LIRK
Lisp AVR with assembly Kernel

Łukasz Dubiel

22 marca 2012

1 Wprowadzenie

Na platformie AVR istnieje już kilka języków mających ugruntowaną pozycję oraz zestaw narzędzi. Kilka głównych z nich to C, Basecom czy Pascal. Większość z nich to języki niezależne od platformy. Jednak mając takie zabawki jakimi są mikrokontrolery z rodziny AVR operacje języki które operują na pełnych słowach czy wielokrotnościach stają się częściowo niewygodne. Dlatego postanowiłem spróbować innego podejścia do problemu.

1.1 Lisp

W roku 1958 John McCarty na MIT stworzył język programowania który został nazwany Lispem. Był to drugi, zaraz po FORTRANie język wysokiego poziomu. Przez ten czas Lisp ewoluował, jak również pojawiały się różne dialekty (oczywiście nie kompatybilne z sobą) aż stworzyły całą rodzinę języków. Obecnie gdy używa się słowa Lisp to mówi się o całej rodzinie języków lub o Common Lispie, dialekcie ustandaryzowanym przez ANSI w 1994 z późniejszymi modyfikacjami.

1.2 Motywacje

Z racji mojego dużego zainteresowania językami z rodziny Lispów oraz problemów jakie napotyka C w momencie kiedy trzeba operować na pojedynczych bitach a nie słowach zdecydowałem się na stworzenie "dialektu" Lispa na rodzinę AVR.

Początkowo miało to być ćwiczenie w celu lepszego poznania Common Lispa, lecz potem zdecydowałem się, że to właśnie jego będę wykorzystywał w moich hobbystycznych projektach.

2 Zamierzenia i cele

Lirk nie ma (na razie) ambicji stania się nowym językiem programowania. Ma jedynie udostępniać funkcje które zminimalizuje ilość "nie potrzebnego" kodu w konstrukcjach stricte assemblerowych. W dalszej części referatu przedstawię cele.

2.1 Ewaluowanie parametrów instrukcji

Do działania Lispa faktycznie potrzebne są dwa byty. Reader i Evaluator. Pierwszy z nich ma za zadanie sparsować dane oraz rozwinąć tzw reader-macra (proste aliasy na najczęściej spotykane

konstrukty językowe takie jak ′ na quote czy #′ na function) oraz wszystkie nazwy zmieni na duże litery (lisp jest caseinsensitive). Drugi z nich evaluator za jedno zadanie wyznaczyć wartość wyrażenia które stworzył reader.

Daje to dość duże możliwości (przykładem jest system makr lispowych), które możemy wykorzystać by wzmocnić ekspresywność assembly.

```
(defun get-number ()
  (LD R20 X+)
  R20)

(main-loop
 (out PORTA (get-number)))
```

Rozwiązanie jest podobne do zagnieżdżaniu funkcji w C jednak tam się to dzieje w momencie wykonania, a Likr używa mechanizmu podczas generowania pliku wykonywalnego. Oczywistym jest, że mamy dostępny dowolny ficzer Common Lispa.

2.2 Definicje oraz wołanie procedur

Jednym z pierwszych problemów jakie nękały programistów była duplikacja kodu. Były badania wskazujące na to, że średnia ilość błędów na jednostkę kodu (KLOC czyli tysiąc linii) jest stała i zależy do wielkości projektu [Potrzebne źródło]. Z racji tego, że assembler jest najmniej ekspresywnym językiem (ilość kodu potrzebna do wykonania zamierzonej akcji jest dość pokaźna) jest podatny na duplikacje.

Oprócz niskopoziomowych aspektów programowania interesuję się bardziej ogołnymi technikami jak i metodykami. Bardzo spodobał się mi manifest programowania zwinnego (ang. agile) oraz późniejszy, chociaż bardziej ważny dla programistów manifest Software Craftsmanship, którego jednej z głównych haseł (obok programowych) jest "Don't repeat yourself." Hasło bardzo ważne, gdyż powtarzanie kodu jest jedną z głownych przyczyn bugów.

Jak widać "klasyczne" assembly, ze względu na swoją ekspresywność, stoi dość mocnym kontraście względem programów tych manifestów.

Na tak małej platformnie jaką jest AVR bardzo dobrze sprawuje się programowanie proceduralne. Na podobnych platformach działało C 30 lat temu. Lekko problematyczne w C jest to iż, każda procedura działa na zasadzie skoku i powrotu (standard C99 zaczerpnął z C++ słowo kluczowe inline, jednak sprawia on, że funkcja nie zawsze jest rozwijana w miejscu i zależy to od wielu czynników).

Lirk z zamierzeniu ma oferować dla programisty obie te metody. Procedura może być wywołana przez skok z odłożeniem adresu powrotu na stos, jak i być rozwinięta w miejscu.

funkcję można wywołać bezpośrednio w celu rozwinięciu jej w miejscu.

```
(defun wait()
  (let ((lab (make-label
              :name (string
                     (gensym)))))
    (ldi R16 250)
    (make-asm-label lab)
    (dec R16)
    (brne lab)))

(main-loop
 (cbi R2 1)
 (out PORTB R2)
 (wait)
 (sbi R2 1)
 (out PORTB R2)
 (wait))
```

```
main:
        cbi R0, 1
        out PORTB, R0
        ldi R16, 250
G676:
        dec R16
        brne G676
        sbi R0, 1
        out PORTB, R0
        ldi R16, 250
G698:
        dec R16
        brne G698
        rjmp main
```

albo wywołać jak wykonując skok
[listing z wykonaniem w programie]

```
(defun wait()
  (let ((lab (make-label
              :name (string
                     (gensym)))))
    (ldi R16 250)
    (make-asm-label lab)
    (dec R16)
    (brne lab)))

(main-loop
 (cbi R2 1)
 (out PORTB R2)
 (call wait)
 (sbi R2 1)
 (out PORTB R2)
 (call wait))
```

```
main:
        cbi R0, 1
        out PORTB, R0
        rcall wiat
        sbi R0, 1
        out PORTB, R0
        rcall wait
        rjmp main

wait:
        ldi R16, 250
G698:
        dec R16
        brne G698
        ret
```

Oczywiście w tym przypadku kompilator automatycznie zarządza etykietami oraz dodanie odpowiedni fragment kodu z instrukcjami powrotu.

2.3 Przerwania

Na przerwania można popatrzyć jak na specyficznie definiowane procedury. Procedury które wyzwalane są eventami generowanymi przez procesor. Wiadome jest że na samym początku przestrzeni programu znajduje się tablica wektorów przerwań.

Skoro jest to specyficzna funkcja możemy sobie zdefiniować makro które ułatwi nam jej definiowanie

```
(defmacro definterrupt (v &body body)
  `(defun ,v ()
     ,@body
     (iret)))
```

Dzięki niemu możemy definiować przerwania w następujący sposób.

```
(definterrupt INT0
  (sbi R1 0)
  (nop)
  (nop)
  (nop)
  (cbi R1 0))
```

2.3.1 Dalsze plany

Dodatkowo w planach jest dodanie możliwości, że raz wołana funkcja jest rozwijana w miejscu, bez dodania niepotrzebnych instrukcji i etykiet, ale jeżeli programista wywoła procedurę powtórnie, w assembly pojawiają się wszystkie potrzebne elementy by wykonać odpowiednie przejścia.

Rozwiązanie umożliwia większą granulację kodu, oraz ułatwia testowanie jednostkowe. Nawet można się pokusić o użycie TDD [http://www.agiledata.org/essays/tdd.html] jednak wymaga to napisanie odpowiednich narzędzi (emulatorów platformy,narzędzi do budowania, framework testowy), które nie byłyby aż tak problematyczne (przynajmniej w ograniczonej formie).

2.4 Tworzenie rozgałęzień sterowania

Asembler oferuje kilkanaście instrukcji warunkowych. Są one w postaci predykatów. Język predykatowy jest naturalnym sposobem tworzenia warunków w językach funkcyjnych.

```
if(PINB == 0)
{
  PORTB |= 0x01;
}
else
{
  PORTB |= 0x02;
}
```

```
(if (zerop PORTB)
    (sbi PORTB 1)
    (sbi PORTB 2))
```

```
    tst PORTB
    brnq else
    sbi PROTB 1
    rjump end
else:
    sbi PORTB 2
end:
```

Widać, że kod w Common Lispie (którego składnie przyejmuje Lirk) wygląda inaczej niż analogiczny kod w C. Różniece nie są duże i bardzo łatwo je przeskoczyć. Dodatkowo podobnie jak w Pythonie a nawet bardziej kod programu przypomina (z drobnymi modyfikacjami) tekst pisany w języku naturalnym opisujący sposób działania. Jednak z racji użycia jako podstawowych funkcji assemblera jest to lekko zaciemnione. Rozwiązaniem jest dodanie aliasów do poszczególnych instrukcji.

Utrzymywanie bardzo rozgałęzionego kodu w assembly wymaga bardzo dużego skupienia oraz bardzo rozległego modelu mentalnego. W przypadku assembly dużo więcej jest w głowie programisty

niż jest napisane. Lirk ma na celu lekkie odzielenie oraz osłabienie wymagań skupienia programisty oraz większe zgrupowania akcji powiązanych z sobą w logiczną całość.

Rozwiązania oparte na predykatach (które w większości stosuje programowanie funkcyjne) bardzo dobrze mapuje się na instrukcje rozgałęzienia sterowania w kodzie, oraz zmniejsza ryzyko popełnienie błędu przez programistę (brak drugiego znaku równości podczas porównywania)

3 Narzędzia

Obecnie Lirk nie posiada żadnych narzędzi, jednak w raz z rozwojem to się zmieni.

3.1 IDE

W pierwszych fazach rozwoju Lirk'a używany będzie Emacs. Oczywiście trzeba napisać główny tryb działania który będzie wspomagała tworzenie w Lirku. Istnieje duża szansa na wykorzystanie trybu Lispowego jako bazy, którą trzba byłoby wzbogacić o kolorowanie mnemoników assemblera.

Bardzo pomocnym trybem może okazać się auto-complete który ułatwi używanie funkcji.

Jest to troszeczkę pracy, jednak dzięki temu tworzenie w liku będzie prostsze.

3.2 Budowanie

Oczywistym jest że obecnie nie mam stworzonego narzędzia do budowania dla Lirka. W tej roli wykorzystywany jest poczciwy GNU Make, który wywołuje na razie intereter lispa z odpowiednimi argumentami.

Sądzę że Lirk może się pokusić o wykorzystanie własnego systemu do budowania. Powodem do jego tworzenie będzie możliwa integracja z programatorami, jak i innymi narzędziami które będą budowane dla Lirka.

Gdy korzysta się z TDD oczywistym jest że będziemy potrzebować narzędzia do tworzenia testów, jego odpalania jak i zbierania informacji.

Jako środowisko do odpalania testów chciałbym wykorzystać projekt simavr jako emulatora platformy AVR. Implementuje on znaczną część funkcjonalności znacznej części mikrokontrolera. Umożliwi to na ,łatwe badanie stanu rejestrów dzięki czemu wykonywanie testów jednostkowych może być dość szybie.

Do procesu TDD oprócz środowiska potrzebny jest mechanizm definiowania testów. W tym miejscu potrzebna będzie biblioteka która pozwoli na łatwe definiowania stanu początkowego, definicja testu oraz system asercji na wartości rejestrów oraz stan pinów. Dzięki temu kod będzie można testować ze względu na jego zachowanie oraz to co robi, dzięki czemu będzie możliwe usunięcie błędnego działania zaraz po jego wprowadzaniu. Zaoszczędzi to czasu na debugowanie które jest dość trudną operacją podczas pisania w assemblerze.

3.3 Pliki specyficzne dla sprzętu

W assmeblerze istnieją pliki definiujące offsety poszczególnych rejestrów oraz stałe dla konkretnego modelu mikroprocesora. By móc zapewnić polimorfizm ze względu na rejestr Lirk również musi posiadać takowy plik który zawierałby podobne dane do jego assemblerowego odpowiednika.

Posiadanie takowego pliku jest również podyktowane tym, że gdzieś w kodzie musi następować dołączenie do kodu assembleroweogo nagłówka z dołączeniem pliku urządzenia. Tak więc plik z

definicjami dla Lirka musiałby istnieć, oraz można go rozszerzyć o kilka stałych oraz rzeczy zależnych od modelu.

Jest wielka szansa na to, iż w ramach jednej rodziny kod pomiędzy modelami będzie dość przenośny dzieki takiemu rozwiązaniu.

4 Podsumowanie

Obecnie Lirk realizuje tylko fragment z opisanego elementu. Spowodowane jest to jeszcze brakiem możliwości zmiany wyjścia gdzie wypisywane są mnemoniki, ale trwają nad tym prace gdyż jest to jeden z głównych celów rozwoju.

Jedynym ograniczeniem jest tu brak czasu (zajęcia na uczelni, oraz inne projekty i zobowiązania). Jednak mimo ograniczeń Lirk powoli ale się rozwija.