc de memòria per emmagatzemar dades. Per tant les dades emmagatzemades són de mida Byte.

mòdul Aquesta classe cal implementar-la en el mòdul memory.

superclasses Memory

atributs No té atributs propis.

mètodes Els mètodes d'aquesta classe són els següents:

1. __init__(self,ncells=1024)

   Inicialitza un banc de memòria d'amplada Byte i ncells cel·les. Si ncells és menor de 32, el banc serà de 32 cel·les.

2. dump_reg(self)

   Retorna un **str** que representa els registres continguts en el banc de memòria en un format com el següent:

   ```
   R00: 00
   R01: 00
   ...
   R31: 00
   X(R27:R26): 0000
   Y(R29:R28): 0000
   Z(R31:R30): 0000
   ```

Flag	bit
CARRY	0
ZERO	1
NEG	2

Taula 1: Constants que faciliten l'accés als diferents bits d'estat (flags).

5.9 La classe «State»

Aquesta classe representa l'estat de la MCU. L'estat d'un computador està format pel conjunt de tots els registres i memòries. Cada vegada que s'executa una instrucció, l'estat acostuma a canviar.

mòdul Aquesta classe cal implementar-la en el mòdul state.

superclasses object

atributs Té un atribut públic per cada element de memòria o registre del AVR, en particular:

- data

  És el banc de memòria de dades de l'AVR.

- prog

  És el banc de memòria de programa de l'AVR.

- pc

  És un Word que implementa el registre comptador de programa (*Program Counter*).

- flags

  És un Byte que implementa el registre d'estat (*status*). Cada bit és un flag d'estat. En aquest simulador només s'usen els flags que es defineixen a la taula 1. S'hi accedeix a través d'unes constants que cal definir en el mòdul state de nom CARRY, ZERO i NEG que permeten indexar fàcilment aquest atribut públic.

mètodes Els mètodes d'aquesta classe són els següents:

1. __init__(self,data=128,prog=128)

   Inicialitza l'estat de la MCU. data és el nombre de cel·les de la memòria de dades i prog el nombre de cel·les de la memòria de programa.

2. dump_dat(self)

   Retorna un **str** que representa el bolcat de la memòria de dades.

3. dump_prog(self)

   Retorna un **str** que representa el bolcat de la memòria de programa.

4. dump_reg(self)

   Retorna un **str** que representa els registres continguts en l'estat. Això inclou també PC i flags. El format ha de ser similar a:

```
R00: 00
R01: 00
...
R31: 00
X(R27:R26): 0000
Y(R29:R28): 0000
Z(R31:R30): 0000
PC: 0000
CARRY: 0 ZERO: 0 NEG: 0
```

5.10 La classe ≪BreakException≫

És una excepció que es llença sistemàticament cada vegada que s'executa la instrucció `BRK`. S'usa per aturar la simulació.

mòdul Aquesta classe cal implementar-la en el mòdul instruction.

superclasses AVRException

5.11 La classe ≪InstRunner≫

En el simulador, cada instrucció del microcontrolador té associada una classe. Per exemple, la instrucció `ADD` té associada la classe Add. Una instància de la classe Add és capaç de reconèixer i simular qualsevol instrucció `ADD`.

La classe InstRunner és la super-classe (abstracta) de totes les classes lligades a instruccions. Concentra els (pocs) serveis comuns que aquestes classes lligades a instrucció tenen.

mòdul Aquesta classe cal implementar-la en el mòdul instruction.

superclasses object

mètodes Els mètodes d'aquesta classe són tots abstractes, és a dir les subclasses estan obligats a redefinir-los convenientment:

1. __repr__(self)

   Retorna una cadena que representa la instrucció.

2. match(self,instr)

   instr és un Word i denota una instrucció. Retorna **True** ssi aquesta instància pot executar la instrucció instr.

3. execute(self, instr, state)

   instr és un Word que denota una instrucció. state és una instància de la classe State. El mètode executa la instrucció i, com a resultat, modifica l'estat del microcontrolador al qual accedeix a través del paràmetre corresponent. Per poder executar la instrucció l'ha de descodificar, obtenir els operands (fetch), calcular el resultat, modificar convenientment el registre d'estat i emmagatzemar el resultat.

5.12 La classe «Add» i la resta de classes del mòdul «instruccio»

Una instància de la classe Add és un executador específic per a la instrucció `ADD`. Si invoquem el seu mètode execute() executa una instrucció `ADD` del microcontrolador.

mòdul Aquesta classe cal implementar-la en el mòdul instruction.

superclasses InstRunner

mètodes Els mètodes d'aquesta classe són tots els que cal redefinir de la super-classe, i.e., __repr__, execute() i match().

De forma anàlega a la classe Add cal implementar les classes Adc, Sub, Subi, And, Or, Eor, Lsr, Mov, Ldi, Sts, Lds, Rjmp, Brbs, Brbc i Nop.

Algunes instruccions tenen comportaments especials del simulador, que no es corresponen amb els del microcontrolador real. Aquest és el cas de `BREAK` que atura l'execució llençant l'excepció BreakException, o de `IN` i `OUT` que segueixen el que s'ha dit a l'apartat 3.3. Cal tenir-ho en compte a l'hora d'implementar les classes corresponents.

5.13 La classe «UnknownCodeError»

És una excepció que s'aixeca quan cal executar una instrucció de llenguatge màquina i aquesta no és coneguda.

mòdul Aquesta classe cal implementar-la en el mòdul repertoire.

superclasses AVRException

5.14 La classe «Repertoir»

Repertoir és una classe les instàncies de la qual són conjunt de InstRunner's. Representen el conjunt d'instruccions d'un MCU. La seva funcionalitat més característica és la que, donada una instrucció, retorna l'objecte InstRunner que és capaç d'executar-la.

mòdul Aquesta classe cal implementar-la en el mòdul repertoir.

superclasses **object**

atributs Els atributs d'aquesta classe són:

1. _li La llista d'objectes de classe InstRunner que constitueixen el repertori d'instruccions.

mètodes Els mètodes d'aquesta classe són:

1. __init__(self, li)

   li és una llista d'instàncies d'InstRunner que constitueixen un repertori d'instruccions.

2. find(self, instr)

   instr és un Word que denota una instrucció. El mètode retorna un objecte InstRunner capaç d'executar la instrucció instr. En cas que no existeixi cap InstRunner capaç d'executar la instrucció, aixeca l'excepció UnknownCodeError.

5.15 La classe «AvrMcu»

AvrMcu és una classe les instàncies de la qual són simuladors de l'MCU AVR. La seva funció més important és executar un programa escrit en assemblador de l'AVR.

mòdul Aquesta classe cal implementar-la en el mòdul avrmcu.

superclasses object

atributs Els atributs de la classe són:

1. _s de classe State. És l'estat del simulador.
2. _rep se classe Repertoir. És el repertori d'instruccions del simulador.

mètodes Els mètodes de la classe són:

1. __init__(self)

   Inicialitza el simulador. Particularment, fa un reset de l'estat: esborra les memòries, inicialitza el PC i les FLAGS a 0. Inicialitza el repertori d'instruccions amb les instàncies d'InstRunner corresponents.

2. reset(self)

   Fa un reset de l'estat deixant-lo de la mateixa forma que el mètode __init__.

3. set_prog(self, p)

   p és una llista d'enters que representen un programa en llenguatge màquina de l'AVR. El mètode instal·la el programa p en la memòria de programa del simulador a partir de l'adreça 0000.

4. dump_reg(self)

   Retorna un string que correspon amb un bolcat dels registres del simulador.

5. dump_dat(self)

   Retorna un **str** que representa un bolcat de la memòria de dades del simulador.

6. dump_prog(self)

   Retorna un **str** que representa un bolcat de la memòria de programa del simulador.

7. run(self)

   És el mètode principal del simulador. Quan s'invoca inicia una iteració infinita que:

   a) Obté la instrucció indicada pel PC.

   b) Busca un InstRunner que pugui executar aquesta instrucció.

   c) Executa la instrucció.

   El mètode té un catcher pel les excepcions UnknownCodeError i BreakException que actuen de forma conseqüent.

8. set_trace(self, t)

   Quan s'invova amb t=**True** activa el mode trace de la memòria de dades. Si s'activa amb t=**False** es desactiva el mode.

6 El programa principal

6.1 Manual d'usuari

El programa principal d'aquesta aplicació ha d'anomenar-se simavr. L'usuari l'ha de poder invocar de manera bàsica fent:

```
$ ./simavr programa.hex
```

El fitxer programa.**hex** conté el programa per al MCU AVR en el format estàndart HEX, [Wik11e].

La comanda simavr admet les següents opcions:

-p En acabar l'execució del simulador, es volca a pantalla la memòria de programa.

-r En acabar l'execució del simulador, es volca a pantalla els registres.

-d En acabar l'execució del simulador, es volca a pantalla la memòria de dades.

-t Activa la traça de les operacions sobre la memòria de dades.

Així, per exemple, si invoquem el simulador així:

```
$ ./simavr -rt prog-prova.hex
```

executa el programa de prova, va mostrant com evolucionen els accessos a la memòria de dades i en acabar bolca l'estat dels registres.

6.2 Implementació

Per implementar el programa principal cal tenir en compte els següents detalls:

1. Cal inicial el fitxer amb el shebang:

   ```
   #!/usr/bin/python
   # -*- coding: utf-8 -*-
   ```

   i modificar-li els permisos d'execució fent:

   ```
   $ chmod a+x simavr
   ```

   D'aquesta forma el vostre programa Python es converteix en una comanda que podeu executar directament, sense necessitat d'invocar l'intèrpret de Python.

2. Per «captar» les dades que un usuari escriu darrera la comanda, recodeu que podeu usar sys.argv, com ja heu fet en altres pràctiques.

3. El fitxer en format HEX conté un programa en llenguatge màquina per a l'AVR. El format HEX és un estàndart per a emmagatzemar aquests tipus de dades. Com que analitzar el contingut d'un fitxer HEX és complicat, és interessant usar un mòdul que ens faciliti aquesta feina.

   El projecte IntelHex, [Bel11], subministra un mòdul de python que permet processar fàcilment fitxers en format HEX. En el programa principal és convenient usar la classe IntelHex16bit del mòdul intelhex per tal de poder llegir les instruccions de llenguatge màquina contingudes en el fitxer HEX que l'usuari vol simular. El funcionament precís d'IntelHex16bit el trobareu en la documentació del mateix mòdul.

4. El programa principal del simulador segueix essencialment els següents passos:
   a) Analitza les dades que l'usuari dona en la línia de comandes.
   b) Usant el mòdul intelhex llegeix el fitxer amb el programa en llenguatge màquina.
   c) Crea una una instància de AvrMcu, inicialitza el programa i executa el mètode run().

5. Com es creen els fitxers HEX? És molt senzill, hi ha un programa anomenat *assemblador* que fa aquesta feina. Instal·leu-vos l'assemblador de l'AVR en el vostre computador fent:

   ```
   $ sudo aptitude install avra
   ```

   Avra, [JWH11], és un assemblador lliure per a la família AVR. Usar-lo és molt senzill. Escriviu un programa en assemblador usant només les instruccions que heu implementat. Per exemple, escrivim el programa exemple1.asm així:

   ```
   LDI R16, 0xff
   LDI R17, 1
   ADD R17, R17
   MOV R0, R17
   AND R0, R16
   BREAK
   ```

   Ara l'assemblador ens pot traduir aquest programa a codi màquina de forma automàtica fent:

   ```
   $ avra exemple1.asm
   ```

   Aquesta comanda genera diversos fitxers, entre els quals el fitxer exemple.**hex** que és el que ens interessa a nosaltres. També podem generar un llistat en el que es mostra el codi màquina resultat. Proveu de fer:

   ```
   $ avra -l exemple1.lst exemple1.asm
   ```

   Observeu el fitxer exemple1.lst i mireu d'entendre el seu significat.

   Un exemple més elaborat amb el que podeu jugar és aquest:

   ```
           ;; Escriu l'abecedari
           EOR R0, R0              ; R0 = 0
   LAB1:
           LDI R17, 65             ; la base del codi ASCII
           ADD R17, R0
           OUT 3, R17              ; escrivim caracter

           SUBI R0, -1             ; iteracio
           LDI R17, 26
           SUB R17, R0
           BRBC 1, LAB1
           BREAK
   ```

   De cara a comprovar el funcionament del vostre emulador és interessant que escriviu diversos programes amb assemblador i els simuleu.

Referències

[Atm11a] Atmel. *8 bit AVR Instruction Set*. Anglès. 2011. URL: http://www.atmel.com/atmel/acrobat/doc0856.pdf.

[Atm11b] Atmel. *Atmel AVR 8- and 32-bit*. Anglès. 2011. URL: http://www.atmel.com/products/avr.

[Bel11] Alexander Belchenko. *IntelHex Project*. Anglès. 2011. URL: https://launchpad.net/intelhex.

[JWH11] Jerry Jacobs, Tobias Weber i John Anders Haugum. *AVRA — Assembler for the Atmel AVR microcontroller family*. Anglès. 2011. URL: http://avra.sourceforge.net.

[Pal12] Pere Palà. *Mini AVR 1*. Digital Systems course notes. Anglès. Enginyeria de Sistemes TIC — Universitat Politècnica de Catalunya, abr. de 2012. URL: http://ocw.itic.cat/assignatures/sd/mini-avr-part-1.

[Pyt11] Python Software Foundation. *The Python Standard Library*. Anglès. 2011. URL: http://docs.python.org/release/2.6.6/library.

[Tan05] Andrew S. Tanenbaum. *Structured Computer Organization*. Anglès. 5a edició. Prentice-Hall, juny de 2005. ISBN: 978-0131485211.

[Wik11a] Wikipedia. *Addressing Mode — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Addressing_mode.

[Wik11b] Wikipedia. *Atmel AVR — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Atmel_AVR.

[Wik11c] Wikipedia. *Harvard Architecture — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Harvard_architecture.

[Wik11d] Wikipedia. *Instruction Set — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Instruction_set_architecture.

[Wik11e] Wikipedia. *Intel HEX — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Intel_HEX.

[Wik11f] Wikipedia. *Machine Code — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Machine_code.

[Wik11g] Wikipedia. *Microcontroller — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Microcontroller.

[Wik11h] Wikipedia. *Opcode — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Opcode.

[Wik11i] Wikipedia. *Reduced Instruction Set Computing — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Reduced_instruction_set_computing.

[Wik11j] Wikipedia. *Two's Complement — Wikipedia, The Free Encyclopedia*. Anglès. 2011. URL: http://en.wikipedia.org/wiki/Two%27s_complement.