Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

До захисту допущено
Завідувач кафедри
_____ Теленик С.Ф.
“___” _____ _____

Пояснювальна записка

до дипломної роботи освітньо-кваліфікаційного рівня «бакалавр»
з напрямку підготовки
6.050201 «Системна інженерія»

на тему: *Розробка мультизадачного Forth-ядра для мікроконтролерів AVR*

Студент групи ІА-72 Глинський Данило Євгенович _____

Керівник роботи ст. викладач Глушко Є. В. _____

Київ — 2011

Зміст

Вступ

Написання програм під 8-бітні мікроконтролери є важливою навичкою для спеціаліста по автоматизації. Разом з тим, іноді важливішими постають задачі прототипування програми у певному зручному середовищі, наприклад, при навчанні, освоєнні нової технології, експериментальних запусках. Важливо, щоб таке середовище було інтерактивним, що сприяє зменшенню часу відлагоджувального процесу. В якості одного такого середовища може виступати ядро на основі Форта.

Форт складається з двух основних частин: адресний інтепретатор і Форт-інтепретатор. Перший грає роль віртуальної машини і довзоляє виконувати динамічно-змінний код навіть на процесорах з гарвардською архітектурою пам'яті. Другий довзоляє динамічно виконувати (або компілювати у зрозумілий для адресного інтепретатора) код, записаний за допомогою ASCII символів. За рахунок даних особливостей Форт може використовуватись як середовище швидкого прототипування для мікроконтролерів.

Окрім того, Форт являє собою цінність у наш час через наявність на ринку апаратних форт-процесорів GreenArrays, з нуль-операндним Форт-асемблером і великою кількістю ядер (до 144) з високою продуктивністю. Даний процесор може скласти хорошу конкуренію існуючим на ринку цифрової обробки сигналів, і якщо це станеться, то уміння програмувати на Форті стане потрібним.

Окремо, проте в рамках даної роботи, розглянуто техніку модифікації програмного середовища під проект, а саме розширення макроасемблера скриптовим макропрепроцесором з оригінальним синтаксисом. Дана технологія відноситься до метапрограмного підходу в програмуванні і розширює звичайний асемблер кодогенеративними можливостями.

1 Аналіз існуючих засобів

Освоєння нової технології завжди супроводжується проблемами. Основними є:

- відсутність важливих наукових доробок
- недостатньо організована документація
- присутність недоліків реалізації технології
- дуже похила крива навчання

У випадку використання мікроконтролерів для знайомства із особливостями організації мікропроцесорних обчислень можна зіткнутись з кожною з вищеназваних проблем. Саме тому потреби спеціалістів по комп'ютерним технологіям є повищеними у сучасному світі.

Дана ситуація, хоч і вставновлена у світі, проте не є особливо привабливою для роботодавців. Останнім потрібно, щоб технологія була якомога простіша у освоєнні і якомога зручніша для підтримки, за рахунок чого можна знизити мінімальний рівень володіння технологією спеціаліста і здешевити процес. З даної точки зору, будь-який процес, направлений у сторону спрощення технології або спрощення її сприйняття, є вигідним для замовника і/або роботодавця.

Одним з способів спрощення освоєння нової технології є використання проміжних шарів. Важливими складовими проміжного шару є:

- простіший підхід до технології, абстрагування від складних для розуміння особливостей
- зручніший інтерфейс з людиною
- хороша документація

Наприклад, даним умовам відповідає мова програмування C, інтегрована система розробки AVR Studio, певний набір сайтів з інтернету, набір типових алгоритмів. Іншими прикладами є відлагоджувальний стенд, операційна система для мікроконтролера, система з термінальним доступом до мікроконтролера, командний рядок для управління.

Приклад із навчальним стендом варто розглянути окремо. Стенд містить у собі певну кількість електронних компонент і приховує від користувача складні моменти проектування зв'язків між цими компонентами, фактично звільнює програміста від непрограмних задач.

Додатковим способом спрощення взаємодії між електронікою і людиною — це абстрагування від апаратного рівня. Для цього зручно використовувати програмні середовища, такі як операційні системи або Форт системи. Опишемо їх більш конкретно.

1.1 Операційні системи

Мікроконтролери AVR є одними з найпопулярніших на ринку. Завдяки цьому було породжено велика кількість операційних систем. Рогзлянемо тільки найпростіші, операційні системи для 8 і 16 бітних мікроконтролерів.

- Мультизадачна система реального часу CMX
- Femto OS
- AVRILOS
- FreeRTOS

Усі вищеназвані операційні системи працюють по одному принципу, показаному на Рисунку 1.1.

У деяких з них реалізовано термінал, проте можливості виконати будь-який код віддалено — немає.

1.2 Форт і його реалізації

Форт (англ. Forth) — це мова і середовище програмування. Він вважається найпростішою мовою для реалізації на асемблері[1)] і при цьому являється мовою високого рівня. В своєму стандартному вигляді він забезпечує:

- компіляцію і виконання скомпільованих програм

[1)] при використанні метрики «розмір коду»

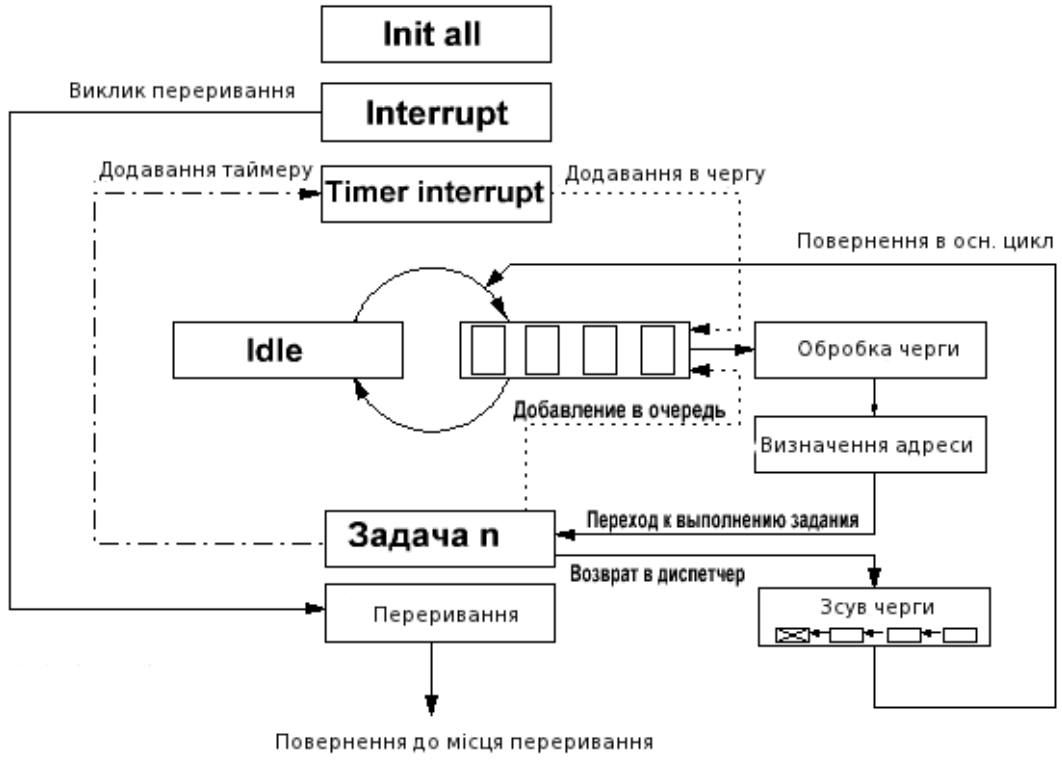

Рисунок 1.1 – Архітектура багаторівневої багатозадачності

– інтерпретацію текстової програми
– підтримку базових пристроїв вводу/виводу (термінал)
– кооперативну багатозадачність
– невеликі потреби у ресурсах для функціювання
– гнучкість і розширюваність, створення специфічних до задачі мов (DSL)

Найвідомішою існуючою реалізацією Форта для AVR мікроконтролерів є проект AMforth. Іншою відомою реалізацією Форта для AVR є avrforth. Обидві ці реалізації використовують прямий шитий код і перезаписують пам'ять ПЗП мікроконтролера.

1.3 Мікроконтролери AVR

AVR являє собою 8-розрядний RISC мікроконтролер, що має швидке процесорне ядро, Flash-пам'ять програм, пам'ять даних SRAM, порти введення/ви-

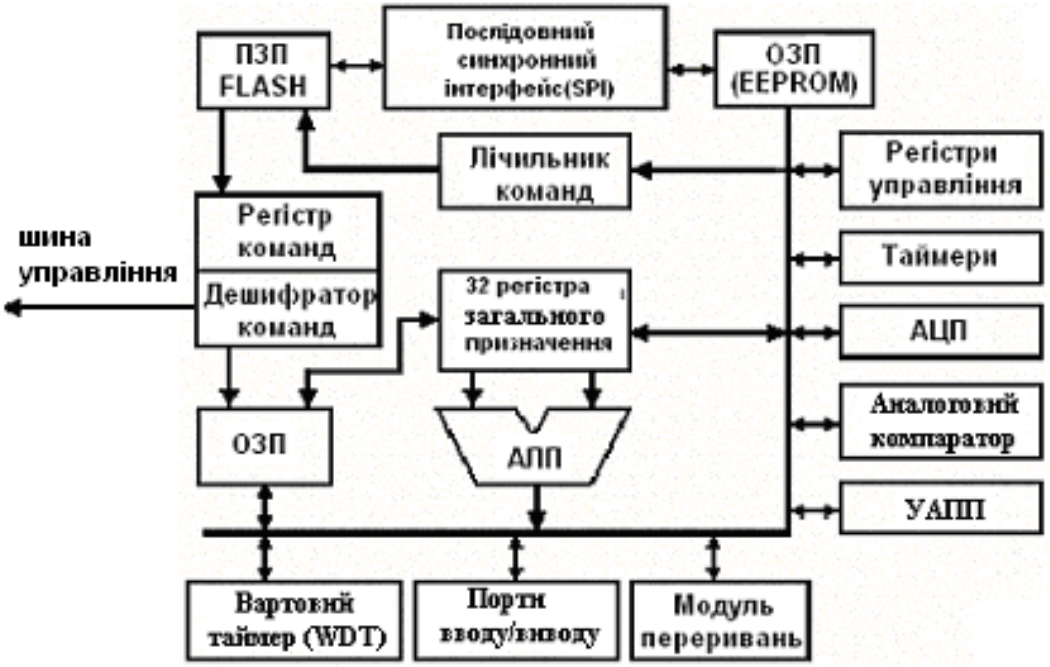

Рисунок 1.2 – Архітектура мікроконтролерів AVR

ведення і інтерфейсні схеми. Гарвардська архітектура AVR реалізує повний логічний і фізичний поділ не тільки адресних просторів, але й інформаційних шин для звертання до ROM і SRAM. Така побудова вже ближче до структури цифрових сигнальних процесорів і забезпечує істотне підвищення продуктивності (див. Рисунок 1.2). Використання однорівневого конвеєра в AVR також помітно скоротило цикл «вибірка - виконання» команди. Наприклад, у стандартних мікроконтролерів сім'ї MCS-51 коротка команда виконується за 12 тактів генератора (1 машинний цикл), протягом якого процесор послідовно зчитує код операції і виконує її. У мікроконтролерах AVR коротка команда в загальному потоці теж виконується за один машинний цикл, але він складає всього один період тактової частоти. Відмінною рисою архітектури AVR є регістровий файл швидкого доступу, що містить 32 байтових регістра загального призначення. Шість регістрів файлу можуть використовуватися як три 16-розрядних покажчика адреси при непрямій адресації даних (X, Y і Z Pointers), що істотно підвищує швидкість пересилання даних при роботі прикладної програми.

Flash-пам'ять програм AVR може бути завантажена як за допомогою звичайного програматора, так і за допомогою SPI-інтерфейсу, у тому числі безпосередньо на робочій платі - функція ISP. Останні версії кристалів "mega" випуску

2001-2002 року мають можливість самопрограмування (функція SPM). Усі AVR мають також блок енергонезалежної пам'яті даних EEPROM, доступний програмі мікроконтролера безпосередньо в ході її виконання. EEPROM звичайно використовується для збереження проміжних даних, констант, таблиць перекодувань, каліброваних коефіцієнтів і т.п. Ця пам'ять може бути завантажена ззовні як через SPI інтерфейс, так і за допомогою звичайного програматора. Два програмованих біти таємності дозволяють захистити ROM і енергонезалежну пам'ять даних EEPROM від несанкціонованого доступу. Внутрішня оперативна пам'ять SRAM є в AVR сім'ях "classic" і "mega", а також в одного представника "tiny" - ATtiny26/L. Для деяких мікроконтролерів можлива організація підключення зовнішньої пам'яті даних об'ємом до 64K (див. Рисунок 1.3).

Внутрішній тактовий генератор AVR може запускатися від зовнішнього генератора або кварцового резонатора, а також від внутрішнього або зовнішнього RC-ланцюга. Усі AVR цілком статичні, їх мінімальна робоча частота нічим не обмежена (аж до покрокового режиму). Мікроконтролер ATtiny15L має додатковий блок PLL для апаратного збільшення основної тактової частоти в 16 разів. При її номінальному значенні 1,6 Мгц одержувана допоміжна периферійна частота дорівнює 25,6 Мгц. Ця частота може служити джерелом для одного з таймерів/лічильників мікроконтролера, значно підвищуючи точність його роботи. Мікроконтролери ATmega64/103/128 також мають цікаву архітектурну особливість, що дозволяє значно знизити енергоспоживання кристала в цілому, коли в процесі роботи доцільно понизити основну тактову частоту мікросхеми. Спеціальний переддільник на кристалі дозволяє ділити основну частоту на ціле число в діапазоні від 2 до 129. Включення/виключення даної функції здійснюється програмно.

Мікроконтролери AVR мають від 1 до 4 таймерів/лічильників загального призначення з розрядністю 8 або 16 біт.

Загальні риси всіх таймерів/лічильників наступні:

– наявність програмованого переддільника вхідної частоти з різними градаціями ділення. Відмінною рисою є можливість роботи таймерів/лічильників на основній тактовій частоті мікроконтролера без попереднього її зниження, що помітно підвищує точність генерації часових інтервалів системи;

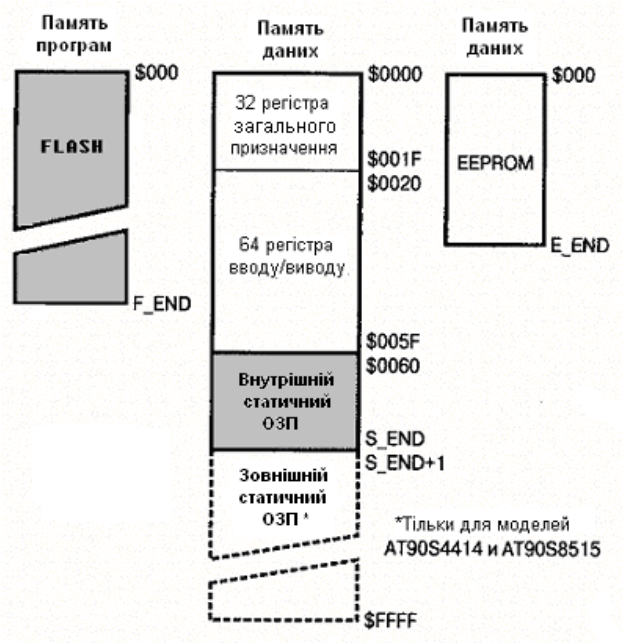

Рисунок 1.3 – Організація пам'яті мікроконтролерів AVR

– незалежне функціонування від режиму роботи процесорного ядра мікроконтролера (тобто вони можуть бути як зчитані, так і завантажені новим значенням у будь-який час);

– можливість роботи або від зовнішнього джерела опорної частоти, або як лічильник зовнішніх подій. Верхній частотний поріг визначений у цьому випадку як половина основної тактової частоти мікроконтролера. Вибір перепаду зовнішнього джерела (фронт або зріз) програмується користувачем;

– наявність різних векторів переривань для подій "переповнення вмісту", "захоплення", "порівняння".

Вартовий таймер у AVR має свій власний RC-генератор з частотою 1 МГц, яка є нестабільною і залежить від величини напруги живлення мікроконтролера і від температури. Вартовий таймер містить окремий програмований переддільник вхідної частоти, що дозволяє підлаштовувати часовий інтервал переповнення таймера і скидання мікроконтролера. Даний таймер можна програмно відключати під час роботи мікросхеми, як в активному режимі, так і в кожному з режимів зниженого енергоспоживання. В останньому випадку це приводить до значного зниження споживаного струму.

Система реального часу (RTC) реалізована у всіх мікроконтролерах "mega" і в двох кристалах "classic" - AT90(L)S8535. Таймер/лічильник RTC має окремий переддільник, що може бути програмним способом підключений або до джерела основної тактової частоти, або до додаткового асинхронного джерела опорної частоти (кварцовий резонатор або зовнішній синхросигнал). Для цієї мети зарезервовані два виводи мікросхеми. Внутрішній осцилятор, підключений до лічильного входу таймера/лічильника RTC, який оптимізован для роботи з зовнішнім "годинним" кварцовим резонатором 32,768 кГц.

Порти введення/виведення AVR мають число незалежних ліній "Вхід/Вихід" від 3 до 53. Вихідні драйвери забезпечують струменеву навантажувальну здатність 20 мА на лінію порту (вхідний струм) при максимальному значенні 40 мА, що дозволяє безпосередньо підключати до мікроконтролера світлодіоди і біполярні транзистори. Архітектура побудови портів введення/виведення AVR із трьома бітами контролю/управління (замість двох, як це зроблено в більшості 8-розрядних мікроконтролерів) дозволяє розробнику цілком контролювати процес введення/виведення, усуває необхідність мати копію вмісту порту в пам'яті для безпеки і підвищує швидкість роботи мікроконтролера при роботі з зовнішніми пристроями. Особливу значимість здобуває дана можливість AVR при реалізації систем, що працюють в умовах зовнішніх електричних завад.

Аналоговий компаратор входить до складу більшості AVR. Він має окремий вектор переривання в загальній системі переривань мікроконтролера. Тип перепаду, що викликає запит на переривання при спрацьовуванні компаратора, може бути запрограмований як фронт, зріз або переключення. Важливою апаратною особливістю є те, що логічний вихід компаратора може бути програмним чином підключений до входу одного з 16-розрядних таймерів/лічильників, що працює в режимі захоплення. Це дає можливість вимірювати тривалості аналогових сигналів, а також реалізовувати АЦП двотактного інтегрування.

Аналого-цифровий перетворювач побудований за схемою АЦП послідовного наближення з пристроєм вибірки/зберігання. Число незалежних каналів перетворення визначається типом мікро контролера. Розрядність АЦП складає 10 біт. Час перетворення вибирається програмно за допомогою установки коефіцієнта дільника частоти, що входить до складу блоку АЦП. Важливою особливістю аналого-цифрового перетворювача є функція придушення шуму при перетворенні, коли на точність не впливають завади, що виникають при роботі процесорного

ядра.

1.3.1 Відмінні риси

Основні особливості мікроконтролерів, наприклад, сім'ї Classic:

- можливість обчислень зі швидкістю до 1 MIPS/Мгц; FLASH-пам'ять програм об'ємом від 1 до 8 Кбайт (число циклів стирання/запису не менш 1000);
- пам'ять даних на основі статичного ОЗП (SRAM) об'ємом до 512 байт; пам'ять даних на основі ЕСППЗП (EEPROM) об'ємом від 64 до 512 байт (число циклів стирання/запису не менш 100000); можливість захисту від зчитування і модифікації пам'яті програм і даних (EEPROM);
- програмування в паралельному (з використанням программатора) або в послідовному (безпосередньо в системі через послідовний SPI-інтерфейс) режимах;
- різні способи синхронізації: вбудований RC-генератор, зовнішній сигнал синхронізації або зовнішній резонатор (п'єзокерамічний або кварцовий);
- наявність декількох режимів зниженого енергоспоживання.

1.3.2 Характеристики ядра контролера

Основними характеристиками центрального процесора мікроконтролерів розглянутої сім'ї є:

- цілком статична архітектура, мінімальна тактова частота дорівнює нулеві;
- АЛП підключений безпосередньо до регістрів загального призначення;
- більшість команд виконується за один машинний цикл;
- багаторівнева система переривань, підтримка черги переривань;
- від 3 до 16 джерел переривань (з них до 2 зовнішніх);
- наявність програмного стека.

1.3.3 Периферійні пристрої

Мікроконтролери сім'ї Classic мають досить розвинуту периферію. Набір периферійних пристроїв, що входять до складу того або іншого мікроконтролера,

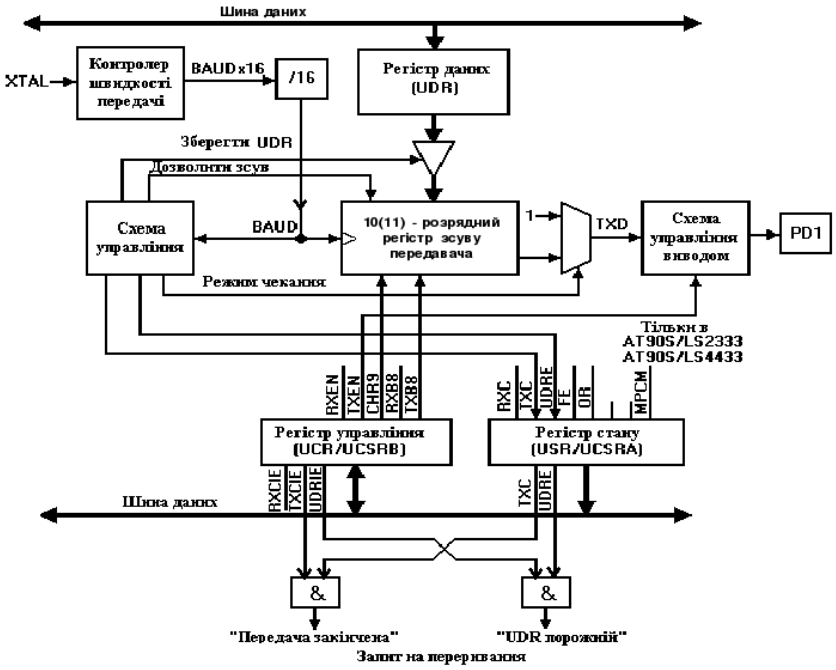

Рисунок 1.4 – Структура асинхронного передавача AVR

залежить від конкретної моделі. Перелічимо всі периферійні пристрої, які так чи інакше зустрічаються в мікроконтролерах сім'ї:

- 8-розрядний таймер/лічильник із переддільником (таймер T0);
- 16-розрядний таймер/лічильник із переддільником (таймер T1);
- 8-розрядний таймер/лічильник з можливістю роботи в асинхронному режимі (таймер T2);
- вартовий таймер (WDT);
- одно- або двоканальний 8..10-розрядний генератор сигналу із широтно-імпульсною модуляцією (ШІМ);
- одноканальний 8-розрядний генератор сигналу із ШІМ;
- аналоговий компаратор;
- 10-розрядний АЦП (6 або 8 каналів);
- універсальний асинхронний приймач-передавач (UART); див. Рисунок 1.4
- послідовний синхронний інтерфейс SPI.

1.3.4 Характеристики підсистеми введення/виведення

Основними характеристиками підсистеми введення/виведення є:

- програмне конфігурування і вибір портів введення/виведення;
- кожен вивід може бути запрограмований як вхідний або як вихідний незалежно від інших;
- вхідні буфери з тригером Шмітта на усіх виводах;
- можливість підключення до усіх входів внутрішніх резисторів, що підтягують, (опір резисторів складає 35... 120 кОм);
- навантажувальна здатність усіх виводів складає до 20 мА, що дозволяє безпосередньо управляти світлодіодними індикаторами.

1.4 Стенд EV8031/AVR

Навчально-налагаджувальний стенд «EV8031/AVR» (див. структурну схему на Рисунку 1.5) — програмно-апартний комплекс, орієнтований на використання в навчальних цілях по курсам програмування (Асемблер, C), а також середовище розробки програмного забезпечення на базі однокристальних контролерів архітектури AVR.

Зв'язок навчально-налагоджувального стенду «EV8031/AVR» відбувається через COM-порт. В комплект зі стендом постачається універсальний з'єднуючий кабель з двума гніздами (25 і 9 пінів) для з'єднання з одним з COM-портів.

Програмування мікроконтролера AVR відбувається через послідовний порт.

Технічні характеристики:

- використовується процесор ATmega8515
- пам'ять даних — 16 Кбайт
- послідовна EEPROM пам'ять, 256 байт
- два послідовні канали передачі даних RS-232
- інтерфейс розширення (16 ліній вихід, 8 ліній вхід/вихід)
- клавіатура 4x3

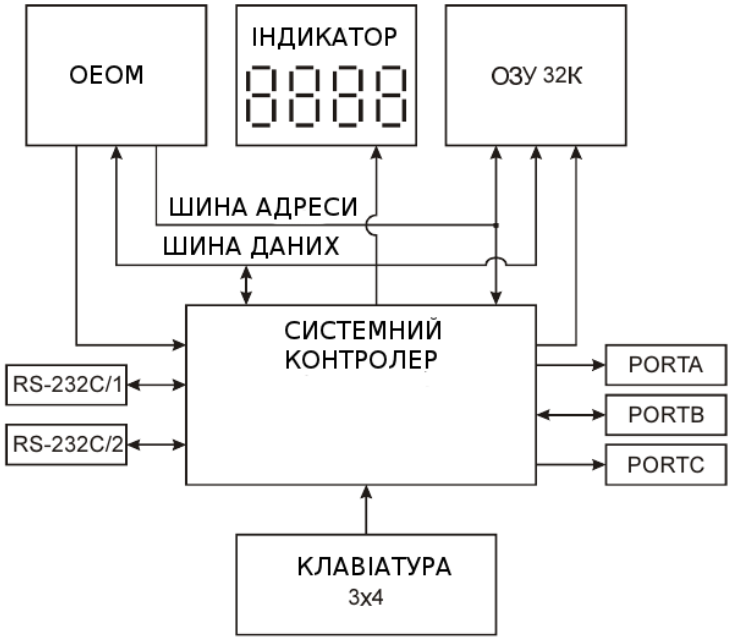

Рисунок 1.5 – Структура відлагоджувального стенду EV8031

- статична 4-розрядна семисегментна світлодіодна індикація
- цифроаналовий та аналогоцифровий перетворювачі (плата розширення)
- генератор з фіксованою частотою генерації — близько 10 КГц, генератор із змінною частотою генерації від 1 КГц до 50 КГц (плата розширення)
- динамічна 4-розрядна світлодіодна індикація (плата розширення)
- пристрій дискретного вводу інформації: 2 кнопки
- статична світлодіодна індикація, 8 шт.
- знакосинтезуючий світлодіодний індикатор 5x7 (плата розширення)
- рідкокристалічний дисплей
- динамік (плата розширення)

1.5 Обгрунтування вибору

Вибір Форту у якості програмного середовища спричинений необхідністю простого для реалізації доступу до електроніки на навчальному стенді EV8031/

AVR. Використання або реалізація окремої операційної системи все-рівно потребували би створення аналізатора команд і/або створення власної мови управління.

Іншою цікавою можливістю при використанні Форта є його гнучкість. Для цього було вибрано структуру з непрямим шитим кодом, адже вона забезпечує можливість виконання програм при гарвардській архітектурі пам'яті. Вибір архітектури також залежить від наявних ресурсів. Оскільки у мікроконтролера ATmega8515 всього 8 Кбайт пам'яті ПЗУ, а у стенда — 32 КБайт ОЗУ, то є зміст використовувати саме оперативну пам'ять для виконавчих процедур, а пам'ять постійну використовувати для збереження ядра. Непрямий шитий код ідеально підходить для таких задач.

Розглянутий вище AMforth не підходить для роботи по цій же причині. Він компілює нові слова у постійну пам'ять, що може відбуватись повільно , має обмеження у 4 Кслів, спричинює старіння пам'яті. Проте його модулі можуть згодитись для вивчення структури Форт ядра, а Форт модулі можуть прямо використовуватись для роботи на новому ядрі.

1.5.1 Стекові процесори нового покоління GreenArrays

У 2009 році американська компанія GreenArrays, Inc приступила до продажу сімейства багаточіпових процесорних систем GA4, GA32, GA144 з чотирма, 32-ма і 144-ма процесорами. Кожен процесор у складі системи має власний ПЗУ і набір регістрів. Набір команд організований у стековому виконанні.

Вузли пронумеровані в зеленій мати один або кілька загальних цілей введення/виводу. Ті, в жовтому є цифрові вводу/виводу з спеціалізованих конфігурацій, які можуть включати в себе загальні контакти і/або фантомні сигнали пробудження. Вузли пронумеровані в синій оснащені аналоговим ввід/вивід. Підписи під номери вузлів вказують ROM спеціалізації; червоні титри зарезервовані для п'яти вузлів, які підтримують завантаження чіпа після скидання. Виняток вузла двісті, який має особливе однопровідний послідовний отримати код в ROM, але не озброєним для завантаження.

Напрямки порт показані на кольорових барах розділяє вузлів. У невеликих кількостях у зовнішній бари являють вводу/виводу сигнал позначення, наприклад, сигнал 100, 17 (вузол 100 GPIO 17) пов'язане із двадцятим контактним процесором.

Використання у якості посередницької системи програмного середовища

Форт може допомогти при навчанні стековому програмуванню, адже середовище програмування для GA мікрокомп'ютерів F18A є стабільним, зрілим дизайном для комп'ютера і його введення/виводу яких має надійність було доведено в багатьох чіп конфігурацій. Це було доведено в 180 нм геометрії, і прототип в 130 нм також працював добре. Комп'ютер мало, вісім вписується в приблизно квадратний міліметр. Залежно від конфігурації чіпа, це дає між 100 000 і 200 000 комп'ютер на 8 дюймів пластини, сприяє низька вартість наших чіпів.

Швидкий, низькоенергетичний дизайн: наша прихильність до простоти підкріплюється засобами розробки, що сили наші інженери, щоб протистояти і вирішувати швидкість і енергія Витрати кожного елементу дизайну на кожній стадії макета та моделювання процесу. Ми постійно прагнути до мінімізації внутрішніх навантаження і схеми, завжди готові винаходити не тільки неефективно звичайної конструкції, але і будь-якій частині нашого власного. F18A є безтактний, повністю асинхронний комп'ютер, який може виконувати основні інструкції за 1.5 наносекунди. На одну команду витрачається порядку 7 пікоджоулів енергії.

Автономні ОЗП і стеки: кожен F18A містить 128 слів у пам'яті (До 512 команд) плюс 20 слів стека і регістрів, немає пам'яті вузьких місць. 18-бітові регістри S і T, двох верхніх елементів стека даних, R, верхній елемент про повернення стека, і, для читання і запису реєстру. Адресний регістр B має 9 біт, і лічильник P має 10.

1.5.2 Можливості використання

- Робототехніка
- маніпулятори/протези/автономно рухомі роботи;
- Нейронні мережі
- класифікація/розпізнавання сигналів/образів;
- «Бортові системи»
- діагностика стану в реальному часі/контроль руху;
- «Академічні» системи — апаратне забезпечення курсів цифрової обробки сигналів, паралельного програмування, архітектури обчислювальних систем;

– «Персональні» обчислювальні системи — розширення ПК/планшетників/«гаджетів»;

– Java/Lisp/Prolog - машини;

– Розпізнавання/синтез мови;

– Управління антенними системами (ЦАР, ФАР);

– Модулятори/демодулятори сигналів.

2 Будова форт-ядра

Традиційна форт-система складається з двох частин: виконуюче середовище (її часто назвают VFM — віртуальна форт-машина) та транслятор. Транслятор займається поглинанням вихідних текстів програм, VFM займається виконанням програмного коду.

Проте дві частини можуть працювати і незалежно. VFM без транслятора може використовуватися для виконання компільованою раніше програми. Якщо ця програма в процесі виконання сама нічого не транслює, то транслятор їй не потрібен і може бути відсутнім. Багато прикладних програм на Форті можуть ставитися до цього класу — використовуватися без транслятора Форту, тільки з VFM.

Транслятор без VFM використовується, наприклад, при цільовій компіляції — створення програми для іншої платформи — коли цільова VFM просто не може працювати на інструментальній VFM. У цьому разі цільової компілятор просто створює код для цільової платформи, виконувати його буде вже інша VFM на іншому комп'ютері.

Прийнято вважати, що відмінні риси віртуальної машини Форту — наявність двох стеків, шитий код і адресний інтерпретатор. Насправді перше — зручний варіант реалізації прийнятого в Форті способу передачі параметрів, друге і третє — просто окремий випадок способу кодогенерації і його виконання. Форт може компілювати звичайний машинний код, що не вимагає адресної інтерпретації, може компілювати байт-код, вимагає інтерпретації, але не адресною, і т.п., і при цьому продовжувати залишатися Фортом.

Абсолютно переважна більшість мов програмування використовують стек для передачі параметрів і для зберігання адрес повернень з підпрограм (функцій, процедур). Більшість роблять це неявним чином — програміст не оперує з поняттям «стек». Але їх виконують середовища — практично завжди використовують стек. Це дуже зручна конструкція для зберігання точок повернення при виклику багаторазово вкладені підпрограм, для передачі параметрів і для зберігання тимчасових локальних змінних.

Але більшості мов вистачає одного стека. Форт — представник досить нечисленного класу процедурних мов. Більшість мов оперує з функціями — видом

підпрограм, повертають одне значення в якості результату. Повернути кілька значень функція може тільки або записуючи їх у змінні, чиї адреси передаються в якості параметрів, або повертаючи в якості результату покажчик на структуру, що містить кілька значень. При виході з функції дані на стеку, використовувані при роботі цією функцією, можуть бути безболісно зняті з стека. І адреса повернення теж. На стеку або нічого не повертається (зазвичай результат функції при поверненні зберігається в регістрі процесора), або повертається відоме число елементів — один елемент. Процедура Форту може використовувати аналогічні способи, але може і повернути декілька значень на тому ж стеку, на якому передавалися параметри (це плюс, а не мінус, але плюси пізніше). І якщо при виклику процедури поміщати адресу повернення в стек над параметрами процедури (як робиться у функціональних мовами), то доведеться приймати спеціальні заходи, щоб витягти цю адресу з під повертаються значень і ще й "згуртувати" дані на стеку для ліквідації "дірки" від адреси повернення. Можна придумати різні способи для вирішення цієї проблеми, але хтось колись вирішив цю проблему з перемішуванням даних і адрес повернень на одному стеку просто не створювати - і так з'явилися два стеки: один для параметрів процедур і повертаються значень, інший для адрес повернення. Це зручно і досить ефективно, тому широко застосовується в Форті.

Однак Форт міг би існувати і з одним стеком, і з трьома, це для мови не принципово. На жаль, у Форті утвердилася практика використання другого стека (стека повернень) не тільки за прямим призначенням автоматично при викликах/поверненнях з процедур, але і явних ручних маніпуляцій з цим стеком для зберігання безіменних локальних і тимчасових змінних і навіть для ручного втручання в хід повернень з процедур (що порушує принципи структурного програмування). З іншого боку, форт не змушує цим користуватися, так що і боротися з цим явищем не потрібно. Я просто намагався показати, що наявність двох стеків не є невід'ємною частиною Форту і його відмінною рисою.

Що ж тоді є відмінною рисою віртуальної машини Форту? По-моєму, у неї немає зовнішніх відмінних рис, крім того факту, що Форт використовує не функції, а процедури. Внутрішні ж структури — стеки, способи компіляції і виконання коду, і т.д. — залежать від конкретної реалізації. Інших якихось особливих рис, типу «загальної спискофікації» Ліспу або «загальної об'єктизацією» Смолтолка з автоматичними збирачами сміття у Форт-машині немає. Форт-машина

дуже близька до звичайних процесорів і тому гранично проста в реалізації.

2.1 Шитий код

Основою Форта є техніка адресної інтерпретації «шитий код». При шитому коді ланцюжки адрес підпрограм складаються послідовно («зшиваються») і формують нову підпрограму. Адресний інтепретатор згодом проходить по такому ланцюжку, послідовно запускаючи підпрограму по зчитаній адресі. За багато років з'явилось багато варіацій шитого коду і вибір конкретної залежить від багатьох обставин: типу процесора, кількості пам'яті, потрібної швидкодії. Для правильного вибору потрібно знати, які бувають види ШК і як вони утворюються.

2.1.1 Непрямий шитий код (Рисунок 2.1)

Класичний тип шитого коду для Форту, описана у більшості книгах. Всі інші техніки є покращеннями даної.

Розглянемо приклад Форт-визначення слова SQUARE:

```
1 : SQUARE DUP * ;
```

В типовому Форті, основаному на непрямому ШК, код буде розміщений в пам'яті так, як показано на рисунку 2.1. Вказівник інтепретації (IP) повинен вказувати на комірку в пам'яті, що зберігається всередині цього «іншого» слова, котре зберігає адресу слова SQUARE. Інтепретатор отримує цю адресу і використовує її для отримання вмісту поля коду слова SQUARE. Вміст даної комірки являє собою адресу машинної підпрограмми, котра виконує слово SQUARE.

Якщо SQUARE написано в машинному коді, тоді завдання інтепретатора на цьому завершується. Код би виконався і потім сам повернув управління адресному інтепретатору. Проте, якщо SQUARE — високорівневе слово (визначене через двокрапку), воно містить не машинний код, а список адрес. Для виконання даного слова інтепретатор повинен перезапуститись на новий потік адрес — поле параметрів слова SQUARE. Для цього у полі коду знаходиться адреса так званої процедури ENTER (або DOCOLON), яка зберігає у стек старе значення IP і записує у IP нове значенн — адресу поля параметрів. Для повернення у попереднє слово використовується підпрограма EXIT, котра просто відновлює зі стеку попередню адресу IP (до входу в слово SQUARE).

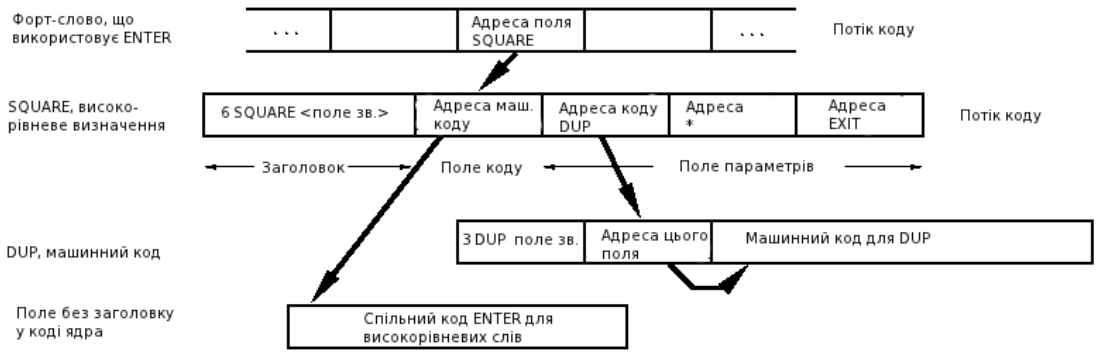

Рисунок 2.1 – Непрямий шитий код

Переваги шитого кода:

- одна комірка (2 байта) на адресу
- можливість розміщення коду у оперативній пам'яті (виконання коду при Гарвардській архітектурі пам'яті)
- класична схема
- переносимість

Недоліки:

- подвійний рівень непрямого адресування, внаслідок цього зменшена швидкодія

2.1.2 Прямий шитий код (Рисунок 2.2)

Прямий ШК відрізняється від непрямого тільки наявністю машинного коду в полі параметрів (замість вказівника на адресу). Фактично, комірка містить команду процесора CALL, яка і викликає потрібний машинний код для слова.

Переваги:

- виграш у швидкості роботи перед непрямим ШК — на третину зменшене слово NEXT

Недоліки:

- збільшено на один-два байти поле коду

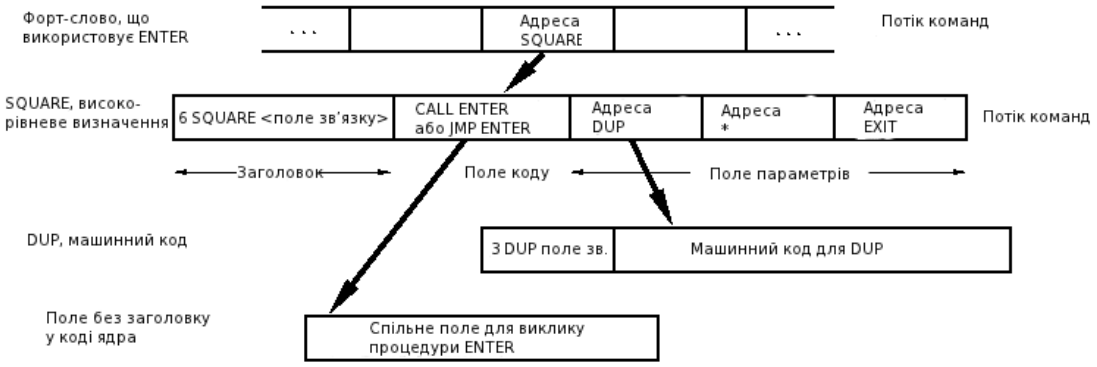

Рисунок 2.2 – Прямий шитий код

– код потрібно розміщувати у області пам'яті, доступної на виконання (потрібна Фон-Нейманівська архітектура пам'яті)

2.1.3 Підпрограмний шитий код (Рисунок 2.3)

Підпрограмний шитий код відрізняється відсутністю вираженого адресного інтепретатора, його роль виконують команди процесора CALL і RET, а показник поточної адреси IP співпадає з регістром IP процесора. Поле параметрів складається з підпрограмних викликів машинних процедур і, відповідно, високорівневі слова також являються машинними процедурами.

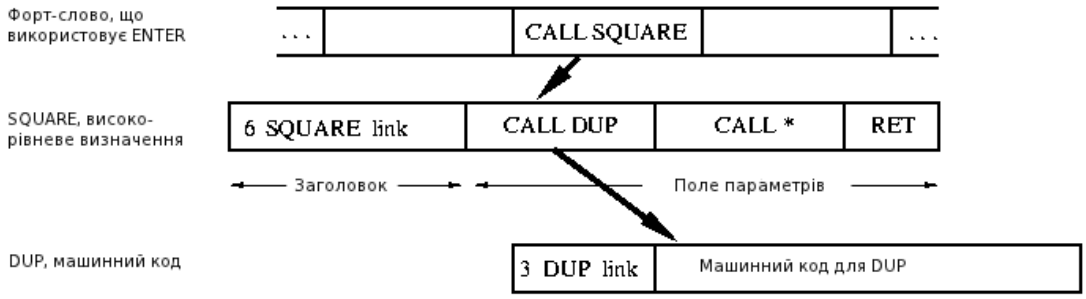

Рисунок 2.3 – Підпрограмний шитий код

Переваги:

– найвища швидкодія

– єдиність механізму роботи процесора і адресного інтепретатора

– зменшено кількість використовуваних регістрів

– широкий діапазон для оптимізації типу інлайн-вставок

Недоліки:

– складність декомпіляції коду

– збільшений розмір

– неможливість динамічного додавання нових слів у Гарвардській архітектурі пам'яті (при відсутності доступу до перезапису кодової пам'яті)

2.1.4 Згорнутий шитий код (Рисунок 2.4)

Згорнутий (або токенізований) шитий код придуманий для зменшення розміру високорівневих визначень і є різновидом непрямого. Поле параметрів є списком однобайтних адрес на таблицю слів. В самій же таблиці слів знаходяться реальні, двобайтні, адреси слів. Тобто, додано ще один рівень непрямого адресування.

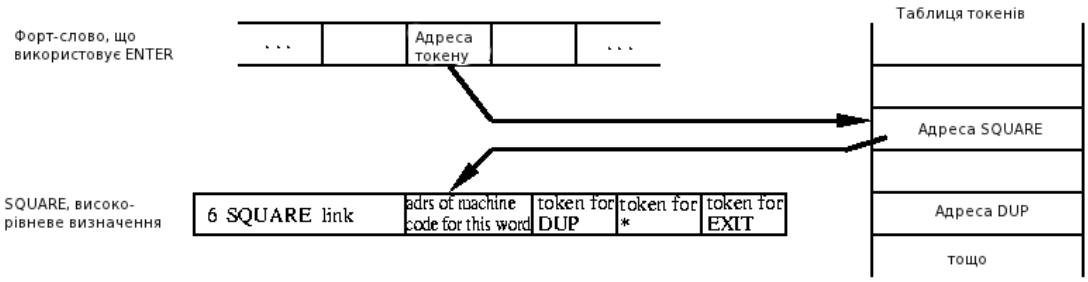

Рисунок 2.4 – Згорнутий шитий код

Переваги:

– найменший можливий розмір коду високорівневих слів

Недоліки:

– обмеження у 256 слів

– потрібна таблиця розміром щонайменше 512 байт

– повільна інтепретація

2.2 Регістри і стеки

Регістрова адресація використовується дуже часто при програмуванні мікроконтролерів. Проте Форт не використовує її для виконання задач. Для арифметичних операцій Форт використовує нуль-операндну систему командних слів. Передача числових параметрів відбувається неявно, через стек.

2.2.1 Стек даних

Основною абстракцією при роботі з даними у Форті є стек даних. Стек даних — програмно або апаратно реалізований список, що працює по принципу «перший ввійшов — останній вийшов». Через стек даних відбувається передача параметрів слів-функцій. Основні параметри стеку даних:

- розрядність
- глибина
- реалізація - програмна чи процесорна
- розміщення

2.2.2 Стек повернень

Стек повернень — стек, що використовується адресним інтерпретатором для збереження/відновлення точки входу у високорівневому слові. Арифметичні операції зі стеком повернень як правило не працюють, проте доступ до нього не закривається у більшості реалізацій Форта. Це дозволяє використовувати стек повернень для тимчасового збереження даних, проте може призвести до порушення цілісності адресної інтерпретації. Стек повернень найчастіше реалізовується на процесорних операціях PUSH і POP. Тому розрядність його майже завжди дорівнює розрядності адреси, а глибина обмежена тільки кількістю вільної пам'яті.

2.2.3 Додаткові стеки

Для зручної роботи також використовуються наступні стеки:

- циклів
- контекстів
- стек дробових чисел

Проте вони не є обов'язковими і в простих реалізаціях можуть бути відсутні.

2.2.4 Використання регістрів

За рахунок нуль-операндних функцій, для реалізації Форта потрібно всього-лиш 3 регістра для повноцінної роботи. Цей приклад показує, що єдина хороша функція регістрів процесора — кешування системних змінних Форта. Індексні регістри відходять під стек даних і IP, інші відходять під робочий регістр W, ще один під верхні один-два елементи стеку даних. Всі інші можуть використовуватись як тимчасові для асемблерних процедур.

Очевидно, розміщення регістрів індивідуальне для кожного сімейства мікроконтролерів. Акумуляторні процесори містять небагато індексних регістрів, тому реалізація Форт інтерпретатора на них супроводжується постійним збереженням/відновленням вказіників IP і DSP.

Стек — це структура, в яку «останнім увійшов, першим вийшов», тобто як магазин автомата Калашникова, пристрій для відкладання чогось на потім і діставання у зворотному порядку. Де ми в наших прикладах відкладали операнди і операції і виконували їх потім у зворотному порядку? Правильно, в прикладі 5 програмі на C. Саме стек використовується транслятором C для розгортання запису виду N = N +5 в послідовність машинних команд «Взяти адресу N, витягти значення, взяти 5, скласти їх, взяти адресу N і записати туди суму», так як тільки таку лінійну послідовність команд може виконати процесор. Тобто програміст, маючи на увазі цю послідовність команд, записує її на C задом наперед, потім транслятор C, розгортає її в пряму послідовність команд і в цьому вигляді вона може бути виконана процесором.

Стек і зворотний запис в наявності. У Форті у всіх прикладах послідовність одна і та ж — пряма, ніякого забігання вперед для обчислення відсутніх операндів. Кожне слово працює вже з обчисленими раніше операндами. І саме як тимчасове сховище обчислених раніше операндів і використовується стек у Форте. У прикладі 3 на стеку Форту зберігалося два операнди — 2 і 3, і слово «+» брало їх звідти. У тому ж прикладі на C при обчисленнях на стеку виявляються не тільки відкладені операнди, але і відкладена операція. Як би там не було — результат один — лінійний машинний код. При виконання цього коду стек може бути присутнім, а може й не бути в обох мовах. Наприклад, якщо операнди в C з плаваючою крапкою, то вони обидва при виконанні потраплять на стек співпроцесора x87 та операції над ними виробляться там в точності також, як над операндами Форту на внутрішньому стеку Форт-процесора або програмної FVM.

Це залежні від реалізації малоістотні деталі. Стеки мають місце у всіх сучасних мовах програмування. Форт може бути «більш стековий» тільки за рахунок того, що у нього є багато операцій для маніпуляції цим стеком в явному вигляді, тоді як в інших мовах цей стек для програміста доступний тільки як контейнер для параметрів і локальних змінних, з можливістю звернення до них тільки по іменах. У Форте теж можна використовувати іменовані локальні змінні, але можна обходитися і без них.

У С ми маємо суміш префіксного, інфіксного і постфіксного типу запису. Крім того, не обійтися без дужок і роздільників виразів «;». Дужки групують операнди та операції — тобто кажуть транслятору я не хочу виконувати операції по порядку, а хочу задом наперед і упереміш. Крапка з комою говорить транлятору що пора зупиниться в забіганні вперед і пора піти по стеку відкладених операцій і операндів тому для розгортання «зворотної американської запису» в лінійну послідовність команд для комп'ютерів Фон-неймановской архітектури. У Форте у всіх прикладах один і той же примітивний порядок запису - пряма послідовність дій. Дужки там не потрібні - якщо потрібна інша послідовність виконання команд, то в Форте це можна записати у вигляді саме цієї іншій послідовності.

Дужки у Форте використовуються для коментування. І обмежувач пропозицій не потрібен, тому що не потрібно повертатися тому. На кожний момент часу все що було «раніше» — вже виконано і повертатися нема чого. Крапка з комою використовується у Форті тільки в Наприкінці визначення процедури (до речі, це даремно, можна теж не обмежувати, але про це пізніше).

Система запису Форту може здатися складним тільки на перший поверхневий погляд. Насправді синтаксис Форту ви вже вивчили - слова та прогалини, і нічого більше. А згадайте скільки років ви вивчали алгебраїчну форму запису, на якій заснований мова С. Форт — синтаксично найпростіший мову. І якщо десь зустрічаються складності, то справа не в Форт-ідеології, а в реалізації конкретних слів. Набір слів стандартного Форту дійсно дуже незвичайний і хаотичний. Причина — хаотичний розвиток протягом 30 років. Неоптимальність мовних конструкцій — хвороба більшості мов, особливо таких старих як Форт. Але Форт гнучкий, і ви можете не використовувати ті його кошти, які вам не сподобаються. Більше того, ви можете ті кошти мови, які вважаєте за потрібне - приклади будуть в цій статті.

2.3 Слова

Слово — основна абстракція в Форті. Її аналог в інших мовах програмування — функція або процедура. Слова Форта організуються у словник — однозв'язний список з так званих «словникових статтей». Словникова стаття складається з заголовку і коду слова. Заголовок використовується для пошуку потрібного слова у словнику, а код (поле параметрів) проганяється через адресний інтепретатор.

Існує певний набір примітивів — слів, які рекомендовані для реалізації будь-яким Фортом, проте даний набір не є чимось постійним і може як завгодно модифікуватись програмістом під свої потреби.

Слова можуть реалізовуватись на асемблері, а можуть складатись з інших Форт-слів. Останні в свою чергу називаються високорівневими словами.

Рядок з слів, розділених прогаликом, називається Форт-програмою, і така Форт-програма може виконуватись на Форт-інтепретаторі.

2.3.1 Форт-інтепретатор

Форт-інтепретатор є прикладом організації одного з найпростіших методів текстового управління потоком команд процесора. Він забезпечує ввід команд від користувача і ввід нових програм. Разом з адресним інтепретатором він утворює середовище програмування, яке надзвичайно просте для реалізації і при цьому залишається достатньо потужним.

Форт відноситься до конкатенативних мов програмування. Конкатенативні програми утворюються шляхом долучення управляючих слів до основної програми і вони можуть виконуватись паралельно з отриманням коду. Конкатенативні мови як правило мають елементарний синтаксис. У випадку Форта — слова розділені прогаликом.

При інтепретації відбуваються наступні дії:

а) З вхідного потоку видаляються розділюючі символи

б) Виділяється слово (обмежене розділюючими символами)

в) Виконується пошук слова у словнику по принципу зв'язаного списку. Якщо не знайдено, то видається помилка. Якщо знайдено, то адреса поля коду кладеться на стек даних

г) Отримана адреса передається слову EXECUTE (в режимі інтепретації) або слову COMPILE (в режимі компіляції).

д) Повторити з пункту а)

2.3.2 Словник

Для описаного вище циклу транслятора, не вистачає реалізації слова TranslateWord, яке повинно проводити пошук слів у словнику і виконувати або компілювати слово. Якщо пошук завершився невдачею — спробувати використовувати слово як літерал.

Процедури Форту — слова — зберігаються в лінійних списках — словниках. Слова при визначенні додаються в кінець списку, і пошук починається з кінця списку, щоб дозволити перевизначення слів. Словників може бути кілька, і пошук слова транслятором виробляється в кількох словниках. Поточний набір словників, які підлягають перегляду, та порядок пошуку задається у спеціальному стеку словників. Цей стек називають контекстом.

Стандартне слово Форту для пошуку в контексті — FIND, однак воно не зовсім зручно, тому що використовує рядок з лічильником як параметр, і повертаються їм значень для деяких застосувань недостатньо. Слово SEARCH-WORDLIST шукає тільки в одному словнику, і теж повертає неповний набір результатів, тому ми реалізуємо свій набір слів для пошуку.

Словники Форту (vocabulary) фактично є пов'язаними списками пар ключ-значення і є близькими аналогами асоціативних масивів, словників (dictionary) і т.п. структур, що є в багатьох інших мовах програмування. Точніше майже у всіх мовах, так як без такої зручної структури обійтися складно. З широко відомих мов словники або їх інші назви є в Perl, PHP, PostScript, Smalltalk, Lisp і т.д.), так що поширена думка про особливу унікальності словників Форту на мій погляд невірно. Хоча є відмінності в реалізації. Наприклад, прапори у елементів словника зустрічаються в інших мовах нечасто (я знаю тільки в PostScript), з іншого боку ці прапори можна «емулювати» і в інших мовах.

Відповідно до ANS Forth94, словники офіційно тепер називаються WORDLIST — список слів.

Словник можна розглядати як окремий випадок більш простої структури — списку. Кожен елемент списку має вміст і покажчик на наступний елемент. У випадку словника вмістом елемента списку буде пара покажчиків — на ім'я та значення елемента словника. Якщо елемент словника — процедура, то ім'я — це ім'я процедури, а значення — адреса коду процедури (не обов'язково реальну

адресу, а те що в стандарті називається execution token, тобто якесь посилання, яку може виконати слово EXECUTE).

2.3.3 Компіляція і інтерпретація

Однією з особливостей роботи Форта є робота у двух режимах — інтепретації та компіляції. При інтепретації слова з вхідного потоку запускаються на миттєве виконання. При компіляції, слова із вхідного потоку перетворюються у адреси поля коду і дописуються до словника. Таким чином, можна скомпілювати нове високорівневе слово і згодом викликати його на виконання.

Компільовані слова мають переваги. Вони виконуються швидше і дозволяють умовні переходи/цикли.

В режимі інтепретації можна створювати змінні і організовувати задачі. Також тимчасовий режим інтерпретації використовується для оптимізації компіляції літералів.

2.4 Багатозадачність

Багатозадачність в Форті найпростіше організувати за допомогою сопроцедур.

Кооперативна багатозадачність — це вид багатозадачності, при якій кожна процедура сама вказує місце, де її можна перервати і передати управління іншій задачі, без втрати важливих даних. Даний метод забезпечує надійне псевдопаралельне виконання програм, проте потребує спеціального проектування програми.

Переваги кооперативної багатозадачності у відсутності необхідності захизати усі структури даних об'єктами типу критичних секцій і м'ютексів, що спрощує програмування і перенесення коду з однозадачних у багатозадачні.

Недоліками є неможливість роботи системи у випадку помилки в одній з процедур: не відбувається передача керування процесорним часом. Ускладнена робота з вводом-виводом.

3 Проектування системи

Архітектура Форт системи вибрана по класичному представленню. Серед набору шитих кодів вибрано саме непрямий, завдяки економному використанню пам'яті і можливості виконання коду з ОЗП.

При подачі живлення відбуваються наступні кроки:

- ініціалізація стеку і зовнішньої пам'яті
- копіювання початкової Форт-прошивки з ПЗП до ОЗП
- заповнення системних змінних в ОЗП
- перехід до очікування вводу програми по УАПП

Такий підхід дозволяє спростити алгоритми, що використовуються у програмі, до мінімуму і швидше приступити до написання важливих слів.

В якості стандарту вибрано стандарт 1984 року в зв'язку з відсутністю потреби у обчисленнях з плаваючою комою.

AVR призначене для функцій цифрової обробки даних, контролю периферії. Також компанією заявляється висока щільність коду (компактність), що в принципі підтверджується тестами та дослідженнями. AVR32 має шістнадцять регістрів, об'єднаних в регістровий файл (регістри R0-R15). Варто відзначити, що покажчик стека (SP), програмний лічильник (PC) і регістр зв'язку (LR) відображаються в регістровому файлі — регістри R13, R15 і R14 відповідно. Можливе виконання інструкцій, таких як, додавання і віднімання з використанням SP, PC і LR регістрів, що призводить до більш ефективної адресації пам'яті. Дані регістри можуть використовуватися в якості операнда джерела або приймача (регістру призначення) у всіх інструкціях, які використовують реєстрові операнди, включаючи арифметичні або логічні інструкції та інструкції завантаження/збереження.

Інструкції, які використовують PC як приймач, слід розглядати як інструкції переходу. Це має на увазі, що очищається конвеєр і виконання поновлюється з адреси, обумовленим новим значенням PC. Регістр R12 призначений для повернення значення з функцій виклику, а так само виступає як приховане значення,

що повертається команд переміщення і тестування. Багатоадресні команди завантаження і витягання з стека мають ті ж самі функціональні можливості, які дозволяють їм використовуватися як інструкції повернення.

Покажчик стека також неявно використовується деякими інструкціями. У всіх режимах регістр загального призначення R14 використовується як регістр зв'язків (LR). Він зберігає адресу повернення з програми. Коли підпрограма виклику виконується різновидом команд call LR запам'ятовує адресу повернення з програми. Повернення з підпрограми відбувається при копіюванні LR в PC однією з різновидом команди mov таких як ldm, popm або ret. У всіх інших випадках регістр зв'язків R14 можна використовувати як регістр загального призначення. Для роботи з числами подвійної точності використовуються реєстрові пари R0-R1, R2-R3 і т.д. Ортогональний набір команд ядра AVR32 дозволяє всі регістри використовувати як покажчики.

Процесор має набір інструкцій для цифрової обробки сигналів: множення з накопиченням — MAC, команди SIMD та інструкції підтримки мови JAVA. Архітектура AVR32 визначає різні мікроархітектури, що мають відмінні характеристики за величиною витрат і збереження енергії, складу регістрів і порядок опрацювання переривань і виняткових ситуацій.

3.1 Створення програмного середовища

Вибір програмного середовища є важливим кроком при розробці програмного проекту. Цей вибір залежить від великої кількості факторів і індивідуальний для кожного проекту.

Для виконання даної роботи було вибрано мову програмування асемблер. Вибір впав не в поле зору C через наступні причини:

а) Форт має достатньо просту і прозору структуру, щоб його можна було без великих затрат реалізовувати на асемблері

б) при програмуванні віртуальної машини важливе точне використання конкретних регістрів; автоматичне розприділення пам'яті може призвести до псування інформації у системних регістрах

в) зберігання системних регістрів у пам'яті призводить до заповільнення роботи

В якості інтегрованої системи розробки було вибрано редактор geany. Даний редактор дозволяє без додаткових налаштувань виконувати скриптові файли, додає підсвітку коду, проводить початковий аналіз коду.

З урахуванням можливостей різних варіантів архітектур (AVR32A, AVR32B), «системні» регістри форт-машини розташовуються в регістрах з R8 по R15. Основне завдання при специфікації функцій регістрів - знайти оптимальний розподіл регістрів, мінімізувати кількість проміжних операцій при моделюванні роботи стекової машини. До розгляду пропонується модель FVM з кешуванням верхніх елементів стеків повернення і даних. Дана модель дозволяє зберігати основні регістри форт системи при виникненні переривань для будь-якої мікроархітектури процесора. Дозволяє за рахунок двох тимчасових регістрів кешувати дані стека для здійснення арифметико-логічних операцій, як одинарної, так і подвійної точності, тимчасово зберігаючи дані в реєстрових парах. При створенні ядра віртуальної форт-машини були розглянуті набори слів, що реалізовуються в різних системах на низькому рівні.

Регістр позначення функція в FVM PC (R15) PC програмний лічильник LR (R14) R0 вершина стека повернень SP (R13) RP покажчик стека повернень R12 B індексний регістр/регістр тимчасового зберігання даних R11 S/A індексний регістр/регістр тимчасового зберігання даних/другий елемент стека даних R10 T вершина стека даних R9 SP покажчик стека даних R8 U покажчик користувацької області R7-R0 відведені під локальні змінні ACBA базовий регістр виклику підпрограм/функцій. Може бути використаний, як покажчик поточного словника системи JAVA_LVx регістри локальних змінних. В тому випадку, якщо є можливість використовувати їх в RISC режимі

3.2 Реалізація віртуальної машини і базового набору слів

Віртуальна машина створена по класичному методу. В її основі лежать три процедури: NEXT, ENTER і EXIT. Кожна з них виконує свої важливі функції. Приведемо лістинги даних процедур на мові програмування абстрактного асемблера.

```
1  NEXT:
2    ld Wl, IP+
3    ld Wh, IP+
4    ld ZL, W+
5    ld ZH, W+
6    ijmp
7
8  ENTER:
9    push IPl
10   push IPh
11   movw IP, W
12   rjmp NEXT
13
14 EXIT:
15   pop IPh
16   pop IPl
17   rjmp NEXT
```

Проведемо аналіз даних процедур.

3.2.1 Процедура NEXT

Процедура NEXT відповідає за перехід до наступного слова у потоці адрес. При своїй роботі вона завантажує поточну адресу у робочий регістр (W). В робочому тепер знаходиться адреса поля коду слова. Після цього відбувається отримання адреси виконавчого коду в ПЗП мікроконтролера. Записавши дану адресу в регістр Z, процедура здійснює непрямий перехід в програмі.

Таким чином, вдається виконати асемблерну процедуру.

Після завершення машинного коду, програма повинна виконати черговий стрибок на процедуру NEXT, щоб віртуальна машина змогла продовжити свою циркуляцію по потоку адрес.

При реалізації цих основних процедур постає питання вирішення проблеми розподілення регістрів. В даній роботі дана проблема була вирішена наступним чином:

- регістр X утримує вказівник на поточну адресу IP
- регістр Y зберігає робочий регістр
- регістр Z не використовується системно. Дане рішення спрощує дизайн додаткових процедур, проте втрачається можливість оптимізаційної техніки «вершина стеку»

3.2.2 Процедура ENTER

Процедура ENTER забезпечує коректний вхід і виконання високорівневого визначення слова. При роботі вона зберігає поточне значення вказівника поточної адреси IP у стек повернень. Після даного збереження вона завантажує вміст робочого регістра до вказівника поточної адреси. Дана дія основана на тому, що робочий регістр зберігає вказує на машинну адресу тільки в одному випадку — при її зчитуванні у процедурі NEXT. Увесь інший час є вказівником на наступну адресу. Завдяки архітектурі слова, поле параметрів йде одразу після поля коду, що означає — робочий регістр автоматично є вказівником на поле параметрів, а отже його можна використовувати як вказівник на новий потік адрес.

Процедура ENTER використовується для високорівневих визначень слів. Саме адреса процедури ENTER знаходиться у полі коду для високорівневих визначень слів. Для порівняння можна привести два приклади слів: машинного і високорівневого.

```
1  mem_AND:   .dw itc_AND      ; асемблерне визначення
2
3  itc_AND:
4    rcall ds_POP_ZZ
5    mov temp2L, ZZL
6    mov temp2H, ZZH
7    rcall ds_POP_ZZ
8    and ZZL, temp2L
9    and ZZH, temp2H
10   rcall ds_PUSH_ZZ
11   rjmp NEXT
```

Високорівневе визначення

```
1  mem_CELL:   .dw ENTER
2    .dw (mem_LIT - START_CODE)*2 + WORDS_START
3    .dw 2
4    .dw (mem_EXIT - START_CODE)*2 + WORDS_START
```

Дані приклади показують типове розміщення адресного потоку для слів Форту.

3.2.3 Процедура EXIT

Процедура EXIT є напростішою з основних. Вона виконує відновлення вказівника поточної адреси, по такому ж принципу, що і асемблерна команда

ret. Дана процедура ніколи не вказується прямим посиланням, як наприклад, ENTER. Натомість її адреса завжди компілюється через проміжний рівень непрямої адресації. Саме тому, для неї потрібне і асемблерне слово. Останнє може бути безіменним.

EXIT і ENTER працюють із стеком повернень і забезпечують ієрархічне виконання потоку команд. Дві інші важливі процедури працюють із стеком даних.

3.2.4 Процедури ds_PUSH_ZZ і ds_POP_ZZ

Процедури ds_PUSH_ZZ і ds_POP_ZZ забезпечують операції зі стеком даних для додаткових процедур. Від ефективності реалізації даних процедур залежить ефективність арифметичних операцій. Враховуючи, що арифметичні операції складають більшу частину коду програми, часто використовується техніка «вершина стеку». При даній техніці комірка, що знаходиться у вершині стеку, кешується в регістри процесора.

Дана реалізація побудована на простішому механізмі штучних операцій із стеком даних і демонструє іншу властивість Форта — зберігання системних змінних у ОЗП. При кожному запиті до процедури відбувається зчитування комірки по адресі в ОЗП і відповідний запис, що означає зсув вершини стеку даних. У випадку ds_PUSH_ZZ зсув відбувається у сторону вищих адрес, у випадку ds_POP_ZZ зсув відбувається у сторону менших адрес.

3.3 Реалізація стеків і арифметика

Всі арифметичні операції і багато логічних повинні використовувати акумулятор. Є тільки одна 16-бітна операція INC DPTR. Апаратний стек повинен використовувати 128-байтний накристальний файловий регістр.

Деякі AVR Форти використовують 16-бітну модель, але вони дуже некваліфіковані. Давайте зробимо деякі компроміси і зробимо більш швидкий Форт для AVR процесора.

У нас є тільки один адресний регістр. Тому давайте використовувати в якості лічильника команд рідної регістр IP 8051 процесора, і виберемо подпрограммний ШК. Якщо компілятор використовує двухбайтное ACALL замість трехбайтних LCALL-ів скрізь, де це можливо, більшість подпрограммного ШК буде

займати так само мало місця, як і у випадку з прямим і непрямим ШК.

Підпрограмний ШК припускає, що в якості покажчика вершини стека повернень використовується апаратний покажчик стека. У AVR осередки простору в накрістальном регістровому файлі, не достатньо місця для зберігання стеків в багатозадачному системі. Тому ви можете: a) обмежитися однозадачной Форт-системою; b) писати таким чином всі Форт-слова, що під час виклику вони зберігають адресу повернення в програмний стек в ОЗУ; або c) робити перемикання завдань із збереженням стека повернень у зовнішнє ОЗУ.

Варіант b повільний. Перенесення 128 байт при кожному перемиканні задач буде швидше пересилання двох байт для кожного Форт-слова. Тому виберемо варіант "a", залишивши відкритими двері для варіанту "c" на майбутнє.

Один єдиний дійсно адресний регістр DPTR буде використовуватися для численних потреб. Він буде багатоцільовим робочим регістром W.

По правді, існує два інших регістра, що дозволяють адресувати зовнішню пам'ять: R0 і R1. Вони працюють тільки з 8бітнимі адресами, старші 8 біт явно виводяться в port 2. Але це дозволене обмеження для стеків, тепер вони будуть обмежені простором в 256 байт. Давайте будемо використовувати R0 як покажчик стека даних (PSP).

Це 256-байтнoє простір може бути використано під область клієнтів. Це робить P2 (другий порт) другим байтом UP, і, подібно 6809, молодший байт буде завжди нулем.

3.4 Забезпечення термінального зв'язку

Термінальний зв'язок утворюється за рахунок мікросхеми COM-USB перехідника. Тобто, використовуючи УАПП, вбудований у мікроконтролер, можна по кабелю USB передавати дані на ПК.

При реалізації передачі даних по УАПП застосовано стандартні техніки зчитування/запису послідовних даних. Це функції:

а) USART_vInit — ініціалізація приймача і передавача

б) USART_Receive — очікування байту з вхідного каналу

в) USART_Transmit — пересилка байту у вихідний канал

Загальна схема вибрана простою, без переривань, для простішого відлагодження механізму.

Переривання готовності працює дещо складніша. Суть в тому, що робота АЦП у нас йде за таким алгоритмом: запускаємо одиночне АЦП перетворення і чекаємо переривання готовності АЦП. У перериванні забираємо дані. Зберігаємо дані. Перемикаємо канал з якого ми знімали показання. Запускаємо наступне перетворення (вже для іншого каналу) Виходимо з переривання і чекаємо следующего переривання.

Даний алгоритм робить все автоматично, в режимі кінечного автомата. У результаті у нас в пам'яті, в масиві ADCCH, завжди лежать 8 свіжих значень, знятих з 8ми каналів АЦП. Залишається їх тільки рахувати і використовувати. При цьому головний цикл крутиться по своїх справах і не париться, знаючи, що свіжі значення завжди його чекають. У самому перериванні активно використовується робота з масками. Для того, щоб змінити номер каналу.

Номер каналу лежить в останніх трьох бітах регістру ADMUX і може мати значення від 0 до 7 (000 і 111 соотвествтенно). І нам треба в кожному виклику збільшувати значення каналу, перебираючи їх по черзі. Просто так ікрементіровать ADMUX не можна, тому що крім номера каналу там лежать ще й бити управління АЦП, вирівнювання і опорного напруги — вони можуть збитися.

3.5 Узагальнення інформації про Форт

Те, що я уже зробив, це ще не Форт у буквальному значенні цього слова, проте це Форт у значенні ідеології. Максимальна зручність при мінімально можливому розмірі, макро-надбудова над асемблером з можливостями мов високого рівня, наявність діалогового режиму, абстрагованість від системи (у прикладах я показував як можна замінити виклик CR двома EMIT-ами, проте тому і існує слово CR, щоб абстрагуватись від способу переведення на новий рядок у різних комп'ютерах), словник та слова – ось що таке Форт-ідеологія, ось що є суттю.

Форт – це не мова програмування, це спосіб мислення. Як видно з попереднього твердження, стеки даних не є необхідним, щоб мова називалась Форт-подібною. В історії існують приклади Форт-систем без стеку даних. Просто, як правило стек реалізується через те, що він є найбільш зручним при найменших

затратах на його організацію.

Історично так склалось, що Форт-слова повинні писатись капсом, проте це зовсім не означає, що я не можу писати слова маленькими буквами. Від цього мій Форт не перестане бути Фортом.

У Форті дуже мало внутрішніх перевірок на коректність роботи програми та коректність дій користувача, а ті які є можна завжди можна обійти. Це наслідок того, що Форт – всього лиш надбудова над асемблером і як в асемблері, так і в Форті дозволено робити все, що заманеться. Мабуть через це Форт досі вважається "метрвою", нежиттєздатною мовою.

В світі великих технологій, швидкого та розприділеного між багатьма людьми програмування, в світі мільйонів вірусів Форт – як біла ворона. Хоча насправді, доки живуть хакери, доти буде жити й Форт. GRUB L.2, OpenBIOS, PostScript – приклади того, що Форт живіший всіх живих і ніколи не помре як ідея. Форт-ідеологію не можливо збагнути, доки сам не напишеш хоча б частину свого власного Форт-інтерпретатора.

Дуже часто у Форті використовуються конструкції, які вганяють у ступор звичайних програмістів. По-перше, стекове мислення, по-друге, – коментарії в дужках, по-третє, – використання символів як ключових слів. Ось неповний список стандартних односимвольних слів:

- . (крапка) – вивести число в вершині стеку на екран
- ((дужка) – означає початок коментаря
- @ (собачка) – зчитати число в пам'яті
- ! (знак оклику) – записати число в пам'ять
- # (шарпик) – забрати у числа останню цифру
- [(кв. дужка) - перейти у режим інтерпретації
- ‘ (апостроф) – знайти слово в словнику
- \ (зворотній слеш) – однорядковий коментар
- : (двокрапка) – створити заголовок для слова і перейти у режим компіляції
- , (кома) – записати число зі стека по адресі HERE і пересунути вказівник DP на CELL байт вперед

Стандарти пишуться на основі досвіду, тому я також буду намагатись слідувати стандарту. Проте, стандарту потрібно дотримуватись тоді, коли потрібна якась взаємодія між програмами, написаними різними людьми.

Форт має такі переваги над іншими мовами: стек даних, можливість зміни роботи інтерпретатора на льоту, можливість створення нових конструкцій мови, мінімально можливий генерований код, відсутність обмежень, простота відладки програми, простота синтаксису і ще багато-багато інших переваг, типу реалізація інтерпретатора Форта на Форт займає всього 5 рядків, зручний інтерактивний режим.

У Форті відсутня типізація, і це є принципом, а не недоліком. Якщо програмісту потрібен якийсь тип, будь-ласка, реалізовуй сам. Благо Форт дозволяє це робити. Нехай вас не лякає, що арифметика Форта цілочисленна. При сильній потребі реалізувати дробові числа можливо і навіть кількома способами.

Наведений приклад заодно вказує на унікальну особливість Форту: відсутність списку параметрів в дужках і можливість програмувати на рідній мові.

Використання словникових конструкцій рідної мови дозволяє зробити програму зрозумілою, що підвищує її надійність. «Зворотний польський запис» арифметичних виразів і наявність декількох стеків. Двоїста природа компілятора Форту. Не можна стверджувати однозначно, чи є Форт компілятором або інтерпретатором.

Практично завжди його можна використовувати в двох режимах, за винятком рідкісних випадків на кшталт «цільової компіляції» (трансляції в машинний код програми для системи з іншою архітектурою). Відсутність системи типів. Подібно мов асемблера, у Форті немає вбудованої системи типів. Немає можливості дізнатися, що лежить на вершині стека - число зі знаком, число без знака, покажчик на рядок, символ, або два числа, що розглядаються як одне довге число.

Контроль типів покладається на програміста. При цьому використовуються спеціальні набори слів (наприклад, запис і читання елементів пам'яті виробляють словами! I @, а символів - словами C! і C @), деякі сутності виносяться в спеціальні стеки (наприклад, стек чисел з плаваючою комою, відповідно до стандарту ANSI FORTH 94; він може бути, а може і не бути, реалізований за допомогою основного стека).

Свобода, що надається програмісту, вимагає сильного самоконтролю. Вхідний поріг для програмування на Форте нижче, ніж у мов типу C++, але вимагає

звикання і розуміння не тільки можливостей і особливостей синтаксису Форту, але, також, розуміння філософії, що лежить в його основі. Форт не підтримує жодну парадигму програмування і підтримує їх всі одночасно.

Написати набір слів для організації ООП у програмі на Форте (а їх може бути одночасно декілька і вони будуть відмінно уживатися разом) набагато простіше, ніж вирішити, які можливості від цього набору слів потрібні. Розбивка програми на безліч дрібних слів дозволяє легко і швидко перевіряти їх окремо, передаючи їм потрібні набори вхідних параметрів і контролюючи те, що залишається на стеку. Фактично, це означає, що для тестування якогось компонента програми можна не завантажувати всі залежні компоненти цілком. Форт не приховує помилки. Цей факт встановлено досвідченим шляхом.

«Відкладені» помилки у програмі на Форте — велика рідкість. Помилки, які, у звичайних мовах програмування, ховаються стандартним перетворенням типів (наприклад, int в char в C++ (хоча більшість сучасних компіляторів видасть, звичайно, попередження) або рядка в число в якому-небудь скриптовій мовою), практично миттєво, при наступному ж тестовий запуск, «обрушують» програму. Більшість реалізацій форту дозволяють зробити декомпіляцію програми. Отриманий текст мало відрізняється від початкового. Форт дозволяє реалізувати будь-яку технологію програмування, доступну в інших мовах і системах. У ньому також припустимі прийоми, заборонені в інших мовах (наприклад — самомодіфікації коду). Усунути негативні наслідки цих прийомів шляхом створення правильного лексикону, стимулюючого грамотну методику їх використання також покладено на програміста. У інтерпретаторі легко реалізувати всі перевірки на межі діапазону адрес, а це при створенні ОС дозволяє відмовитися від захищеного режиму процесора. Виходить суттєвий виграш у швидкості роботи.

Розмір коду Форту для 16-розрядних систем, при грамотному написанні програми, іноді в 10-20 разів менше коду, скомпільованого з програми на Сі. Для 32-розрядних систем цей розрив ще більше. В операційних системах загальний виграш може становити вже сотні, а то й тисячі разів. Причина дуже проста — готова завдання на Форте має розмір кілька байт, всі допоміжні підпрограми реалізовані у вигляді визначень, які доступні всім. Система на Форте вміститься в процесор, у який інші системи влізти в принципі не здатні. Синхронізація процесів та потоків в багатозадачних системах, перемикання контексту, реалізація доступу до обмежених ресурсів — найскладніші проблеми при написанні ОС.

Для підтримки цих можливостей навіть створюються спеціальні команди в мікропроцесорах. Для інтерпретатора це взагалі не проблема, оскільки він емулює будь-який процесор і будь-яку необхідну команду. Можливо, що насправді найбільше розвитку Форту перешкоджає «важка спадщина», що прийшов з машин з низькими можливостями, для яких він спочатку створювався.

При програмуванні з активним використанням арифметики з плаваючою точкою, цю норму стандарту традиційно ігнорують. Аналогічна норма існує відносно стека потоку керування. Тут усе не так просто, так як часто це саме так і є — в процесі компіляції стек використовується самим компілятором.

В абсолютній більшості випадків ніякого впливу на програму це не робить, але про саму особливість треба пам'ятати. Наприклад, якщо ви хочете в процесі компіляції обчислити якесь число, за межами початку визначення, а потім вставити його в слово як константу, то для цього доведеться використовувати який-небудь обхідний шлях.

Визначення багатьох слів у стандарті занадто низькорівневі. Наприклад, слово 2* виробляє не множення на два, як випливає з його назви, а «зміщує число на один біт до старшого двійковому розряду, заповнюючи молодший біт нулем». Звичайно, на більшості сучасних машин - це одне і те ж, але сам факт використання особливостей конкретної архітектури насторожує. (Існують також більш очевидні стандартні слова для зрушення бітів — LSHIFT і RSHIFT.)

4 Інструкція користувача

4.1 Необхідне програмне забезпечення

4.1.1 Операційна система Linux

Linux — загальна назва UNIX-подібних операційних систем на основі однойменного ядра. Це один із найвидатніших прикладів розробки з відкритим кодом та вільного програмного забезпечення; на відміну від пропрієтарних операційних систем, на кшталт Microsoft Windows та MacOS X, її вихідні коди доступні усім для використання, модифікації та розповсюдження абсолютно вільно (в т.ч. безкоштовно).

Про TCP/IP, одному з найбільше інтенсивно використовуваних стеков протоколів. Книга починається з елементарного введення в теорію комп'ютерних мереж і межсетевого взаємодії, потім іде виклад мережних моделей OSI і TCP/IP, далі випливають опису кожного рівня й кожного протоколу стека TCP/IP, супроводжувані прикладами з реалізації цієї моделі в Linux. Опис кожного нового протоколу й кожного нового поняття йде по тій же схемі від простого до складного, що й увесь виклад, тому книга вдало поєднує в собі доступність поступового введення з обґрунтованістю монографії. У той же час, достаток прикладів не дає читачеві згубитися в нетрях абстракцій, створюючи відчуття реальності, недолік якої часто утрудняє засвоєння складного матеріалу. Авторові вдалося сполучити повноту викладу з виразністю, що суттєво розширює коло читачів. Там, де подальша деталізація загрожує вийти за розумні межі, приводяться посилання на відповідну літературу й мережі.

Як у кожній бочці меду є ложка дьогтю, так і кожна система має свого адміністратора. А адміністрування системи - це дуже важлива і іноді пожирає силу-силенну часу робота, навіть якщо ви єдиний користувач системи.

Ми постараємося обговорити тут найбільш важливі речі, пов'язані з адмініструванням, про який ви повинні знати при використанні Linux, щоб не відчували незручностей при роботі з ОС. Щоб бути не надто балакучими і приємними співрозмовниками, ми і раніше розглядали тільки основні риси, пропускаючи багато важливі деталі. Це допоможе вам краще зрозуміти як там все відбувається, і як там все взаємодіє. У крайньому випадку, варто все це переглянути, щоб знати що у книзі міститься і якої допомоги вам слід від неї очікувати.

UNIX розрізняє різних користувачів, так що те, що вони можуть зробити один одному і системі, регулюється (наприклад, не хочеться, щоб хтось читав чужі любовні листи). Кожен користувач отримує account (реєструється в системі), що включає ім'я користувача, домашній каталог і т.д. На додаток до реєстрації реальних людей, реєструються (для них також відкривається рахунок кілька спеціальних користувачів, що мають привілеї. Найбільш "важливий" навіть серед них користувач — root (корінь).

Звичайні користувачі в загальному випадку обмежені так, що вони не можуть заподіяти шкоду кому-небудь іншому в системі (включаючи саму систему), крім самих себе. Права доступу до файлів в системі організовані таким чином, що простий користувач не може видалити або змінити файл, файл у каталогах, які користувачі використовують спільно (такі як /bin і /usr/bin). Більшість користувачів також захищають свої власні файли так, що не можуть їх змінити, а іноді і взагалі дістатися до них.

Всі ці обмеження не поширюються на користувача root. Користувач root може читати, модифікувати або видаляти будь-який файл системи, змінювати його права доступу або змінювати його власника. Він (root) може також виконувати спеціальні (привілейовані) програми, такі як розбиття диска на розділи або створення файлової системи. Основна ідея полягає в тому, що той, хто виконує реєстрацію користувачів, повинен, якщо це необхідно, мати можливість виконувати роботи, які не можуть бути виконані звичайним рядовим користувачем. Оскільки root може робити все, що завгодно, йому легко зробити якусь помилку, що приводить до катастрофічних наслідків.

Наприклад, якщо ви як звичайний користувач випадково спробуєте видалити файл в /etc, система не дозволить вам це зробити. Але, якщо ви увійшли як root, система навіть не пискне, виконуючи все, що накажете. Легко знищити систему, перебуваючи в системі як root.

Посидіти на власних долоньках, перш ніж натиснути return для виконання команди, яка може бути причиною катастрофи. Наприклад, якщо ви збираєтеся очистити каталог, перед натисканням return перечитайте всю команду і переконайтеся, що вона написана правильно. Чи не звикайте використовувати root. Чим більш комфортно вам буде в ролі root, тим більше ви будете плутати ваші привілеї з привілеями нормального користувача. Наприклад, ви можете подумати, що ви зараз перебуваєте в системі як larry, хоча насправді будете нестримним root.

Використовуйте відрізняється підказку для root. Для цього слід внести зміни до root-івський. Bashrc або. Login файл для того, щоб зробити підказку для root відмінною від інших. Наприклад, багато хто використовує символ “$” в підказках звичайних користувачів і залишають символ “#” для підказки root. Заходьте під ім’ям root тільки тоді, коли це абсолютно необхідно. І, як тільки ви закінчите роботу root-а, вийдіть (виведіть root-а з системи). Чим менше використовуєте root, тим менше нашкодите системі. Зрозуміло, є плем’я хакерів, які використовують root практично завжди і скрізь. Але кожен з них колись по дурості знищив хоча б (у кращому випадку) одну систему. Є загальне правило: поки ви не познайомилися з необмеженими можливостями root, і не звикли до відсутності обмежень, входьте під root в крайньому випадку.

Давайте по-іншому, якщо ви представите використання root як носіння спеціального чарівної шапки, яка дає вам могутність, так що ви можете помахом руки зруйнувати цілі міста, то доречна думка, що треба дуже стежити за своїми руками. А оскільки така міць небезпечна (та й рук незручно), краще без великої потреби не надягати чарівну шапку, навіть якщо в шапці у вас підвищується самоповага.

З приходом відчуття влади приходить бажання шкодити. Це темна сторона адміністрування в UNIX, але всякий через це колись має пройти. Більшість користувачів UNIX ніколи не отримають можливість випробувати це на університетських і виробничих системах UNIX. Тільки високооплачувані та високоосвічені системні адміністратори можуть входить в систему під іменем root. Дійсно, у багатьох таких закладах пароль root - це строго охороняється секрет. Це священна корова фірми. Багато робиться спроб пролізти під ім’ям root в систему; вона представляється мудрою і страхітливою силою, піддаються тільки тим, хто знає заклинання.

Така позиція по відношенню до root дуже легко призводить до небезпек і спокусам. Оскільки root настільки одурманюючих штука, то коли користувач дориваються до можливості увійти під root, простежується початок використання звалилися привілеїв в плані шкідництва. Я знав таких "системних адміністраторів", які читали без дозволу пошту інших користувачів і взагалі вели себе як діти, яким дали таку потужну кльову "іграшку".

Оскільки root має в системі такі привілеї, потрібен певний рівень зрілості і самоконтролю, щоб використовувати цей account (цей привілейований "раху-

нок"), як це було задумано - для експлуатації системи. Існує негласний закон честі у відносинах адміністратора з користувачами. Як ви будете почуватися, якщо системний адміністратор читає ваші листи і переглядає ваші файли. До цих пір немає достатньо серйозної юридичної основи для недоторканності особистої інформації в багатокористувацьких комп'ютерних системах. У системах сімейства UNIX користувач root має можливість долати всі штатні механізми захисту системи. Важливо, щоб у адміністратора були довірчі відносини з користувачами системи. Неможливо переоцінити важливість цього.

Питання безпеки були домислені «в догонку» — початково система створювалася в неформальній атмосфері, коли всі втручалися в роботу один одного. Завдяки цьому, навіть незважаючи на заходи безпеки, у нормальної користувача існують можливості заподіяти системі шкоду.

Системний адміністратор може вибрати дві тактики взаємодії з користувачами. Це може бути параноїдна тактика і тактика довіри. Системний адміністратор з параноєю звичайно своїми діями завдає більше шкоди, ніж запобігає. Ніколи не списуй на шкідливість те, що можна списати на незрозумілість. Погляньте з іншого боку, більшість користувачів не мають можливостей і знань, щоб заподіяти реальну шкоду системі. Девяносто відсотків того, що робить користувач, завдаючи шкоди системі (наприклад, забиваючи власний розділ величезними файлами або виконуючи відразу кілька екземплярів величезної програми), він робить просто не підозрюючи, що він комусь щось створює проблеми. Мені доводилося стикатися з користувачами, які були джерелами величезних неприємностей, але вони діяли по простоті душевній, а не зі зла.

Коли ви маєте справу з користувачами, які небезпечні потенційно, не накидають на них із звинуваченнями. Старе правило все ще не скасували. Краще всього поговорити з користувачем, попитати про його проблеми, замість того, щоб йти на конфронтацію. Найгірше, це намагатися відповідати йому «зустрічним». Це створить навколо вас - системного адміністратора - багато підозр, поставить під сумнів вашу здатність коректно підтримку системи. Якщо користувач вирішить, що ви не вірите йому або навіть не любите, він може звинуватити вас у тому, що ви видаляєте його файли і взагалі підглядаєте. Навряд чи ви хочете опинитися в такій ситуації.

Якщо ви створили керівництво для користувачів системи, переконайтеся, що причини введення тих чи інших правил їм зрозумілі. Якщо ви цього не зро-

бите, користувачі творчо підійдуть до того, як обходити ці правила. може бути і не усвідомлюючи, що вони їх дійсно обходять.

Ми не можемо до останньої деталі розписати вам, як експлуатувати систему. Велика частина філософії залежить від того, як ви використовуєте систему. Якщо у вас багато користувачів, то це сильно відрізняється від того, коли їх мало, або взагалі ви один. Але при будь-якому розкладі дуже корисно замислитись, що в даній конкретній системі дійсно означають слова «системний адміністратор» (або «адміністратор системи»).

Посада адміністратора системи не робить вас крутим юніксістом. На світі багато системних адміністраторів, які мало що знають про UNIX. Схоже, що існує багато «нормальних» користувачів, які, знають про UNIX більше будь-якого системного адміністратора. Перебування на посаді адміністратора не дає вам права використовувати загрози в адресу користувачів. Саме тому, що система дає вам привілей влаштувати з файлів користувача все, що завгодно, ви не маєте ніякого права це робити.

Нарешті, бути системним адміністратором, це казна-що. При цьому не має значення, опікуєтеся ви маленький триста вісімдесят другому або суперкомп'ютер Cray. Знання заповітного пароля root не принесе вам грошей та слави, воно допоможе супроводжувати систему і підтримувати її працездатність.

4.1.2 Асемблер avra

Асемблер avra є основним інструментом при розробці програм для AVR мікроконтролерів на Лінукс.

У простому випадку асемблер переводить одне речення початкової програми в один об'єкт (команду, константу) модуля завантаження (т. з. трансляція «один в один»). При цьому взаємне розташування об'єктів в модулі завантаження і, зрештою, в пам'яті машини визначається порядком пропозицій в початковій програмі на автокоді і повністю залежить від програміста. Асемблер виконує і допоміжні функції, такі, як підготовка до друку документів необхідної форми, реєстрація зв'язків даної програми з іншими програмами і т. д. Для цієї мети в автокодах передбачаються команди асемблера, які не породжують об'єктів в робочій програмі і призначені тільки для вказівки допоміжних дій асемблера.

Трансляція зазвичай вимагає двох переглядів початкової програми: при першому перегляді здійснюється розподіл пам'яті і надання значень символічним іменам; при другому — формується робоча програма у вигляді модуля заванта-

ження. В процесі трансляції асемблер проводить повний синтаксичний контроль початкової програми (див. синтаксичний аналіз програм), забезпечуючи при цьому достатньо точну діагностику помилок за місцем і характером.

Розширення можливостей автокодів досягається за рахунок використання макрокоманд, що будуються за правилами, близькими до правил написання команд автокоду, але що описують складніші функції, для реалізації яких потрібна група звичайних команд. В цьому випадку перед трансляцією проводиться заміна макрокоманд макророзширеннями — послідовностями команд на базовій мові відповідно до макроозначень. У останніх задається прототип макрокоманди із структурою списку параметрів і процедура генерування макророзширення.

Транслятор, що виконує функції макрогенератора і асемблера, називається макроасемблером. При трансляції з мов високого рівня асемблер нерідко використовується для виконання завершальної фази трансляції.

4.1.3 Скриптовий інтерпретатор Python

Python — інтерпретована об'єктно-орієнтована мова програмування високого рівня з динамічною семантикою. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднання існуючих компонент. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій так і у вихідній формі на всіх основних платформах. В мові програмування Python підтримується декілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Python підтримує динамічну типізацію, тобто, тип змінної визначається лише під час виконання. З базових типів слід зазначити підтримку цілих чисел довільної довжини і комплексних чисел. Python має багату бібліотеку для роботи з рядками, зокрема, кодованими в юнікоді. З колекцій Python підтримує кортежі (tuples), списки (масиви), словники (асоціативні масиви) і від версії 2.4, множини. Система класів підтримує множинне успадкування і метапрограмування. Будь-який тип, включаючи базові, входить до системи класів, й за необхідності можливе успадкування навіть від базових типів.

Подібно Ліспу та Прологу в режимі відлагодження, інтерпретатор Python має інтерактивний режим роботи, при якому введені з клавіатури оператори відразу ж виконуються, а результат виводиться на екран. Цей режим цікавий не

тільки новачкам, але й досвідченим програмістам, які можуть протестувати в інтерактивному режимі будь-яку ділянку коду, перш ніж використовувати його в основній програмі, або просто використовувати як калькулятор з великим набором функцій.

Для роботи потрібна версія 3. Основні зміни, внесені до версії 3.0:

- Синтаксична можливість для анотації параметрів і результату функцій (наприклад, для передачі інформації про тип або документування).
- Повний перехід на unicode для рядків.
- Введення нового типу «незмінні байти» і типу «змінюваний буфер». Обидва необхідні для подання двійкових даних.
- Нова підсистема вводу-виводу (модуль io), що має окремі вигляди для бінарних і текстових даних.
- Абстрактні класи, абстрактні методи. Ієрархія типів для чисел.
- Зміни print з вбудованого виразу у вбудовану функцію. Це дозволить модулям робити зміни, підлаштовуючись під різне використання функції, а також спростить код. У Python 2.6 ця можливість активується введенням from __future__ import print_function.
- Переміщення reduce (але не map або filter) з вбудованого простору в модуль functools (використання reduce істотно менш читабельне в порівнянні з циклом).
- Видалення деяких застарілих можливостей, які підтримуються у гілці 2.x для сумісності, зокрема: класи старого стилю, цілочисельний поділ з обрізанням результату як поведінка за вмовчанням, рядкові винятки, неявний відносний імпорт, оператор exec тощо
- Реорганізація стандартної бібліотеки.
- Новий синтаксис для метаклассів.
- Змінений синтаксис присвоєння. Стало можливим, наприклад, надання (a, * rest, b) = range(5). З іншого боку, формальні параметри функцій на зразок def foo (a, (b, c)) більше неприпустимі.

4.2 Написання Форт-програм

Для написаннння Форт-програм можна використовувати будь-який текстовий редактор.

Приклад коду Форт програми приведено у додатку А. Зауважте, що використання скрипта на мові програмування Python для завантаження програми — обов'язкове!

4.3 Користування макропрепроцесором

Макропрепроцесор виконує роль метапрограмного підходу до програмування, а саме генерацію коду. При цьому підході код програми не пишеться вручну, а створюється автоматично програмою-генератором на основі іншої, більш простої програми. Такий підхід набуває сенсу, якщо при програмуванні виробляються різні додаткові правила (більш високорівневі парадигми, виконання вимог зовнішніх бібліотек, стереотипні методи реалізації певних функцій та ін.) При цьому частина коду втрачає змістовний сенс і стає лише механічним виконанням правил. Коли ця частина стає значною, виникає думка задавати вручну лише змістовну частину, а решту додавати автоматично. Це і робить генератор.

Реалізується двома основними методами: шаблони (найбільш відомі випадки застосування — препроцесор C та шаблони в C++) вирішують завдання, якщо дотримання «правил» зводиться до вставки в програму повторюваних (або майже повторюваних) шматків коду. Крім цього, мають ще ряд переваг: наприклад, допомагають повторному використанню.

Зовнішні засоби (наприклад генератори синтаксичних і лексичних аналізаторів lex, yacc, bison) застосовуються у випадках, якщо простих засобів на зразок шаблонів недостатньо. Мова генератора складається так, щоб автоматично або з мінімальними зусиллями з боку програміста реалізовувати правила парадигми або необхідні спеціальні функції. Фактично, це — більш високорівнева мова програмування, а генератор — не що інше, як транслятор. Генератори пишуться, як правило, для створення спеціалізованих програм, в яких дуже значна частина стереотипна, або для реалізації складних парадигм.

Макропрепроцесор призначений для пришвидшення процесу написання

програми і створення локальних програмних конструкцій для зручного представлення програмних абстракцій. Макропрепроцесор потрібно використовувати у наступний випадах:

- усунення дублювання у простих випадках
- необхідність нової псевдокоманди для процесора
- необхідність генерації асемблерного коду
- потрібен аналіз глобальних декларативних визначень і відповідна поведінка кодогенератора

Для забезпечення даних властивостей мова асемблера розширена синтаксисом, зрозумілим макропрепроцесором. Код макропрепроцесора приведено у Додатку Б.

4.3.1 Перехід у скриптовий режим

Перехід від мови асемблера до мови Python відбувається за рахунок спеціального коментара — «;>python», вихід з мови Python до асемблера відбувається за рахунок спеціального коментара — «;>endpy». Рядок з спеціальним коментарем ігнорується і асемблером, і Python. Приклад наведено у лістингу:

```
1 mem_LIT: .dw itc_LIT
2 mem_ENTER: .dw ENTER
3 mem_EXIT: .dw EXIT
4 ;>python
5 addCompiledWord("LIT", "mem_LIT")
6 addCompiledWord("ENTER", "mem_ENTER")
7 addCompiledWord("EXIT", "mem_EXIT")
8 ;>endpy
```

Скриптовий режим не дозволяє виконувати умовні переходи, якщо термінальні конструкції для умовного переходу знаходяться за межами одного скриптового блоку, обмеженого спеціальними коментарями.

4.3.2 Додавання нових макросів

Макроси представляють собою функції на мові Python і додаються за допомогою службової процедури «addMacro». Макроси бувають двух видів:

- кодозамінюючі
- кодогенеративні

Перший тип дозволяє робити заміну виклику макросу на текст макросу з відповідним підставлянням аргументів макросу. Другий тип дозволяє задавати процедуру будь-якого рівня складності з аргументами макросу у якості вхідних параметрів.

Приклад створення кодозамінюючого макросу з іменем «ldiw»:

```
## ‘ldiw temp, imm16
# temp - register (r16-r30, only even, may be X Y Z)
addMacro("ldiw", """
  ldi @0L, low(@1)
  ldi @0H, high(@1)
""")
```

Приклад створення кодогенеруючого макросу з іменем «COMPILE_RAW»:

```
addMacro("COMPILE_RAW", lambda text: True and
  pasteMacro("\n".join(["\t\t.dw {0}"
              .format(iif(wordDictHasKey(x),
                  lambda:FUNCS["rammap"](wordDictF(x)),
                  lambda:x)())
            for x in eval(text).split()]),
        True, True) )
```

4.3.3 Додавання нових функцій

Функції відрізняються від макросів тим, що вони можуть використовуватись в середині рядка. Таке проекте рішення прийнято з єдиною метою — спрощення фази аналізу макропрепроцесора. Функції, як і макроси, можуть бути кодозамінними, так і кодогенеративними.

Приклад функції приведено у лістингу:

```
## "rammap" converts address in flash to address in RAM after
## loading program to RAM
addFunc("rammap", "(@0 - START_CODE)*2 + WORDS_START")
```

4.3.4 Використання макросів і функцій

Для виклику макроса потрібно вказати його назву і аргументи в такому ж стилі, як програмуються мнемоніки асемблера, проте перед назвою макроса поставити символ «зворотній апостроф».

```
.org 0x40 RESET:
  ; Ініціалізація стеку повернень
  ‘out SPL, low(RETURN_STACK)
  ‘out SPH, high(RETURN_STACK)
```

Для виклику функції потрібно вказати її назву, а аргументи задати в дужках. Перед назвою функції потрібно поставити символ «октоторп».

```
1    sts #rammap(CTIB), ZL
2    sts #rammap(CTIB)+1, ZH
3    'store #rammap(_IN), 0
4    ; Встановити курсор на 0 символ текстового буферу
```

Як видно з прикладу, функції і макроси можна сумісно використовувати при однорівневій глибині вкладеності.

4.4 Робота з стендом EV8031/AVR

В першу чергу потрібно зазначити, що робота зі стендом довзолена тільки після прочитання інструкції та правил безпеки.

При роботі зі стендом типовими є дві операції: завантаження бінарного коду ядра у ПЗУ мікроконтролера і зв'язок з ядром по COM-порту. Перша операція потребує підключення кабелю від послідовного порту ПК до роз'єму на платі, поряд з мікроконтролером. При підключенні кабелю живлення повинно бути уже під'єднано. При другому типі операцій кабель до послідовного порту ПК повинен бути від'єднаний від плати.

4.4.1 Завантаження ядра у мікроконтролер

Завантаження відбувається за допомогою програми avrdude.

```
1 sudo avrdude -p m8515 -U flash:w:main.hex:i -v -c stk201
```

Налаштуванню типово підлягають наступні параметри:

- шлях до бінарного коду ядра (у прикладі «main.hex»)
- тип програматора (у прикладі «stk201»)

Вихідний код ядра приведено у Додатку В.

4.4.2 Налаштування параметрів терміналу

Підключення до терміналу стенду відбувається через утиліти стандарту POSIX, для їх налаштування використовується утиліта stty

```
1 sudo stty -F /dev/ttyUSB0 9600 raw cs8 -parity
```

Змінним параметром є адреса термінального порта, що задається ключем «-F» і швидкість передачі.

4.4.3 Утворення термінального зв'язку

Термінальний зв'язок утворюється за допомогою наступних утиліт стандарту POSIX:

– echo — для виводу у порт

– cat — для вводу із порту

– перенаправлення потоку даних (оператор «»>)

Висновки

В даній роботі було розглянуто способи створення Форт інтепретатора для мікроконтролерів AVR. В результаті було отримано робочу реалізацію, готову для використання. Отримана Форт-система має необхідну гнучкість для подальшого розширення.

Було проведено дослід по модифікації програмного середовища під проект. Отримано наступні результати:

- на синтаксичний аналізатор витрачено 24 людино-години (3 дні)
- отримано набагато більш потужний препроцесор, ніж йде по змовчуванню із більшістю сучасних асемблерів
- зекономлено час на створення програмних конструкцій певного вигляду і спрощено підтримку коду в майбутньому
- закладено можливість декларативного налаштування кодогенерації

Оскільки робота завершена і позитивні результати отримані, дослід вважається успішним і дана технологія рекомендується для подальшої перевірки на робочому проекті.

Додаток А

Лістинг коду програми завантажувача

```
1  import sys, os
2
3  source_text = """
4    0 ->LEDS 0 ->IND
5    1 2* 2* ->portA
6    0 C@ 1 + DUP 1 2* 2* 2* 2* 2* AND INVERT 2/ 2/ ->portC 0 C!
7    0 >IN ! LOOP FOREVA!!!!
8  """
9
10
11 def main():
12   #inp = sys.argv[1].replace('\\!', '!')
13   inp = source_text.replace("\n", " ").replace("\t", " ")
14   packets = []
15   temp = ""
16   for x in inp:
17     temp += x
18     if len(temp) == 6:
19       packets += [temp]
20       temp = ""
21   if len(temp) > 0:
22     temp += " "*(6-len(temp))
23     packets += [temp]
24   packets += ["     \r"]
25   for pack in packets:
26     os.system("echo '{0}' > /dev/ttyUSB1".format(pack))
27
28 if __name__ == "__main__":
29   main()
```

Додаток Б

Лістинг коду макропрепроцесора

```
1  #!/usr/bin/python3
2
3  # Макропрепроцесор для проекту StandForth
4  # Підтримуються наступні команди:
5  # Загальні
6  # - pasteMacro - звичайна або( функціональна) макропідстановка
7  # - gEX - виконання будьякого- Пітонівського коду
8  # - callLambda - виклик lambda функції по списку аргемнтів
9  # - addMacro - декларація макросу
10 # - addFunc - декларація однорядкової макропідставновки
11 # - pasteLine - звичайна вставка тексту у вихідний файл
12 # Специфічні
13 # - addCompiledWord - додати слово до списку компільованих
14 # - wordDictF - повернути список слів
15 # - wordDictHasKey - перевірка наявності ключа у словнику
16 # - printF - вивід на екран відладки текстового повідомлення
17
18 import shlex
19
20 MACROTEXT = ""
21
22 MACROS = {}
23
24 FUNCS = {}
25
26 lastCFA = 0
27
28 def pasteMacro(text1, startNewLine = True, endNewLine = True)
      :
29   global MACROTEXT
30   if startNewLine:
31     MACROTEXT += "\n"
32   MACROTEXT += text1
33   if endNewLine:
34     MACROTEXT += "\n"
35   return True
```

```
36
37 def gEX(execStr):
38   exec(execStr, globals())
39   return True
40 def lEX(execStr):
41   exec(execStr, locals())
42   return True
43
44 def callLambda(lam, args):
45   if len(args) == 1:
46     lam(args[0])
47   elif len(args) == 2:
48     lam(args[0], args[1])
49   elif len(args) == 3:
50     lam(args[0], args[1], args[2])
51   elif len(args) == 4:
52     lam(args[0], args[1], args[2], args[3])
53
54 def addMacro(macroName, macroFunc):
55   if type("") == type(macroFunc):
56     MACROS[macroName] = lambda *args: pasteMacro(
     pasteWithReplace(macroFunc, args), True, False)
57   else:
58     MACROS[macroName] = macroFunc
59
60 def addFunc(funcName, macroFunc):
61   if type("") == type(macroFunc):
62     FUNCS[funcName] = lambda *args: pasteWithReplace(
     macroFunc, args)
63   else:
64     FUNCS[funcName] = macroFunc
65
66 def pasteWithReplace(text, args):
67   global MACROTEXT
68   for i in range(0, len(args)):
69     text = text.replace("@" + str(i), str(args[i]))
70   return text
71
72 def iif(a,b,c):
73   if a:
74     return b
```

```
75    else:
76      return c
77
78
79  def preprocessText(text):
80    global MACROTEXT
81
82    state = "asm"
83    resultPy = ""
84    for line in text.splitlines():
85      if state == "asm":
86        #line = line.strip()
87        if line.strip().endswith(";>python"):
88          state = "py"
89        else:
90          while True:
91            res = parseForFunc(line)
92            if not res["result"] == "found":
93              break
94            #print(res)
95            line = res["preline"] + FUNCS[res["funcname"]](*res
      ["args"]) + res["postline"]
96
97          if line.strip().startswith("`"):
98            #print("start     " + line)
99            parseMacroLine(line)
100           #resultText += MACROTEXT + "\n"
101           #MACROTEXT = ""
102         else:
103           #resultText += line + "\n"
104           MACROTEXT += line + "\n"
105     elif state == "py":
106       if line.endswith(";>endpy"):
107         state = "asm"
108         #print(resultPy)
109         gEX(resultPy)
110         resultPy = ""
111       else:
112         resultPy += line + "\n"
113         pass
114
```

```
115 def pasteLine(line):
116   global MACROTEXT
117   MACROTEXT += line + "\n"
118
119 def parseMacroLine(line):
120   for someCode in [ x for x in line.splitlines() if len(x.
      strip())>0]:
121     firstWord = someCode.split()[0]
122     restWords = someCode.strip().lstrip(firstWord)
123     if firstWord[0] == "‘":
124       firstWord = firstWord[1:]
125       restWordsStrippedSpaces = "".join([x for x in shlex.
      shlex(restWords)])
126       lexer = shlex.shlex(restWordsStrippedSpaces)
127       lexer.whitespace = ","
128       lexer.commenters = ";"
129       lexer.whitespace_split = True
130       args = [token for token in lexer]
131       if firstWord in MACROS:
132         #print(restWords+"\n", firstWord, args)
133         callLambda(MACROS[firstWord], args)
134       else:
135         print("!! Unknown macro")
136         print(line)
137         exit()
138
139
140 def parseForFunc(line):
141   ret = {}
142   ret["result"] = "nofunc"
143   sharpstart = line.find("#")
144   if sharpstart == -1:
145     return ret
146   else:
147     ret["preline"] = line[:sharpstart]
148     line = line[sharpstart+1:]
149     parentstart = line.find("(")
150     if parentstart == -1:
151       ret["result"] = "error"
152       return ret
153     else:
```

```
154         state = "start"
155         oldstate = state
156         funcname = line[:parentstart]
157         line = line[parentstart+1:]
158         parcount = 1
159         counter = 0
160         endsymb = -1
161         for symb in line:
162           counter += 1
163           if symb == "(" and not state == "string":
164             parcount += 1
165           elif symb == ")" and not state == "string":
166             parcount -= 1
167           elif symb == '"' and not state == "string":
168             oldstate = state
169             curstate = "string"
170           elif symb == '"' and state == "string":
171             state = oldstate
172           else:
173             pass
174           if parcount == 0:
175             endsymb = counter
176             break
177         if not parcount == 0 or endsymb == -1:
178           ret["result"] = "error"
179           return ret
180         else:
181           ret["result"] = "found"
182           ret["postline"] = line[endsymb:]
183           ret["funcname"] = funcname
184           lexer = shlex.shlex("".join([x for x in shlex.shlex(
      line[:endsymb-1])]))
185           lexer.whitespace = ","
186           lexer.commenters = ";"
187           ret["args"] = [token.strip() for token in lexer]
188           return ret
189
190 wordDict = {}
191
192 def addCompiledWord(w, naddr):
193   global wordDict
```

```
194   wordDict[w] = naddr
195   return True
196
197 def wordDictF(wwww):
198   global wordDict
199   return wordDict[wwww]
200
201 def wordDictHasKey(k):
202   global wordDict
203   return k in wordDict
204
205 def printF(*x):
206   print(*x)
207   return True
208
209 import os
210
211 def main():
212   inputFile = open("standforth.asm").read()
213   # перегнати код через макропрепроцесор
214   preprocessText(inputFile)
215   outputFile = open("_gen_standforth.asm", "w")
216   outputFile.write(MACROTEXT)
217   # скопілювати отриманий файл
218   os.system("avra _gen_standforth.asm")
219   print("Ok")
220   pass
221
222 if __name__ == "__main__":
223   main()
```

Додаток В

Лістинг коду основної програми

```
;################################################
;; GLOBAL CONFIG
;;
;>python
## Compiler type. Current available modes:
## - "avra" - linux avr assembler
## - "avrstudio" - windows AVRstudio4 assembler
Compiler = "avra"

## Tab size used in this document
TabSize = 4
;>endpy

;;================================================
;; PART DEFINITION
;>python
if Compiler == "avra":
    pasteLine('.include "partdef.inc"')
elif Compiler == "avrstudio":
    pasteLine('.include "m8515def.inc"')
;>endpy

;;================================================
;; REGISTER MAP
;.def dspL = r2 ; data stack pointer
;.def dspH = r3

.def destL = r4
.def destH = r5
.def sourceL = r6
.def sourceH = r7
.def countL = r8
.def countH = r9

.def ZZL = r20 ; акумулятор
.def ZZH = r21
```

```
37
38  .def tempIL = r10
39  .def tempIH = r11
40
41  .def temp   = r16
42  .def temp2  = r17
43  .def tempL  = r18
44  .def tempH  = r19
45  .def temp2L = r22
46  .def temp2H = r23
47
48  ; free
49  ; .def r0 =  ; використовується у операціях множення
50  ; .def r1 =
51  ; .def r12 =
52  ; .def r13 =
53  ; .def r14 =
54  ; .def r15 =
55   .def dspL = r24
56   .def dspH = r25
57
58
59  ;#################################################
60  ;; DATA SEGMENT
61  .dseg
62
63  DATA_STACK:     .byte 50
64  RETURN_STACK:   .byte 2
65
66  ENTRY_POINT:    .byte 60
67  PROG_START:     .byte 60
68  WORDS_START:    .byte 2
69
70  ; Environment specific defs
71  .include "ev8031.asm"
72
73
74  ;#################################################
75  ;; CHIP settings
76  ; Define baud rate
77  ;.equ USART_BAUD = 38400
```

```
78  .equ USART_UBBR_VALUE = 47
79
80  ;#################################################
81  ;; MACRO and FUNCTIONS
82  ;>python
83  # out IOregister, imm
84  addMacro("out", """
85      ldi temp, @1
86      out @0, temp
87  """)
88
89  # 'store ADDR, imm16
90  addMacro("store","""
91      ldi tempL, low(@1)
92      ldi tempH, high(@1)
93      sts @0,    tempL
94      sts @0+1, tempH
95  """)
96
97  ## "rammap" converts address in flash to address in RAM after
98  ##   loading program to RAM
99  addFunc("rammap", "(@0 - START_CODE)*2 + WORDS_START")
100
101 ## 'ldiw temp, imm16
102 # temp - register (r16-r30, only even, may be X Y Z)
103 addMacro("ldiw", """
104     ldi @0L, low(@1)
105     ldi @0H, high(@1)
106 """)
107 ;>endpy
108
109
110 ;#################################################
111 ;; CODE SEGMENT
112 .cseg
113
114     rjmp RESET
115
116 .org 0x40
117 RESET:
118     ; Ініціалізація стеку повернень
```

```
119      ‘out SPL, low(RETURN_STACK)
120      ‘out SPH, high(RETURN_STACK)
121
122      ; Дозволити зовнішню пам’ять
123      ;  Заодно налаштувати зовнішні переривання
124      ‘out MCUCR, MAGIC_MEMORY_INIT; | (1<<ISC01) | (0<<ISC00)
125
126      ; Дозволити переривання
127      ;‘out SREG, 1<<INT_ENABLE  ;? замінити на sei?
128      ;‘out GICR, 1<<INT0
129
130      ; Ініціалізація стеку даних в( регістрах dspH:dspL)
131      ‘ldiw dsp, DATA_STACK
132
133      ; Копіювати прошивку з flash у RAM
134      rcall copyFirmware
135
136      ; Заповнити системні змінні Форту
137      ‘store #rammap(_CP),          #rammap(END_CODE)
138      ;‘store #rammap(TIB),          #rammap(FORTH_PROG_START)
139      ‘store #rammap(TIB), 0x4000
140      ‘store #rammap(CTIB),         (FORTH_PROG_END -
       FORTH_PROG_START)*2
141      ‘store #rammap(LATEST),       #rammap(mem_NOOP) ; переробити
       через lateVar
142
143      ;;;;;;;;;; UAARTTTT
144  UART_WAIT:
145      rcall USART_vInit
146      clr r15
147      rcall USART_Receive
148
149      ‘ldiw Z, 0x4000
150  uart_loop:
151      rcall USART_Receive
152      sts LEDs, temp
153      sts INDL, temp
154      inc r14
155      cpi temp, 13
156        breq need_interpret
157      st Z+, temp
```

```
158      rjmp uart_loop
159
160  need_interpret:
161      rcall USART_Flush
162      subi ZL, low(0x4000)
163      sbci ZH, high(0x4000)
164      sts #rammap(CTIB), ZL
165      sts #rammap(CTIB)+1, ZH
166      ‘store #rammap(_IN),         0 ; Встановити курсор на 0
       символ текстового буферу
167      ; Запустити асемблерний інтепретатор.
168
169      ‘out UCSRC, (1<<URSEL)
170      ‘out UCSRB, 0
171
172      rjmp interpret
173
174  USART_Flush:
175      sbis UCSRA, RXC
176      ret
177      in temp, UDR
178      rjmp USART_Flush
179
180
181  USART_vInit:
182      ‘out UBRRH, high(USART_UBBR_VALUE)
183      ‘out UBRRL, low(USART_UBBR_VALUE)
184
185      ‘out UCSRC, (1<<URSEL) | (3<<UCSZ0)
186      ‘out UCSRB, (1<<RXEN) | (1<<TXEN)
187  ret
188
189  USART_Receive:
190      ; Wait for data to be received
191      sbis UCSRA, RXC
192      rjmp USART_Receive
193      ; Get and return received data from buffer
194      in temp, UDR
195      ret
196
197  USART_Transmit:
```

```
198      ; Wait for empty transmit buffer
199      sbis UCSRA,UDRE
200      rjmp USART_Transmit
201      ; Put data (r16) into buffer, sends the data
202      out UDR, temp
203      ret
204
205  USART_SendACK:
206      ldi temp, 0x33
207      rcall USART_Transmit
208      ldi temp, 0x33
209      rcall USART_Transmit
210      ldi temp, 0x33
211      rcall USART_Transmit
212      ret
213
214  copyFirmware:
215      `ldiw Z, START_CODE * 2
216      `ldiw X, WORDS_START
217      `ldiw Y, END_CODE * 2
218      sub YL, ZL ; Y = END_CODE - START_CODE
219      sbc YH, ZH
220      ldi temp, 0
221        start_loop:
222          lpm tempL, Z+
223          lpm tempH, Z+
224          st X+, tempL
225          st X+, tempH
226
227          sbiw YL, 2
228          cp YL, temp
229          cpc YH, temp
230            brne start_loop
231      ret
232
233
234  ;;;;;;;;>python
235  lastCFA = ""
236
237  def toName(w):
238      return w[::-1][1:-1]
```

```
239
240 ## ‘HEADER "w", faddr, naddr
241 # w - word name
242 # faddr - assembler code address
243 # naddr - generated memory CFA address
244 addMacro("HEADER", lambda w, faddr, naddr: True and
245     pasteMacro("""
246                 .db "{0}", {1}
247                 .dw {2}
248     {3}:        .dw {4}
249     """.format(
250                 iif(len(w)%2 == 0," ", "") + toName(w), len(
    toName(w)),
251                 iif(lastCFA=="","0",FUNCS["rammap"](lastCFA))
    ,
252     naddr,      faddr
253                 ), True, False) and
254     gEX("lastCFA = ’{0}’".format(naddr)) and
255     addCompiledWord(eval(w), naddr)
256 )
257 addMacro("HEADER_IMM", lambda w, faddr, naddr: True and
258     pasteMacro("""
259                 .db "{0}", {1}
260                 .dw {2}
261     {3}:        .dw {4}
262     """.format(
263                 iif(len(w)%2 == 0," ", "") + toName(w), len(
    toName(w))+128,
264                 iif(lastCFA=="","0",FUNCS["rammap"](lastCFA))
    ,
265     naddr,      faddr
266                 ), True, False) and
267     gEX("lastCFA = ’{0}’".format(naddr)) and
268     addCompiledWord(eval(w), naddr)
269 )
270
271 addMacro("COMPILE_RAW", lambda text: True and
272     pasteMacro("\n".join(["\t\t.dw {0}".format(iif(
    wordDictHasKey(x),lambda:FUNCS["rammap"](wordDictF(x)),
    lambda:x)()) for x in eval(text).split()]), True, True)
273 )
```

```
274 ;;;;;;;;;>endpy
275
276
277 ; Word structure
278 ; - Name (padded spaces left to 2n+1 bytes)
279 ; - Name length (1 byte)
280 ; - Link (2 bytes)
281 ; - CFA (2 bytes)
282 ; - rest
283
284 START_CODE:
285
286 ;FORTH_PROG_START: .db "   TEST2 0 >IN !  "
287
288 mem_LIT:     .dw itc_LIT
289 mem_ENTER:   .dw ENTER
290 mem_EXIT:    .dw EXIT
291 ;>python
292 addCompiledWord("LIT", "mem_LIT")
293 addCompiledWord("ENTER", "mem_ENTER")
294 addCompiledWord("EXIT", "mem_EXIT")
295 ;>endpy
296
297 ; Використовується для виклику повернення у асемблерний інтерпретатор
298 mem_RET:     .dw gethere
299 mmRET:       .dw #rammap(mem_RET)
300
301 ;;;;;; АСЕМБЛЕРНІ СЛОВА!!! ;;;;;;;;
302               ‘HEADER "+",      itc_PLUS,     mem_PLUS
303               ‘HEADER "-",      itc_MINUS,    mem_MINUS
304               ‘HEADER ">R",     itc_TO_R,     mem_TO_R
305               ‘HEADER "R>",     itc_R_FROM,   mem_R_FROM
306               ‘HEADER "DUP",    itc_DUP,      mem_DUP
307               ‘HEADER "DROP",   itc_DROP,     mem_DROP
308               ‘HEADER "SWAP",   itc_SWAP,     mem_SWAP
309               ‘HEADER "OVER",   itc_OVER,     mem_OVER
310               ‘HEADER "@",      itc_AT,       mem_AT
311               ‘HEADER "!",      itc_EXCLAM,   mem_EXCLAM
312               ‘HEADER "C!", itc_C_EXCLAM, mem_C_EXCLAM
313               ‘HEADER "C@", itc_C_AT, mem_C_AT
314               ‘HEADER "2*", itc_2STAR, mem_2STAR
```

```
315             ‘HEADER "2/", itc_2SLASH, mem_2SLASH
316             ‘HEADER "0",     itc_0,       mem_0
317             ‘HEADER "-1", itc_FFFF, mem_FFFF
318             ‘HEADER "CSWAP", itc_CSWAP,   mem_CSWAP
319             ‘HEADER "AND", itc_AND, mem_AND
320             ‘HEADER "XOR", itc_XOR, mem_XOR
321             ‘HEADER "OR", itc_OR, mem_OR
322             ‘HEADER "INVERT", itc_INVERT, mem_INVERT
323             ‘HEADER "=", itc_EQUAL, mem_EQUAL
324
325             ‘HEADER "TOKEN", itc_TOKEN, mem_TOKEN
326
327 ;;;;;;; Змінні ;;;;;;;;
328 _CP:    .dw 0
329         ‘HEADER "TIB", DOVAR, mem_TIB
330 TIB:    .dw 0
331         ‘HEADER "#TIB", DOVAR, mem_CTIB
332 CTIB:   .dw 0
333         ‘HEADER ">IN", DOVAR, mem__IN
334 _IN:    .dw 0
335         ‘HEADER "BASE", DOVAR, mem_BASE
336 BASE:   .dw 10
337         ‘HEADER "LATEST", DOVAR, mem_LATEST
338 LATEST: .dw 0
339         ‘HEADER "STATE", DOVAR, mem_STATE
340 STATE:  .dw 0
341
342
343
344 ;;;;;; ФОРТслова-!!! ;;;;;;;;;;
345
346         ‘HEADER "->LEDS", ENTER, mem_LEDS
347         ‘COMPILE_RAW "LIT"
348         .dw LEDs
349         ‘COMPILE_RAW "C! EXIT"
350
351         ‘HEADER "->IND", ENTER, mem_IND
352         ‘COMPILE_RAW "CSWAP LIT"
353         .dw INDICATOR
354         ‘COMPILE_RAW "! EXIT"
355
```

```
356         ‘HEADER "->INDL", ENTER, mem_INDL
357         ‘COMPILE_RAW "LIT"
358         .dw INDL
359         ‘COMPILE_RAW "C! EXIT"
360
361         ‘HEADER "->INDH", ENTER, mem_INDH
362         ‘COMPILE_RAW "LIT"
363         .dw INDH
364         ‘COMPILE_RAW "C! EXIT"
365
366         ‘HEADER "->INDDP", ENTER, mem_INDDP
367         ‘COMPILE_RAW "LIT"
368         .dw INDDP
369         ‘COMPILE_RAW "C! EXIT"
370
371         ‘HEADER "->portA", ENTER, mem_PORTA
372         ‘COMPILE_RAW "LIT"
373         .dw PA_REG
374         ‘COMPILE_RAW "C! EXIT"
375         ‘HEADER "->portB", ENTER, mem_PORTB
376         ‘COMPILE_RAW "LIT"
377         .dw PB_REG
378         ‘COMPILE_RAW "C! EXIT"
379         ‘HEADER "->portC", ENTER, mem_PORTC
380         ‘COMPILE_RAW "LIT"
381         .dw PC_REG
382         ‘COMPILE_RAW "C! EXIT"
383
384         ‘HEADER "1", ENTER, mem_1
385         ‘COMPILE_RAW "0 -1 - EXIT"
386
387         ‘HEADER "NEGATE", ENTER, mem_NEGATE
388         ‘COMPILE_RAW "INVERT 1 + EXIT"
389
390         ‘HEADER "CELL", ENTER, mem_CELL
391         ‘COMPILE_RAW "LIT 2 EXIT"
392
393         ‘HEADER "CELL+", ENTER, mem_CELLPLUS
394         ‘COMPILE_RAW "CELL + EXIT"
395
396         ‘HEADER "CELLS", ENTER, mem_CELLS
```

```
397           ‘COMPILE_RAW "2* EXIT"
398
399           ‘HEADER "HERE", ENTER, mem_HERE
400           ‘COMPILE_RAW "LIT "
401           .dw #rammap(_CP)
402           ‘COMPILE_RAW "@ EXIT"
403
404           ‘HEADER "ALLOT", ENTER, mem_ALLOT
405           ‘COMPILE_RAW "HERE + LIT"
406           .dw #rammap(_CP)
407           ‘COMPILE_RAW "! EXIT"
408
409           ‘HEADER ",", ENTER, mem_COMMA
410           ‘COMPILE_RAW "HERE ! CELL ALLOT EXIT"
411
412           ‘HEADER "ROT", ENTER, mem_ROT
413           ‘COMPILE_RAW ">R SWAP R> SWAP EXIT"
414
415           ‘HEADER_IMM "[", ENTER, mem_LEFTBRACKET ; interpret
416           ‘COMPILE_RAW "0 STATE ! EXIT"
417
418           ‘HEADER "]", ENTER, mem_RIGHTBRACKET ; compile
419           ‘COMPILE_RAW "1 STATE ! EXIT"
420
421
422
423
424           ‘HEADER "NOOP", ENTER, mem_NOOP
425           ‘COMPILE_RAW "EXIT"
426
427
428
429 FORTH_PROG_START:
430 ;     .db " 1                 "
431 ;     .db " DUP LEDS C! 2* "
432 ;     .db " DUP LEDS C! 2* "
433 ;     .db " DUP LEDS C! 2* "
434 ;     .db " DUP LEDS C! 2* "
435 ;     .db " DUP LEDS C! 2* "
436 ;     .db " DUP LEDS C! 2* "
437 ;     .db " DUP LEDS C! 2* "
```

```
438 ;    .db " DUP LEDS C! 2* "
439      .db " BASE @ "
440      .db "   DUP >INDL "
441      .db "   1 2* 2* BASE ! "
442      .db "   BASE @ >INDH "
443      .db " BASE !"
444      .db " 0 >IN  !    "
445
446 FORTH_PROG_END:   .db "  ", 0 , 0
447
448
449
450 END_CODE:
451
452 ;;;;;;; CORE ;;;;;;;
453 ; IP      - XH:XL
454 ; W+2     - YH:YL
455 ; DSP     - dspH:dspL
456
457 NEXT:
458      ld YL, X+
459      ld YH, X+
460      ld ZL, Y+
461      ld ZH, Y+
462      ijmp
463
464 ENTER:
465      push XL
466      push XH
467      movw XL, YL
468      rjmp NEXT
469
470 EXIT:
471      pop XH
472      pop XL
473      rjmp NEXT
474
475 mRET:
476      ret
477
478 ds_PUSH_ZZ:
```

```
479      movw ZL, dspL
480      st Z+, ZZL
481      st Z+, ZZH
482      movw dspL, ZL
483      ret
484
485  ds_POP_ZZ:
486      movw ZL, dspL
487      ld ZZH, -Z
488      ld ZZL, -Z
489      movw dspL, ZL
490      ret
491
492  DOVAR:
493      movw ZZL, YL
494      rcall ds_PUSH_ZZ
495      rjmp NEXT
496  ;;;;;;;;;;;;;;;;;;;;
497
498
499  itc_DUP:
500      rcall _DUP
501      rjmp NEXT
502  _DUP:
503      movw ZL, dspL
504      ld ZZH, -Z
505      ld ZZL, -Z
506      adiw ZL, 2
507      st Z+, ZZL
508      st Z+, ZZH
509      movw dspL, ZL
510      ret
511
512  itc_DROP:
513      rcall _DROP
514      rjmp NEXT
515  _DROP:
516      movw ZL, dspL
517      subi ZL, 2
518      sbci ZH, 0
519      movw dspL, ZL
```

```
520      ret
521
522  itc_OVER:
523      rcall _OVER
524      rjmp NEXT
525  _OVER:
526      movw ZL, dspL
527      subi ZL, 4
528      sbci ZH, 0
529      ld ZZH, Z+
530      ld ZZL, Z+
531      movw ZL, dspL
532      st Z+, ZZL
533      st Z+, ZZH
534      movw dspL, ZL
535      ret
536
537  itc_SWAP:
538      rcall _SWAP
539      rjmp NEXT
540  _SWAP:
541      movw ZL, dspL
542      ld tempH, -Z
543      ld tempL, -Z
544      ld ZZH, -Z
545      ld ZZL, -Z
546      st Z+, tempL
547      st Z+, tempH
548      st Z+, ZZL
549      st Z+, ZZH
550      movw dspL, ZL
551      ret
552
553  itc_CSWAP:
554      rcall _CSWAP
555      rjmp NEXT
556  _CSWAP:
557      movw ZL, dspL
558      ld tempH, -Z
559      ld tempL, -Z
560      st Z+, tempH
```

```
561      st Z+, tempL
562      movw dspL, ZL
563      ret
564
565  itc_ROT:
566      rcall _ROT
567      rjmp NEXT
568  _ROT:
569      rcall _SWAP
570      pop ZZH
571      pop ZZL
572      rcall ds_PUSH_ZZ
573      rcall _SWAP
574      ret
575
576  itc_TO_R:
577      rcall ds_POP_ZZ
578      push ZZH
579      push ZZL
580      rjmp NEXT
581  _TO_R:
582      rcall ds_POP_ZZ
583      pop tempH
584      pop tempL
585      push ZZH
586      push ZZL
587      push tempL
588      push tempH
589      ret
590
591  itc_R_FROM:
592      pop ZZL
593      pop ZZH
594      rcall ds_PUSH_ZZ
595      rjmp NEXT
596  _R_FROM:
597      pop tempH
598      pop tempL
599      pop ZZL
600      pop ZZH
601      push tempL
```

```
602      push tempH
603      rcall ds_PUSH_ZZ
604      ret
605
606  itc_R_AT:
607      pop ZZL
608      pop ZZH
609      push ZZH
610      push ZZL
611      rcall ds_PUSH_ZZ
612      rjmp NEXT
613  _R_AT:
614      pop tempH
615      pop tempL
616      pop ZZL
617      pop ZZH
618      push tempL
619      push tempH
620      rcall ds_PUSH_ZZ
621      push ZZH
622      push ZZL
623      ret
624
625  itc_EXCLAM:
626      rcall _EXCLAM
627      rjmp NEXT
628  _EXCLAM:
629      movw tempL, XL
630      movw XL, dspL
631      ld ZH, -X
632      ld ZL, -X
633      ld ZZH, -X
634      ld ZZL, -X
635      st Z+, ZZL
636      st Z+, ZZH
637      movw dspL, XL
638      movw XL, tempL
639      ret
640
641  itc_AT:
642      rcall _AT
```

```
643      rjmp NEXT
644  _AT:
645      rcall ds_POP_ZZ
646      movw ZL, ZZL
647      ld ZZL, Z+
648      ld ZZH, Z+
649      rcall ds_PUSH_ZZ
650      ret
651
652  itc_LIT:
653      rcall _LIT
654      rjmp NEXT
655  _LIT:
656      ld ZZL, X+
657      ld ZZH, X+
658      rcall ds_PUSH_ZZ
659      ret
660
661  itc_PLUS:
662      rcall _PLUS
663      rjmp NEXT
664  _PLUS:
665      rcall ds_POP_ZZ
666      mov temp2L, ZZL
667      mov temp2H, ZZH
668      rcall ds_POP_ZZ
669      add ZZL, temp2L
670      adc ZZH, temp2H
671      rcall ds_PUSH_ZZ
672
673      ret
674
675  itc_MINUS:
676      rcall _MINUS
677      rjmp NEXT
678  _MINUS:
679      rcall ds_POP_ZZ
680      mov temp2L, ZZL
681      mov temp2H, ZZH
682      rcall ds_POP_ZZ
683      sub ZZL, temp2L
```

```
684      sbc ZZH, temp2H
685      rcall ds_PUSH_ZZ
686      ret
687
688  itc_AND:
689      rcall _AND
690      rjmp NEXT
691  _AND:
692      rcall ds_POP_ZZ
693      mov temp2L, ZZL
694      mov temp2H, ZZH
695      rcall ds_POP_ZZ
696      and ZZL, temp2L
697      and ZZH, temp2H
698      rcall ds_PUSH_ZZ
699      ret
700
701  itc_OR:
702      rcall _OR
703      rjmp NEXT
704  _OR:
705      rcall ds_POP_ZZ
706      mov temp2L, ZZL
707      mov temp2H, ZZH
708      rcall ds_POP_ZZ
709      or ZZL, temp2L
710      or ZZH, temp2H
711      rcall ds_PUSH_ZZ
712      ret
713
714  itc_XOR:
715      rcall _XOR
716      rjmp NEXT
717  _XOR:
718      rcall ds_POP_ZZ
719      mov temp2L, ZZL
720      mov temp2H, ZZH
721      rcall ds_POP_ZZ
722      eor ZZL, temp2L
723      eor ZZH, temp2H
724      rcall ds_PUSH_ZZ
```

```
725     ret
726
727 itc_NEG:
728     rcall _NEG
729     rjmp NEXT
730 _NEG:
731     rcall _INV
732     ldi ZZL, 1
733     clr ZZH
734     rcall ds_PUSH_ZZ
735     rcall _PLUS
736     ret
737
738 itc_INV:
739     rcall _INV
740     rjmp NEXT
741 _INV:
742     rcall ds_POP_ZZ
743     com ZZL
744     com ZZH
745     rcall ds_PUSH_ZZ
746     ret
747
748
749
750 itc_HERE:
751     rcall _HERE
752     rjmp NEXT
753 _HERE:
754     ldi ZL, low(#rammap(_CP))
755     ldi ZH, high(#rammap(_CP))
756     ld ZZL, Z+
757     ld ZZH, Z+
758     rcall ds_PUSH_ZZ
759     ret
760
761 itc_ALLOT:
762     rcall _ALLOT
763     rjmp NEXT
764 _ALLOT:
765     ldi ZL, low(#rammap(_CP))
```

```
766     ldi ZH, high(#rammap(_CP))
767     ld ZZL, Z+
768     ld ZZH, Z+
769     rcall ds_PUSH_ZZ
770     rcall _PLUS
771     rcall ds_POP_ZZ
772     ldi ZL, low(#rammap(_CP))
773     ldi ZH, high(#rammap(_CP))
774     st Z+, ZZL
775     st Z+, ZZH
776     ret
777
778 itc_EXECUTE:
779     rcall _EXECUTE
780     rjmp NEXT
781 _EXECUTE:
782     rcall ds_POP_ZZ
783     movw YL, ZZL
784     ld ZL, Y+
785     ld ZH, Y+
786     ijmp
787     ret
788
789
790 itc_COLON:
791     rcall _COLON
792     rjmp NEXT
793 _COLON:
794     ret
795
796 itc_SEMICOLON:
797     rcall _SEMICOLON
798     rjmp NEXT
799 _SEMICOLON:
800     ret
801
802
803
804 itc_EQUAL:
805     rcall ds_POP_ZZ
806     mov temp2L, ZZL
```

```
807      mov temp2H, ZZH
808      rcall ds_POP_ZZ
809      cp ZZL, temp2L
810      cpc ZZH, temp2H
811      brne itc_0
812      rjmp itc_FFFF
813
814
815  itc_0:
816      clr ZZL
817      clr ZZH
818      rcall ds_PUSH_ZZ
819      rjmp NEXT
820
821  itc_FFFF:
822      ser ZZL
823      ser ZZH
824      rcall ds_PUSH_ZZ
825      rjmp NEXT
826
827  itc_C_AT:
828      rcall _C_AT
829      rjmp NEXT
830  _C_AT:
831      rcall ds_POP_ZZ
832      movw ZL, ZZL
833      ld ZZL, Z
834      clr ZZH
835      rcall ds_PUSH_ZZ
836      ret
837
838  itc_C_EXCLAM:
839      rcall _C_EXCLAM
840      rjmp NEXT
841  _C_EXCLAM:
842      movw tempL, XL
843      movw XL, dspL
844      ld ZH, -X
845      ld ZL, -X
846      ld ZZH, -X
847      ld ZZL, -X
```

```
848      st Z, ZZL
849      movw dspL, XL
850      movw XL, tempL
851      ret
852
853  itc_2STAR:
854      rcall _2STAR
855      rjmp NEXT
856  _2STAR:
857      rcall ds_POP_ZZ
858      lsl ZZL
859      rol ZZH
860      rcall ds_PUSH_ZZ
861      ret
862
863  itc_INVERT:
864      rcall _INVERT
865      rjmp NEXT
866  _INVERT:
867      rcall ds_POP_ZZ
868      com ZZL
869      com ZZH
870      rcall ds_PUSH_ZZ
871      ret
872
873  itc_2SLASH:
874      rcall _2SLASH
875      rjmp NEXT
876  _2SLASH:
877      rcall ds_POP_ZZ
878      asr ZZh
879      ror ZZl
880      rcall ds_PUSH_ZZ
881      ret
882
883  ;itc_ROT:
884      ;rcall ds_POP_ZZ
885      ;movw tempL, ZZL
886
887      ;movw temp0, tosl
888      ;ld temp2, Y+
```

```
889      ;ld temp3, Y+
890      ;loadtos
891
892      ;st -Y, temp3
893      ;st -Y, temp2
894      ;st -Y, temp1
895      ;st -Y, temp0
896
897      ;rjmp NEXT
898
899
900  ;>python
901  addMacro("load","""
902      ldi ZL, low(@1)
903      ldi ZH, high(@1)
904      ld @0L, Z+
905      ld @0H, Z+
906  """)
907  ;>endpy
908
909
910
911  ; Змінює >IN - пропускає прогалики і 0. Якщо кінець рядку, то >IN =
       #TIB
912  trailing:
913      `load source, #rammap(TIB)
914      `load count, #rammap(_IN)
915      `load temp, #rammap(CTIB)
916
917      add tempL, sourceL ; temp = TIB + CTIB    equ     end of
       line
918      adc tempH, sourceH
919
920      movw ZL, sourceL
921      add ZL, countL      ; Z = TIB + _IN     equ    current
922      adc ZH, countH
923
924      check_if_line_ended:
925          cp ZL, tempL
926          ; brne is_current_char_space
927          cpc ZH, tempH
```

```
928           brne is_current_char_space
929           ; oh no! line ended!
930           rjmp finish_trailing
931       is_current_char_space:
932           ld temp, Z+
933           cpi temp, 32
934           breq check_if_line_ended
935           cpi temp, 0
936           breq check_if_line_ended
937           ; current char is not space!
938           sbiw ZL, 1
939     finish_trailing:
940       sub ZL, sourceL
941       sbc ZH, sourceH
942       movw countL, ZL
943
944       ldi ZL, low(#rammap(_IN))
945       ldi ZH, high(#rammap(_IN))
946       st Z+, countL
947       st Z+, countH
948
949       ret
950
951   itc_TOKEN:
952       `load temp, #rammap(_CP)
953       subi tempL, -100
954       sbci tempH, 0
955       movw destL, tempL
956       rcall word
957       `load ZZ, #rammap(_CP)
958       subi ZZL, -100
959       sbci ZZH, 0
960       rcall ds_PUSH_ZZ
961       rjmp NEXT
962   word:
963       rcall trailing
964       push XL
965       push XH
966
967       `load source, #rammap(TIB)
968       `load count, #rammap(_IN)
```

```
969       `load temp, #rammap(CTIB)
970
971       add tempL, sourceL
972       adc tempH, sourceH
973
974       movw ZL, sourceL
975       add ZL, countL
976       adc ZH, countH
977
978       movw XL, destL
979       adiw XL, 1
980
981       check_if_line_ended_word:
982           cp ZL, tempL
983            ;brne is_current_char_space_word
984           cpc ZH, tempH
985            brne is_current_char_space_word
986           ; oh no! line ended!
987           rjmp finish_word
988       is_current_char_space_word:
989           ld temp, Z+
990           cpi temp, 32
991            breq endd
992           cpi temp, 0
993            breq endd
994           st X+, temp
995           rjmp check_if_line_ended_word
996       endd:
997           ; current char is not space!
998           sbiw XL, 1
999           sbiw ZL, 1
1000    finish_word:
1001      sub ZL, sourceL
1002      sbc ZH, sourceH
1003      movw countL, ZL
1004      sub XL, destL
1005      sbc XH, destH
1006
1007      movw ZL, destL
1008      st Z+, XL
1009
```

```
1010      ldi ZL, low(#rammap(_IN))
1011      ldi ZH, high(#rammap(_IN))
1012      st Z+, countL
1013      st Z+, countH
1014      pop XH
1015      pop XL
1016      ret
1017
1018  interpret:
1019      rcall trailing
1020
1021      ‘load count, #rammap(_IN)
1022      ‘load temp, #rammap(CTIB)
1023
1024      cp tempL, countL
1025       brne noteol
1026      cp tempH, countH
1027       brne noteol
1028
1029      ;rcall USART_SendACK
1030      rjmp UART_WAIT ; EOL, wait for input!
1031      looop:
1032          ldi temp, 0xFF
1033          sts LEDs, temp
1034          rjmp looop ; EOL!!!
1035
1036     noteol:
1037      ‘load dest, #rammap(_CP)
1038      rcall word
1039
1040      push XH
1041      push XL
1042          rcall find
1043      pop XL
1044      pop XH
1045
1046      rcall ds_POP_ZZ
1047      ;rjmp intepret_execute
1048
1049      movw ZL, ZZL
1050      sbiw ZL, 3
```

```
1051      ld temp, Z
1052      andi temp, 0x80
1053      cpi temp, 0
1054        brne intepret_execute
1055
1056      ldi ZL, low(#rammap(STATE))
1057      ldi ZH, high(#rammap(STATE))
1058      ld temp, Z
1059      cpi temp, 0
1060        breq intepret_execute
1061
1062  interpret_compile:
1063      rcall ds_PUSH_ZZ
1064      ldi ZZL, low(#rammap(mem_COMMA))
1065      ldi ZZH, high(#rammap(mem_COMMA))
1066
1067  intepret_execute:
1068      ; EXECUTE
1069      ldi XL, low(#rammap(mmRET))
1070      ldi XH, high(#rammap(mmRET))
1071      movw YL, ZZL
1072      ld ZL, Y+
1073      ld ZH, Y+
1074      ijmp
1075
1076  gethere:
1077      rjmp interpret
1078
1079
1080  ;source, dest (LINK)
1081  compare:
1082      movw XL, sourceL
1083      movw ZL, destL
1084      ld temp2, X
1085      inc temp2
1086      compare_each:
1087          ld tempL, X+
1088          ld tempH, -Z
1089          andi tempH, 0b01111111
1090          cp tempL, tempH
1091            brne notequal
```

```
1092          dec temp2
1093          cpi temp2, 0
1094           breq equal
1095      rjmp compare_each
1096
1097
1098  find:
1099      lds ZL, #rammap(LATEST)
1100      lds ZH, #rammap(LATEST)+1
1101      lds sourceL, #rammap(_CP)
1102      lds sourceH, #rammap(_CP)+1
1103      cycle:
1104          movw ZZL, ZL
1105          ld temp2H, -Z
1106          ld temp2L, -Z
1107          movw destL, ZL
1108          rjmp compare
1109      equal:
1110          ; ZZL - CFA of found word
1111          rcall ds_PUSH_ZZ
1112          ret
1113      notequal:
1114          cpi temp2L, 0
1115           brne continue
1116          cp temp2L, temp2H
1117           brne continue
1118          rjmp notfound
1119      continue:
1120          movw ZL, temp2L
1121          rjmp cycle
1122          ret
1123
1124
1125  notfound:
1126          ldi temp, 0x82
1127          sts LEDs, temp
1128          rjmp notfound
```

Перелік посилань

1. Баранов С. Н., Ноздрунов Н. Р. Язык Форт и его реализации. — М.: Наука и техника, 1989. — 108 с.
2. Семенов Ю. А.. Программирование на языке Форт. — М.: Наука и техника, 1992. — 75 с.
3. Leo Brodie. Starting Forth. — S.: Forth FIG Group, 1981. — 208 с.
4. Leo Brodie. Thinking Forth. — S.: Forth FIG Group, 1985. — 420 с.
5. Мікроконтролери AVR. Коппект лекцій. Укладач — Новацький О.А. М.:НТУУ «КПІ»
6. Brad Rodriguez. Porting Forth. Serie of web-published articles. 1993.