Обосновка на прехода от x86 към ARM за дисциплината «Микропроцесори» на специалност «СИТ»

от гл. ас. д-р инж. Лъчезар Георгиев, катедра КНТ, ФИТА, ТУ–Варна

Колеги, преди повече от 10 години ние, преподавателите по дисциплината «МПТ», проведохме паметна среща, на която след оживена дискусия надделя мнението, че е време да преминем от 8-битовия микропроцесор «СМ601» (MC6800), който се изучаваше дотогава, към 16-битовия I8086/8088. Който е присъствал на тази среща, сигурно си я спомня. Дошло е времето отново да поставим въпроса за промяна на изучавания микропроцесор, този път от 16-битов към 32-битов. Той може да бъде или 80x86, или ARM. Отдавна колегата Георги Върбанов (вече пенсионер) предлага да изберем ARM. Тази година това предложи и водещият дисциплините «МПТ» и «Микропроцесори» доц. д-р Тодор Ганчев. След като се запознах с особеностите на семейството микропроцесори ARM, това предложение подкрепих и аз. В следващите две страници ще се опитам да обоснова това с примери и факти...

```
; Върни в EAX най-малкото общо кратно (НОК) на числата в EAX и EDX

global  _lcmult
_lcmult:
        MOV     EBX,EAX  ; съхрани числото A
        MOV     ECX,EDX  ; съхрани числото B
        MUL     EDX      ; EDX:EAX = A * B
        XCHG    EAX,EBX  ; EAX = A, EBX = младша_дума(A * B)
        XCHG    EDX,ECX  ; EDX = B, ECX = старша_дума(A * B)
        CALL    GCD      ; намери НОД(A, B)
        XCHG    EAX,EBX  ; EAX = младша_дума(A * B), EBX = НОД
        MOV     EDX,ECX  ; EDX = старша_дума(A * B)
        DIV     EBX      ; EAX = A * B / НОД(A, B) = НОК
        RET              ; (при деленето е възможно препълване, нищо че НОД > 0)
GCD:
%include "gcdivis.nas"
```

```
// Върни в R0 най-малкото общо кратно (НОК) на числата в R0 и R1
// (произведението им трябва да е до 32 бита поради ограничението на
// командата UDIV; ARMv7 няма команда за делене с 64-битово делимо).

.global lcmult
lcmult:
        STR     LR,[SP,#-4]!//Съхрани LR, защото BL ще го промени!
        MUL     R2,R1,R0  // R2 = A * B (няма смисъл от UMULL, защото няма UDIVL)
        BL      GCD       // намери НОД(A, B)
        UDIV    R0,R2,R0  // R0 = A * B / НОД(A, B) = НОК
        LDR     PC,[SP],#4//Върни се в извикващата програма
GCD:
.include "gcdivis.s"
```

```
; Върни в EAX най-големия общ делител на числата в EAX и EDX
; (алгоритъм - Евклид от Александрия, реализация: Paul Hsieh).

        NEG     EAX             ; unsigned gcd(unsigned a, unsigned b)
        JE      L3              ; {
L1:     NEG     EAX             ;       if (a == 0 && b == 0) return 1;
        XCHG    EAX,EDX         ;\      if (b == 0) return a;
L2:     SUB     EAX,EDX         ; \     if (a != 0)
        JA      L2              ;  \_                   while (a != b)
        JB      L1              ;                               if (a < b)
L3:     ADD     EAX,EDX         ;\                                      b -= a;
        JNE     L4              ; \_                            else a -= b;
        INC     EAX             ;       return b;
L4:     RET                     ; }
```

```
// Върни в R0 най-големия общ делител на числата в R0 и R1
// (алгоритъм - Евклид от Александрия).

        ORRS    R3,R0,R1        //unsigned gcd(unsigned a, unsigned b)
        MOVEQ   R0,#1           //{
        CMPNE   R1,#0           //\__   if (a == 0 && b == 0) return 1;
        BXEQ    LR              //      if (b == 0) return a;
        CMP     R0,#0           //      if (a != 0)
        MOVEQ   R0,R1           //   ___ while (a != b)
L1:     CMP     R0,R1           //___/            if (a < b)
        SUBLO   R1,R1,R0        //                      b -= a;
        SUBHI   R0,R0,R1        //              else a -= b;
        BNE     L1              //      return b;
        BX      LR              //}
```

```
;unsigned getVLQ(unsigned char **src)
;{                              // прочети величина с променлива дължина
;   unsigned char c, *p = *src; // (вж. Standard MIDI-File Format Spec. 1.1)
;   unsigned value = *p++;
;   if (value & 0x80) {
;               value &= 0x7f;
;               do
;                  value = (value << 7) + ((c = *p++) & 0x7f);
;               while (c & 0x80);
;   }
;   *src = p;
;   return value;
;}

global  _getvlq                 ; в GCC декларирай с __attribute__((fastcall))
_getvlq:
        PUSH    ESI             ; src = ECX, c = AL, p = ESI
        XOR     EDX,EDX         ; value се съставя в EDX
        XOR     EAX,EAX         ; старшите 3 байта трябва да бъдат 0 в началото
        MOV     ESI,[ECX]       ; получи указателя към потока от данни
        JMP     SHORT L2        ; премини към получаване на първия байт
L1:     AND     AL,7FH          ; маскирай бит 7, остави значещите битове 0-6
        OR      DL,AL           ; „сгломи“ най-младшия засега байт
        SHL     EDX,7           ; value <<= 7
L2:     LODSB                   ; получи следващия байт и обнови адреса
        TEST    AL,AL           ; има ли още байтове?
        JS      L1              ; да, продължи
        OR      EAX,EDX         ; не, върни стойността в EAX
        MOV     [ECX],ESI       ; съхрани указателя към следващите данни
        POP     ESI             ; възстанови ESI (CGG изисква той да се запази)
        RET
```

```
/*
unsigned getVLQ(unsigned char **src)
{                               // прочети величина с променлива дължина
  unsigned char c, *p = *src;   // (вж. Standard MIDI-File Format Spec. 1.1)
  unsigned value = *p++;
  if (value & 0x80) {
        value &= 0x7f;
        do
          value = (value << 7) + ((c = *p++) & 0x7f);
        while (c & 0x80);
  }
  *src = p;
  return value;
}
*/

.global  getvlq
getvlq:
        STMFD   SP!,{R4,LR}     //Съхрани използваните регистри (ARM IHI 0042F)
        LDR     R4,[R0]         // src = R0, c = R1, p = R4
        MOV     R2,#0           // value ще се натрупва в R2
L1:     LDRB    R1,[R4],#1      // получи следващия байт и обнови адреса
        AND     R3,R1,#0x7F     // маскирай бит 7, остави значещите битове 0-6
        ORR     R2,R3,R2,LSL #7 //value <<= 7, „сгломи“ най-младшия засега байт
        CMP     R1,#0x80        // има ли още байтове?
        BHS     L1              // да, продължи
        STR     R4,[R0]         // не, съхрани указателя към следващите данни
        MOV     R0,R2           // върни стойността в R0
        LDMFD   SP!,{R4,PC}
```

```
;unsigned char *putVLQ(unsigned value, unsigned char *dest) // запиши ВПД
;{                              // (вж. Standard MIDI-File Format Spec. 1.1)
;   unsigned buffer = value & 0x7f;
;   while ((value >>= 7) > 0) {
;               buffer <<= 8;
;               buffer |= 0x80;
;               buffer += value & 0x7f;
;   }
;   for ( ; ; ) {
;               *dest++ = (unsigned char)buffer;
;               if (buffer & 0x80)
;                  buffer >>= 8;
;               else
;                  return dest;
;   }
;}

global  _putvlq                 ;void __attribute__((fastcall)) putvlq(...) {
_putvlq:                        ;//value се предава в ECX, dest - в EDX
        MOV     EAX,ECX         ;//buffer е удобно да бъде в EAX заради STOSB
        XCHG    EDX,EDI         ;//съхрани EDI, STOSB изисква dest да е в EDI
        AND     EAX,7FH         ;   buffer = value & 0x7f;
L1:     SHR     ECX,7           ;   while ((value >>= 7) > 0)
        JZ      L2              ;   {
        SHL     EAX,8           ;               buffer <<= 8;
        OR      AL,80H          ;               buffer |= 0x80;
        OR      AL,CL           ;               buffer |= value & 0xFF;
        JMP     L1              ;   } for ( ; ; ) {
L2:     STOSB                   ;               *dest++ = (unsigned char)buffer;
        OR      AL,AL           ;               if (buffer & 0x80)
        JNS     L3              ;               {
        SHR     EAX,8           ;                       buffer >>= 8;
        JMP     L2              ;               }
L3:     MOV     EAX,EDI         ;               else return dest;
        MOV     EDI,EDX         ;   }
        RET                     ;}
```

```
/*
unsigned char *putVLQ(unsigned value, unsigned char *dest) // запиши ВПД
{                               // (вж. Standard MIDI-File Format Spec. 1.1)
  unsigned buffer = value & 0x7f;
  while ((value >>= 7) > 0) {
        buffer <<= 8;
        buffer |= 0x80;
        buffer += value & 0x7f;
  }
  for ( ; ; ) {
        *dest++ = (unsigned char)buffer;
        if (buffer & 0x80)
                buffer >>= 8;
        else
                return dest;
  }
}
*/

.global  putvlq                 // Регистри: value = R0, dest = R1, buffer = R2
putvlq:                         //putVLQ(unsigned value, unsigned char *dest) {
        AND     R2,R0,#0x7F     //      unsigned buffer = value & 0x7f;
L1:     MOVS    R0,R0,LSR #7    //      while ((value >>= 7) > 0) {
        MOVNE   R2,R2,LSL #8    //              buffer <<= 8;
        ORRNE   R2,R2,#0x80     //              buffer |= 0x80;
        ANDNE   R3,R0,#0x7F     //              buffer += value & 0x7f;
        ADDNE   R2,R2,R3        //      _}
        BNE     L1              //__/ for ( ; ; ) {
L2:     STRB    R2,[R1],#1      //              *dest++ = (unsigned char)buffer;
        TST     R2,#0x80        //              if (buffer & 0x80)
        MOVNE   R2,R2,LSR #8    //                      buffer >>= 8;
        BNE     L2              //              else return;
        MOV     R0,R1           //      }
        BX      LR              //}
```

На горните три примера са дадени **оптимизирани** линейни и циклично-разклонени програми – вляво във вариант за 80x86, а вдясно – за ARM. Архитектурата ARM е RISC и е създадена по-късно от x86, която е CISC. Шест важни фактора правят програмите за ARM по-кратки от тези за 80x86:

1. Наличието на *13 регистъра* за обща употреба срещу 7 за 80x86.
2. Наличието на *3 до 4 операнда* на команда при аритметично-логическите операции срещу 2 за 80x86 и дори само 1 за умножението и деленето (наистина командата *IMUL* (80186+) има 3-операнден вариант, а някои нови FMA4- и XOP-команди имат до 5 операнда, но са твърде сложни).
3. Възможността всяка команда да бъде направена *условна*.
4. Възможността за избор дали командата да *променя флаговете* или не.
5. Възможността да се работи с *изместено* копие на десния операнд.
6. *Ортогоналният* набор от команди и адресни режими (на 80x86 е неортогонален).

На следващия пример е дадена **оптимизирана** линейна програма (вляво за 80x86, вдясно за ARM), на чийто размер те влияят особено сериозно, а на последния – **оптимизирана** циклична програма за x86 (вляво) и съответна линейна за ARM (вдясно), възможна благодарение на командата *RBIT*. Считам, че тези фактори и особено последният ще улеснят значително изучаването на езика Асемблер от студентите, ако преминем от x86 към ARM. За това ще спомогне и краткият, но изчерпателен справочник за командите и адресните режими на ARM, чиято версия от 1999 г. е само 3 страници. Версията от 2003 г. вече е 6 страници, но по-новите команди там се използват рядко и можем да се ограничим до 56-те, налични до 1999 г. плюс *RBIT* и *UDIV*, без тези за работа с копроцесор (за сравнение, микропроцесорът I8086 има 80 команди, ако броим всички условни преходи за една команда *Jcc*).

```
; 64-битовите числа без знак в регистри EDX:ECX (Y1:Y0) и EBX:EAX (Z1:Z0) да се
; умножат и 128-битовият резултат да се върне в рег. EDX:ECX:EBX:EAX (A:B:C:D).
; Алгоритъм: Peter Norton, "Advanced Assembly Language", 1991, стр. 229-230

global  _um64x64
_um64x64:
        PUSH    ESI
        PUSH    EDI
        PUSH    EBP
        MOV     ESI,EAX ; Z0
        MOV     EDI,EDX ; Y1
        PUSH    EDX     ; съхрани Y1
        MUL     ECX     ; Z0 * Y0
        XCHG    EAX,ESI ; ESI = D = мл.д.(Z0 * Y0), EAX = Z0
        XCHG    EDX,EDI ; EDI = C = ст.д.(Z0 * Y0), EDX = Y1
        MUL     EDX     ; Z0 * Y1
        XOR     EBP,EBP ; A = 0
        ADD     EDI,EAX ; C = ст.д.(Z0 * Y0) + мл.д.(Z0 * Y1)
        ADC     EDX,EBP ; ст.д.(Z0 * Y1)
        ADC     EBP,EBP ; A
        XCHG    EDX,ECX ; ECX = B = ст.д.(Z0 * Y1), EDX = Y0
        MOV     EAX,EBX ; Z1
        MUL     EDX     ; Z1 * Y0
        ADD     EDI,EAX ; C = ст.д.(Z0 * Y0) + мл.д.(Z0 * Y1) + мл.д.(Z1 * Y0)
        ADC     ECX,EDX ; B = ст.д.(Z0 * Y1) + ст.д.(Z1 * Y0)
        ADC     EBP,0   ; A
        XCHG    EAX,EDI ; EAX = C
        XCHG    EAX,EBX ; EAX = Z1, EBX = C
        POP     EDX     ; възстанови Y1
        MUL     EDX     ; Z1 * Y1
        ADD     ECX,EAX ; B = ст.д.(Z0 * Y1) + ст.д.(Z1 * Y0) + мл.д.(Z1 * Y1)
        ADC     EDX,EBP ; A = ст.д.(Z1 * Y1)
        XCHG    EAX,ESI ; EAX = D
        POP     EBP
        POP     EDI
        POP     ESI
        RET
```

```
// 64-битовите числа без знак в регистри R3:R2 (Y1:Y0) и R1:R0 (Z1:Z0) да се
// умножат и 128-битовият резултат да се върне в рег. R3:R2:R1:R0 (A:B:C:D).
// Алгоритъм: Peter Norton, "Advanced Assembly Language", 1991, стр. 229-230

.global  um64x64
um64x64:
        STMFD   SP!,{R4-R9,LR}  //Съхрани използваните регистри (ARM IHI 0042F)
        UMULL   R6,R9,R0,R3     // R9:R6 = Z0 * Y1
        UMULL   R0,R5,R2,R0     // R5:R0 = C:D = Z0 * Y0
        MOV     R7,#0           // A = 0
        ADDS    R5,R5,R6        // C = ст.д.(Z0 * Y0) + мл.д.(Z0 * Y1)
        ADCS    R9,R9,R7        // ст.д.(Z0 * Y1)
        ADC     R7,R7,R7        // A
        UMULL   R8,R6,R1,R2     // R6:R8 = Z1 * Y0
        UMULL   R4,R3,R1,R3     // R3:R4 = Z1 * Y1
        ADDS    R1,R5,R8        // C = ст.д.(Z0 * Y0) + мл.д.(Z0 * Y1) + мл.д.(Z1 * Y0)
        ADCS    R9,R9,R6        // B = ст.д.(Z0 * Y1) + ст.д.(Z1 * Y0)
        ADC     R7,R7,#0        // A
        ADDS    R2,R9,R4        // B = ст.д.(Z0 * Y1) + ст.д.(Z1 * Y0) + мл.д.(Z1 * Y1)
        ADC     R3,R3,R7        // A = ст.д.(Z1 * Y1)
        LDMFD   SP!,{R4-R9,PC}  // Възстанови регистрите и върни управлението
```

```
; Да се напише подпрограма, която да обръща огледално битовете на двойната
; дума в регистри EDX:ECX.

global  _bitrev64           ; в GCC декларирай с __attribute__((fastcall))
_bitrev64:                  ; младшата дума се предава в ECX, старшата - в EDX
        XCHG    EAX,ECX ; Освободи ECX
        MOV     ECX,32  ; Инициализирай го като брояч на битове
        SHL     EDX,1   ; Изведи най-старшия бит в CF
L1:     RCR     EAX,1   ; Въведи най-старшия бит от CF, изведи там най-младшия
        RCL     EDX,1   ; Въведи най-младшия бит от CF, изведи там най-старшия
        LOOP    L1      ; Следваща итерация на цикъла
        RET
```

```
// Да се напише подпрограма, която да обръща огледално битовете на двойната
// дума в регистри R1:R0.

.global  bitrev64
bitrev64:
        MOV     R2,R1
        RBIT    R1,R0
        RBIT    R0,R2
        BX      LR
```

Винаги съм предпочитал микропроцесорите с ортогонален набор от команди. Прекрасен пример за това е семейството M68K, което, уви, отдавна не се произвежда. За радост, ARM също има ортогонален набор от команди и за разлика от M68K, става все по-актуален. Всеки студент вече има такъв микропроцесор в джоба си в буквалния смисъл на думата. Нанокомпютърът «Orange Pi PC», на който ще се водят упражненията, струва само 44 лв. и има 4-ядрен микропроцесор H3 (Cortex-A7) с работна честота 1,6 GHz и 1 GB RAM, което позволява работата с операционната система «Linux» на работни станции тип «thin client» вместо PC. У нас се продава и по-мощният нанокомпютър «ODROID-XU4» с 8-ядрен микропроцесор Exynos 5422 (Cortex-A15/A7) и 2 GB RAM, който струва 189 лв., а и още доста видове нанокомпютри, някои от които произвежда пловдивската фирма «Олимекс».

През месец февруари миналата година, когато изработвахме новите учебни планове, настоях да се запази дисциплината «МПТ» за специалност «СИТ», защото в нито една друга дисциплина не се изучава Асемблер, а студентите не познават дори целия език C и не могат да програмират на ниско ниво. Тъй като в тази специалност не се изучава дисциплина «МПС», доц. Рускова предложи тя да се преименува на «Микропроцесори», с което аз се съгласих. От тази дисциплина не зависи никоя друга и затова спокойно можем да сменим изучавания микропроцесор, без промяната да се отрази на каквото и да е. Освен това лабораторните упражнения по нея се водят само от мен и тази промяна няма да повлияе на колегите, които водят «МПТ» на специалност «КСТ». Така дисциплината «Микропроцесори» ще се превърне в пилотна за проверка на правилността на предлаганата промяна. След време тя може да се направи и за специалност «КСТ». В интерес на истината, фирмата «Intel» няма намерение да се предава и затова се очертава много дълго противоборство между архитектурите x86 и ARM при всички видове компютри без джобните и нанокомпютрите, където ARM вече е победила. Затова мисля, че ще бъде добре на специалност «КСТ» да продължи да се преподава и по-старата архитектура 80x86 под някаква форма.

Предлагам следната тематика на ЛУ, основана на старата учебна програма, но с ударение върху побитови операции и работа с масиви и низове:

1. Запознаване с нанокомпютъра, ОС *Linux*, редактора *nano* и програмата *make*. Побитови операции в C и C++. Bitset. Двоичен вход/изход на C++.
2. Програмен модел и таблица с команди и адресни режими на ARM. «Здравей, свят!» на Асемблер. Настройка и оживяване на програмата с *gdb*.
3. Команди за прехвърляне на данни. Аритметични команди. Линейни програми на Асемблер. 64-битово умножение с резултат 128 бита без знак.
4. Команди за предаване на управлението. Условни и безусловни преходи. Разклонени програми. Изчисляване на модула на 64-битова разлика.
5. Циклични програми. Изчисляване на НОК и НОД на две 32-битови числа. Изчисляване на цялата част на квадратния корен на 64-битово число.
6. Команди за манипулация на битове. Огледално обръщане на 64-битово число. Пресмятане броя на съвпадащите битове в три 32-битови числа.
7. Първа контролна работа.
8. Команди за работа с паметта. Методи за адресация. Операции върху символни низове. Анализ на некоментирана програма за обработка на низ.
9. Аритметични задачи. Задачи за работа с масиви. Запис на 6 числа на Силвестър в масив. Последователно търсене на 64-битово число в масив.
10. Задачи за работа с масиви – част II. Сортиране във възходящ ред на масив по метода на вмъкването и двоично търсене в сортиран масив.
11. Задачи за работа с масиви – част III. Сортиране във възходящ ред на масив по метода «пирамидално сортиране». Бързодействие на метода.
12. Задачи за работа с низове от байтове. Четене от и запис в низ от байтове на 28-битови величини с променлива дължина.
13. Втора контролна работа.

14. Преобразуване от символен низ в десетична бройна система в число и обратно. Четене от и запис в низ на 32-битов Интернет-адрес (IPv4).
15. Преобразуване от символен низ в шестнадесетична бройна система в число и обратно. Четене от и запис в низ на 48-битов MAC-адрес.

Програмите на Асемблер без «Здравей, свят» имат тестови програми на C. Предаването на параметри към програмите на Асемблер се вижда в *gdb*. То става в регистри R0–R3 съгласно «Procedure Call Standard for the ARM Architecture» (ARM IHI 0042F). Има и много контролно-изпитни задачи.

Списък от заглавия, от които могат да се изберат подходящи за раздела «Литература» на учебната програма, е даден тук.

Лъчезар Георгиев,
юли/август 2017 г.

П.П. След като написах горното, прочетох следното в книгата на доц. Лари Пайет «Modern Assembly Language Programming with the ARM Processor»:

«Traditionally, assembly language courses have consisted of a mechanistic learning of a set of instructions, registers, and syntax. Partially because of this approach, over the years, assembly language courses have been marginalized in, or removed altogether from, many CS and CE curricula. The author feels that this is unfortunate, because a solid understanding of assembly language leads to better understanding of higher-level languages, compilers, interpreters, architecture, operating systems, and other important CS an CE concepts.

[…]

Because of their ubiquity, x86 based systems have been the platforms of choice for most assembly language courses over the last two decades. **The author believes that this is unfortunate, because in every respect other than ubiquity, the x86 architecture is the worst possible choice for learning and teaching assembly language.** The newer chips in the family have hundreds of instructions, and irregular rules govern how those instructions can be used. In an attempt to make it possible for students to succeed, typical courses use antiquated assemblers and interface with the antiquated IBM PC BIOS, using only a small subset of the modern x86 instruction set. The programming environment has little or no relevance to modern computing.

Partially because of this tendency to use x86 platforms, and the resulting unnecessary burden placed on students and instructors, as well as the reliance on antiquated and irrelevant development environments, assembly language is often viewed by students as very difficult and lacking in value. The author hopes that this textbook helps students to realize the value of knowing assembly language. The relatively simple ARM processor family was chosen in hopes that the students also learn that although assembly language programming may be more difficult than high-level languages, it can be mastered.

The recent development of very low-cost ARM based Linux computers has caused a surge of interest in the ARM architecture as an alternative to the x86 architecture, which has become increasingly complex over the years. This book should provide a solution for a growing need.

Many students have difficulty with the concept that a register can hold variable x at one point in the program, and hold variable y at some other point. They also often have difficulty with the concept that, before it can be involved in any computation, data has to be moved from memory into the CPU. Using a load-store architecture helps the students to more readily grasp these concepts.

Another common difficulty that students have is in relating the concepts of an address and a pointer variable. You can almost see the little light bulbs light up over their heads, when they have the “eureka!” moment and realize that pointers are just variables that hold an address. The author hopes that the approach taken in this book will make it easier for students to have that “eureka!” moment. The author believes that load-store architectures make that realization easier.

Many students also struggle with the concept of recursion, regardless of what language is used. In assembly, the mechanisms involved are exposed and directly manipulated by the programmer. Examples of recursion are scattered throughout this textbook. Again, the clean architecture of the ARM makes it much easier for the students to understand what is going on.

Some students have difficulty understanding the flow of a program, and tend to put many unnecessary branches into their code. Many assembly language courses spend so much time and space on learning the instruction set that they never have time to teach good programming practices. This textbook puts strong emphasis on using structured programming concepts. The relative simplicity of the ARM architecture makes this possible.

One of the major reasons to learn and use assembly language is that it allows the programmer to create very efficient mathematical routines. The concepts introduced in this book will enable students to perform efficient non-integral math on any processor. These techniques are rarely taught because of the time that it takes to cover the x86 instruction set. With the ARM processor, less time is spent on the instruction set, and more time can be spent teaching how to optimize the code.

The combination of the ARM processor and the Linux operating system provides the least costly hardware platform and development environment available. A cluster of 10 Raspberry Pis, or similar hosts, with power supplies and networking, can be assembled for 500 US dollars or less. This cluster can support up to 50 students logging in through ssh.

[...]

The main objective of this book is to provide an improved course in assembly language by replacing the x86 platform with one that is less costly, more ubiquitous, well-designed, powerful, and easier to learn.»

(стр. xxi–xxiii)

«When learning assembly language, the specific instruction set is not critically important, because what is really being learned is the fine detail of how a typical stored-program machine uses different storage locations and logic operations to convert a string of bits into a meaningful calculation. However, when it comes to learning assembly languages, some processors make it more difficult than it needs to be. Because some processors have an instruction set that is extremely irregular, non-orthogonal, large, and poorly designed, they are not a good choice for learning assembly. **The author feels that teaching students their first assembly language on one of those processors should be considered a crime, or at least a form of mental abuse.** Luckily, there are processors that are readily available, low-cost, and relatively easy to learn assembly with. This book uses one of them as the model for assembly language.»

(стр. 7–8)

(Подчертаванията в цитирания по-горе текст са мои.)