powerunsigned.c (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* powerunsigned.c                                                    */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*--------------------------------------------------------------------*/
 9:
10: static unsigned long ulBase;       /* Bad style. */
11: static unsigned long ulExp;        /* Bad style. */
12: static unsigned long ulPower = 1;  /* Bad style. */
13: static unsigned long ulIndex;      /* Bad style. */
14:
15: /*--------------------------------------------------------------------*/
16:
17: /* Read a non-negative base and exponent from stdin.  Write base
18:    raised to the exponent power to stdout.  Return 0. */
19:
20: int main(void)
21: {
22:    printf("Enter the base:  ");
23:    scanf("%lu", &ulBase);   /* Should validate. */
24:
25:    printf("Enter the exponent:  ");
26:    scanf("%lu", &ulExp);    /* Should validate. */
27:
28:    for (ulIndex = 1; ulIndex <= ulExp; ulIndex++)
29:       ulPower *= ulBase;
30:
31:    printf("%lu to the %lu power is %lu.\n", ulBase, ulExp, ulPower);
32:
33:    return 0;
34: }
```

powerunsignedflat.c (Page 1 of 1)

```
 1: /*--------------------------------------------------------------------*/
 2: /* powerunsignedflat.c                                                */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*--------------------------------------------------------------------*/
 9:
10: static unsigned long ulBase;       /* Bad style. */
11: static unsigned long ulExp;        /* Bad style. */
12: static unsigned long ulPower = 1;  /* Bad style. */
13: static unsigned long ulIndex;      /* Bad style. */
14:
15: /*--------------------------------------------------------------------*/
16:
17: /* Read a non-negative base and exponent from stdin.  Write base
18:    raised to the exponent power to stdout.  Return 0. */
19:
20: int main(void)
21: {
22:    printf("Enter the base:  ");
23:    scanf("%lu", &ulBase);   /* Should validate. */
24:
25:    printf("Enter the exponent:  ");
26:    scanf("%lu", &ulExp);    /* Should validate. */
27:
28:    ulIndex = 1;
29: powerLoop:
30:    if (ulIndex > ulExp) goto powerLoopEnd;
31:    ulPower *= ulBase;
32:    ulIndex++;
33:    goto powerLoop;
34: powerLoopEnd:
35:
36:    printf("%lu to the %lu power is %lu.\n", ulBase, ulExp, ulPower);
37:
38:    return 0;
39: }
```

powerunsigned.s (Page 1 of 3)

```
    1:          //-------------------------------------------------------------
    2:          // powerunsigned.s
    3:          // Author: Bob Dondero and William Ughetta
    4:          //-------------------------------------------------------------
    5:
    6:          .section .rodata
    7:
    8: basePromptStr:
    9:          .string "Enter the base:  "
   10:
   11: expPromptStr:
   12:          .string "Enter the exponent:  "
   13:
   14: scanfFormatStr:
   15:          .string "%lu"
   16:
   17: printfFormatStr:
   18:          .string "%lu to the %lu power is %lu.\n"
   19:
   20:          //-------------------------------------------------------------
   21:
   22:          .section .data
   23:
   24: ulPower:
   25:          .quad   1
   26:
   27:          //-------------------------------------------------------------
   28:
   29:          .section .bss
   30:
   31: ulBase:
   32:          .skip   8
   33:
   34: ulExp:
   35:          .skip   8
   36:
   37: ulIndex:
   38:          .skip   8
   39:
   40:          //-------------------------------------------------------------
   41:
   42:          .section .text
   43:
   44:          //-------------------------------------------------------------
   45:          // Read a non-negative base and exponent from stdin.  Write
   46:          // base raised to the exponent power to stdout.  Return 0.
   47:          // int main(void)
   48:          //-------------------------------------------------------------
   49:
   50:          // Must be a multiple of 16
   51:          .equ    MAIN_STACK_BYTECOUNT, 16
   52:
   53:          .global main
   54:
   55: main:
   56:
   57:          // Prolog
   58:          sub     sp, sp, MAIN_STACK_BYTECOUNT
   59:          str     x30, [sp]
   60:
   61:          // printf("Enter the base:  ")
   62:          adr     x0, basePromptStr
   63:          bl      printf
```

```
 64:
 65:         // scanf("%d", &ulBase)
 66:         adr     x0, scanfFormatStr
 67:         adr     x1, ulBase
 68:         bl      scanf
 69:
 70:         // printf("Enter the exponent:  ")
 71:         adr     x0, expPromptStr
 72:         bl      printf
 73:
 74:         // scanf("%d", &ulExp)
 75:         adr     x0, scanfFormatStr
 76:         adr     x1, ulExp
 77:         bl      scanf
 78:
 79:         // ulIndex = 1
 80:         mov     x0, 1
 81:         adr     x1, ulIndex
 82:         str     x0, [x1]
 83:
 84: powerLoop:
 85:
 86:         // if (ulIndex > ulExp) goto powerLoopEnd
 87:         adr     x0, ulIndex
 88:         ldr     x0, [x0]
 89:         adr     x1, ulExp
 90:         ldr     x1, [x1]
 91:         cmp     x0, x1
 92:         bhi     powerLoopEnd
 93:
 94:         // ulPower *= ulBase
 95:         adr     x0, ulPower
 96:         ldr     x1, [x0]
 97:         adr     x2, ulBase
 98:         ldr     x2, [x2]
 99:         mul     x1, x1, x2
100:         str     x1, [x0]
101:
102:         // ulIndex++
103:         adr     x0, ulIndex
104:         ldr     x1, [x0]
105:         add     x1, x1, 1
106:         str     x1, [x0]
107:
108:         // goto powerLoop
109:         b       powerLoop
110:
111: powerLoopEnd:
112:
113:         // printf("%ld to the %ld power is %ld.\n",ulBase,ulExp,ulPower)
114:         adr     x0, printfFormatStr
115:         adr     x1, ulBase
116:         ldr     x1, [x1]
117:         adr     x2, ulExp
118:         ldr     x2, [x2]
119:         adr     x3, ulPower
120:         ldr     x3, [x3]
121:         bl      printf
122:
123:         // Epilog and return 0
124:         mov     w0, 0
125:         ldr     x30, [sp]
126:         add     sp, sp, MAIN_STACK_BYTECOUNT
```

powerunsigned.s (Page 3 of 3)

```
127:          ret
128:
129:          .size   main, (. - main)
```

Princeton University
COS 217: Introduction to Programming Systems
ARMv8 Condition Flags

Condition Flags

Bits in the `pstate` register

`CMP` *`Xs`*`|SP,` *`Xm`*

CPU performs the subtraction *`Xs`*`|SP` − *`Xm`*
More precisely, CPU performs the addition *`Xs`*`|SP + onescomp(`*`Xm`*`)` + 1 and sets the condition flags depending upon the sum:

Condition Code	
Z (zero flag)	CPU sets Z to 1 iff all bits of the sum are 0.
N (negative flag)	CPU sets N to 1 iff the most significant bit of the sum is 1.
C (carry flag)	CPU sets C to 1 iff the addition caused a carry.
V (overflow flag)	CPU sets V to 1 iff both addends are >= 0 and the sum is < 0, or both addends are < 0 and the sum is >= 0.

Conditional Branch Instructions (Used After Comparing Unsigned Numbers)

Instruction	Branch if and only if:
`beq (branch iff equal)`	`Z==1`
`bne (branch iff not equal)`	`Z==0`
`blo (branch iff lower)`	`C==0`
`bhs (branch iff higher or same)`	`C==1`
`bls (branch iff lower or same)`	`C==0 || Z==1`
`bhi (branch iff higher)`	`C==1 && Z==0`

Why does `blo` branch iff `C==0`? Examples (assuming a 4-bit computer):

(1) $5 - 3 = 0101_B - 0011_B = 0101_B + 1100_B + 1 = 0010_B$, C==1 => don't branch

(2) $5 - 0 = 0101_B - 0000_B = 0101_B + 1111_B + 1 = 1010_B$, C==1 => don't branch

(3) $3 - 5 = 0011_B - 0101_B = 0011_B + 1010_B + 1 = 1110_B$, C==0 => branch

(3) $0 - 5 = 0000_B - 0101_B = 0000_B + 1010_B + 1 = 1011_B$, C==0 => branch

So branch if and only if `C==0`.

Conditional Branch Instructions
(Used After Comparing Signed Numbers)

Instruction	Branch if and only if:
beq (branch iff equal)	Z==1
bne (branch iff not equal)	Z==0
blt (branch iff less than)	N!=V
bge (branch iff greater than or equal)	N==V
ble (branch iff less than or equal)	N!=V || Z==1
bgt (branch iff greater than)	N==V && Z==0

Why does `blt` branch iff if `N!=V`? Examples (assuming a 4 bit computer):

(1) 5 - 3 = 0101_B - 0011_B = 0101_B + 1100_B + 1 = 0010_B, N==0, V==0 => N==V => don't branch

(2) 3 - 5 = 0011_B - 0101_B = 0011_B + 1010_B + 1 = 1110_B, N==1, V==0 => N!=V => branch

(3) -5 - -3 = 1011_B - 1101_B = 1011_B + 0010_B + 1 = 1110_B, N==1, V==0 => N!=V => branch

(4) -3 - -5 = 1101_B - 1011_B = 1101_B + 0100_B + 1 = 0010_B, N==0, V==0 => N==V => don't branch

(5) 3 - -2 = 0011_B - 1110_B = 0011_B + 0001_B + 1 = 0101_B, N==0, V==0 => N==V => don't branch

(6) 3 - -6 = 0011_B - 1010_B = 0011_B + 0101_B + 1 = 1001_B, N==1, V==1 => N==V => don't branch

(7) -3 - 2 = 1101_B - 0010_B = 1101_B + 1101_B + 1 = 1111_B, N==1, V==0 => N!=V => branch

(8) -3 - 6 = 1101_B - 0110_B = 1101_B + 1001_B + 1 = 0111_B, N==0, V==1 => N!=V => branch

So branch if and only if `N!=V`.

Princeton University
COS 217: Introduction to Programming Systems
ARMv8 Memory Operands

Let `reg[Xn]` mean "the contents of register *Xn*".

Register Memory Operands

```
[Xn]              Compute an address as reg[Xn].
```

Examples:

```
(1) ldr x1, [x0]
(2) str x1, [x0]
(3) ldr w1, [x0]
(4) str w1, [x0]
```

Immediate Offset Memory Operands

```
[Xn, imm]         Compute an address as reg[Xn] + imm.
```

Examples:

```
(5) ldr x1, [x0, 4]
(6) str x1, [x0, 8]
(7) ldr w1, [x0, 12]
(8) str w1, [x0, 16]
```

Register Offset Memory Operands

```
[Xn, Xm]          Compute an address as reg[Xn] + reg[Xm].
```

Examples:

```
(9)  ldr x2, [x0, x1]
(10) str x2, [x0, x1]
(11) ldr w2, [x0, x1]
(12) str w2, [x0, x1]
```

Scaled Register Offset Memory Operands

```
[Xn, Xm, lsl 3]   Compute an address as reg[Xn] + (reg[Xm] << 3).
                  The loaded/stored object must consist of 8 bytes.
[Xn, Xm, lsl 2]   Compute an address as reg[Xn] + (reg[Xm] << 2).
                  The loaded/stored object must consist of 4 bytes.
[Xn, Xm, lsl 1]   Compute an address as reg[Xn] + (reg[Xm] << 1).
                  The loaded/stored object must consist of 2 bytes.
```

Examples:

```
(13) ldr  x2, [x0, x1, lsl 3]
(14) str  x2, [x0, x1, lsl 3]
(15) ldr  w2, [x0, x1, lsl 2]
(16) str  w2, [x0, x1, lsl 2]
(17) ldrh w2, [x0, x1, lsl 1]
(18) strh w2, [x0, x1, lsl 1]
```

rev.c (Page 1 of 1)

```
 1: /*-------------------------------------------------------------------------*/
 2: /* rev.c                                                                   */
 3: /* Author: Bob Dondero                                                     */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*-------------------------------------------------------------------------*/
 9:
10: /* The number of elements in the array. */
11: enum {ARRAY_LENGTH = 5};
12:
13: long alNums[ARRAY_LENGTH];  /* Bad style */
14: long lIndex;                /* Bad style */
15:
16: /*-------------------------------------------------------------------------*/
17:
18: /* Read ARRAY_LENGTH integers from stdin, and write them in reverse
19:    order to stdout.  Return 0. */
20:
21: int main(void)
22: {
23:    printf("Enter %d integers:\n", ARRAY_LENGTH);
24:    for (lIndex = 0; lIndex < ARRAY_LENGTH; lIndex++)
25:       scanf("%ld", &alNums[lIndex]);
26:
27:    printf("\n");
28:
29:    printf("The integers in reverse order are:\n");
30:    for (lIndex = ARRAY_LENGTH-1; lIndex >= 0; lIndex--)
31:       printf("%ld\n", alNums[lIndex]);
32:
33:    return 0;
34: }
```

revflat.c (Page 1 of 1)

```
 1: /*-------------------------------------------------------------------*/
 2: /* revflat.c                                                         */
 3: /* Author: Bob Dondero                                               */
 4: /*-------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*-------------------------------------------------------------------*/
 9:
10: /* The number of elements in the array. */
11: enum {ARRAY_LENGTH = 5};
12:
13: long alNums[ARRAY_LENGTH];  /* Bad style */
14: long lIndex;                /* Bad style */
15:
16: /*-------------------------------------------------------------------*/
17:
18: /* Read ARRAY_LENGTH integers from stdin, and write them in reverse
19:    order to stdout.  Return 0. */
20:
21: int main(void)
22: {
23:    printf("Enter %d integers:\n", ARRAY_LENGTH);
24:
25:    lIndex = 0;
26: readLoop:
27:    if (lIndex >= ARRAY_LENGTH) goto readLoopEnd;
28:    scanf("%ld", &alNums[lIndex]);
29:    lIndex++;
30:    goto readLoop;
31: readLoopEnd:
32:
33:    printf("\n");
34:    printf("The integers in reverse order are:\n");
35:
36:    lIndex = ARRAY_LENGTH-1;
37: writeLoop:
38:    if (lIndex < 0) goto writeLoopEnd;
39:    printf("%ld\n", alNums[lIndex]);
40:    lIndex--;
41:    goto writeLoop;
42: writeLoopEnd:
43:
44:    return 0;
45: }
```

rev.s (Page 1 of 3)

```
    1: //-----------------------------------------------------------------------
    2: // rev.s
    3: // Author: Bob Dondero
    4: //-----------------------------------------------------------------------
    5:
    6:         .equ    ARRAY_LENGTH, 5
    7:
    8: //-----------------------------------------------------------------------
    9:
   10:         .section .rodata
   11:
   12: promptStr:
   13:         .string "Enter %d integers:\n"
   14: scanfFormatStr:
   15:         .string "%ld"
   16: newLineStr:
   17:         .string "\n"
   18: messageStr:
   19:         .string "The integers in reverse order are:\n"
   20: printfFormatStr:
   21:         .string "%ld\n"
   22:
   23: //-----------------------------------------------------------------------
   24:
   25:         .section .data
   26:
   27: //-----------------------------------------------------------------------
   28:
   29:         .section .bss
   30:
   31: alNums:
   32:         .skip   8 * ARRAY_LENGTH
   33: lIndex:
   34:         .skip   8
   35:
   36: //-----------------------------------------------------------------------
   37:
   38:         .section .text
   39:
   40:         //-------------------------------------------------------------
   41:         // Read ARRAY_LENGTH integers from stdin, and write them in
   42:         // reverse order to stdout. Return 0.
   43:         //-------------------------------------------------------------
   44:
   45:         // Must be a multiple of 16
   46:         .equ    MAIN_STACK_BYTECOUNT, 16
   47:
   48:         .global main
   49:
   50: main:
   51:
   52:         // Prolog
   53:         sub     sp, sp, MAIN_STACK_BYTECOUNT
   54:         str     x30, [sp]
   55:
   56:         // printf("Enter %d integers:\n", ARRAY_LENGTH)
   57:         adr     x0, promptStr
   58:         mov     w1, ARRAY_LENGTH
   59:         bl      printf
   60:
   61:         // lIndex = 0
   62:         adr     x0, lIndex
   63:         str     xzr, [x0]
```

```
 64:
 65: readLoop:
 66:
 67:         // if (lIndex >= ARRAY_LENGTH) goto readLoopEnd
 68:         adr     x0, lIndex
 69:         ldr     x0, [x0]
 70:         cmp     x0, ARRAY_LENGTH
 71:         bge     readLoopEnd
 72:
 73:         // scanf("%ld", &alNums[lIndex])
 74:         adr     x0, scanfFormatStr
 75:         adr     x1, lIndex
 76:         ldr     x1, [x1]
 77:         lsl     x1, x1, 3
 78:         adr     x2, alNums
 79:         add     x1, x1, x2
 80:         bl      scanf
 81:
 82:         // lIndex++
 83:         adr     x0, lIndex
 84:         ldr     x1, [x0]
 85:         add     x1, x1, 1
 86:         str     x1, [x0]
 87:
 88:         // goto readLoop
 89:         b       readLoop
 90:
 91: readLoopEnd:
 92:
 93:         // printf("\n")
 94:         adr     x0, newLineStr
 95:         bl      printf
 96:
 97:         // printf("The integers in reverse order are:\n")
 98:         adr     x0, messageStr
 99:         bl      printf
100:
101:         // lIndex = ARRAY_LENGTH-1
102:         mov     x0, ARRAY_LENGTH-1
103:         adr     x1, lIndex
104:         str     x0, [x1]
105:
106: writeLoop:
107:
108:         // if (lIndex < 0) goto writeLoopEnd
109:         adr     x0, lIndex
110:         ldr     x0, [x0]
111:         cmp     x0, 0
112:         blt     writeLoopEnd
113:
114:         // printf("%ld\n", alNums[lIndex])
115:         adr     x0, printfFormatStr
116:         adr     x1, alNums
117:         adr     x2, lIndex
118:         ldr     x2, [x2]
119:         lsl     x2, x2, 3
120:         add     x1, x1, x2
121:         ldr     x1, [x1]
122:         bl      printf
123:
124:         // lIndex--
125:         adr     x0, lIndex
126:         ldr     x1, [x0]
```

rev.s (Page 3 of 3)

```
127:         sub     x1, x1, 1
128:         str     x1, [x0]
129:
130:         // goto writeLoop
131:         b       writeLoop
132:
133: writeLoopEnd:
134:
135:         // Epilog and return 0
136:         mov     w0, 0
137:         ldr     x30, [sp]
138:         add     sp, sp, MAIN_STACK_BYTECOUNT
139:         ret
140:
141:         .size   main, (. - main)
```

revregoffset.s (Page 1 of 3)

```
    1: //-----------------------------------------------------------------------
    2: // revregoffset.s
    3: // Author: Bob Dondero
    4: //-----------------------------------------------------------------------
    5:
    6:         .equ    ARRAY_LENGTH, 5
    7:
    8: //-----------------------------------------------------------------------
    9:
   10:         .section .rodata
   11:
   12: promptStr:
   13:         .string "Enter %d integers:\n"
   14: scanfFormatStr:
   15:         .string "%ld"
   16: newLineStr:
   17:         .string "\n"
   18: messageStr:
   19:         .string "The integers in reverse order are:\n"
   20: printfFormatStr:
   21:         .string "%ld\n"
   22:
   23: //-----------------------------------------------------------------------
   24:
   25:         .section .data
   26:
   27: //-----------------------------------------------------------------------
   28:
   29:         .section .bss
   30:
   31: alNums:
   32:         .skip   8 * ARRAY_LENGTH
   33: lIndex:
   34:         .skip   8
   35:
   36: //-----------------------------------------------------------------------
   37:
   38:         .section .text
   39:
   40:         //-------------------------------------------------------------
   41:         // Read ARRAY_LENGTH integers from stdin, and write them in
   42:         // reverse order to stdout. Return 0.
   43:         //-------------------------------------------------------------
   44:
   45:         // Must be a multiple of 16
   46:         .equ    MAIN_STACK_BYTECOUNT, 16
   47:
   48:         .global main
   49:
   50: main:
   51:
   52:         // Prolog
   53:         sub     sp, sp, MAIN_STACK_BYTECOUNT
   54:         str     x30, [sp]
   55:
   56:         // printf("Enter %d integers:\n", ARRAY_LENGTH)
   57:         adr     x0, promptStr
   58:         mov     w1, ARRAY_LENGTH
   59:         bl      printf
   60:
   61:         // lIndex = 0
   62:         adr     x0, lIndex
   63:         str     xzr, [x0]
```

```
 64:
 65: readLoop:
 66:
 67:         // if (lIndex >= ARRAY_LENGTH) goto readLoopEnd
 68:         adr     x0, lIndex
 69:         ldr     x0, [x0]
 70:         cmp     x0, ARRAY_LENGTH
 71:         bge     readLoopEnd
 72:
 73:         // scanf("%ld", &alNums[lIndex])
 74:         adr     x0, scanfFormatStr
 75:         adr     x1, lIndex
 76:         ldr     x1, [x1]
 77:         lsl     x1, x1, 3
 78:         adr     x2, alNums
 79:         add     x1, x1, x2
 80:         bl      scanf
 81:
 82:         // lIndex++
 83:         adr     x0, lIndex
 84:         ldr     x1, [x0]
 85:         add     x1, x1, 1
 86:         str     x1, [x0]
 87:
 88:         // goto readLoop
 89:         b       readLoop
 90:
 91: readLoopEnd:
 92:
 93:         // printf("\n")
 94:         adr     x0, newLineStr
 95:         bl      printf
 96:
 97:         // printf("The integers in reverse order are:\n")
 98:         adr     x0, messageStr
 99:         bl      printf
100:
101:         // lIndex = ARRAY_LENGTH-1
102:         mov     x0, ARRAY_LENGTH-1
103:         adr     x1, lIndex
104:         str     x0, [x1]
105:
106: writeLoop:
107:
108:         // if (lIndex < 0) goto writeLoopEnd
109:         adr     x0, lIndex
110:         ldr     x0, [x0]
111:         cmp     x0, 0
112:         blt     writeLoopEnd
113:
114:         // printf("%ld\n", alNums[lIndex])
115:         adr     x0, printfFormatStr
116:         adr     x1, alNums
117:         adr     x2, lIndex
118:         ldr     x2, [x2]
119:         lsl     x2, x2, 3
120:         ldr     x1, [x1, x2] // Register offset addressing
121:         bl      printf
122:
123:         // lIndex--
124:         adr     x0, lIndex
125:         ldr     x1, [x0]
126:         sub     x1, x1, 1
```

revregoffset.s (Page 3 of 3)

```
127:         str     x1, [x0]
128:
129:         // goto writeLoop
130:         b       writeLoop
131:
132: writeLoopEnd:
133:
134:         // Epilog and return 0
135:         mov     w0, 0
136:         ldr     x30, [sp]
137:         add     sp, sp, MAIN_STACK_BYTECOUNT
138:         ret
139:
140:         .size   main, (. - main)
```

revscaledregoffset.s (Page 1 of 3)

```
    1: //-----------------------------------------------------------------------
    2: // revregscaledregoffset.s
    3: // Author: Bob Dondero
    4: //-----------------------------------------------------------------------
    5:
    6:         .equ    ARRAY_LENGTH, 5
    7:
    8: //-----------------------------------------------------------------------
    9:
   10:         .section ".rodata"
   11:
   12: promptStr:
   13:         .string "Enter %d integers:\n"
   14: scanfFormatStr:
   15:         .string "%ld"
   16: newlineStr:
   17:         .string "\n"
   18: messageStr:
   19:         .string "The integers in reverse order are:\n"
   20: printfFormatStr:
   21:         .string "%ld\n"
   22:
   23: //-----------------------------------------------------------------------
   24:
   25:         .section ".data"
   26:
   27: //-----------------------------------------------------------------------
   28:
   29:         .section ".bss"
   30:
   31: alNums:
   32:         .skip   8 * ARRAY_LENGTH
   33: lIndex:
   34:         .skip   8
   35:
   36: //-----------------------------------------------------------------------
   37:
   38:         .section ".text"
   39:
   40:         //-------------------------------------------------------------
   41:         // Read ARRAY_LENGTH integers from stdin, and write them in
   42:         // reverse order to stdout. Return 0.
   43:         //-------------------------------------------------------------
   44:
   45:         // Must be a multiple of 16
   46:         .equ    MAIN_STACK_BYTECOUNT, 16
   47:
   48:         .globl  main
   49:
   50: main:
   51:
   52:         // Prolog
   53:         sub     sp, sp, MAIN_STACK_BYTECOUNT
   54:         str     x30, [sp]
   55:
   56:         // printf("Enter %d integers:\n", ARRAY_LENGTH)
   57:         adr     x0, promptStr
   58:         mov     w1, ARRAY_LENGTH
   59:         bl      printf
   60:
   61:         // lIndex = 0
   62:         adr     x0, lIndex
   63:         str     xzr, [x0]
```

```
   64:
   65: readLoop:
   66:
   67:         // if (lIndex >= ARRAY_LENGTH) goto readLoopEnd
   68:         adr     x0, lIndex
   69:         ldr     x0, [x0]
   70:         cmp     x0, ARRAY_LENGTH
   71:         b.ge    readLoopEnd
   72:
   73:         // scanf("%ld", &alNums[lIndex])
   74:         adr     x0, scanfFormatStr
   75:         adr     x1, lIndex
   76:         ldr     x1, [x1]
   77:         lsl     x1, x1, 3
   78:         adr     x2, alNums
   79:         add     x1, x1, x2
   80:         bl      scanf
   81:
   82:         // lIndex++
   83:         adr     x0, lIndex
   84:         ldr     x1, [x0]
   85:         add     x1, x1, 1
   86:         str     x1, [x0]
   87:
   88:         // goto readLoop
   89:         b       readLoop
   90:
   91: readLoopEnd:
   92:
   93:         // printf("\n")
   94:         adr     x0, newlineStr
   95:         bl      printf
   96:
   97:         // printf("The integers in reverse order are:\n")
   98:         adr     x0, messageStr
   99:         bl      printf
  100:
  101:         // lIndex = ARRAY_LENGTH-1
  102:         mov     x0, ARRAY_LENGTH-1
  103:         adr     x1, lIndex
  104:         str     x0, [x1]
  105:
  106: writeLoop:
  107:
  108:         // if (lIndex < 0) goto writeLoopEnd
  109:         adr     x0, lIndex
  110:         ldr     x0, [x0]
  111:         cmp     x0, 0
  112:         blt    writeLoopEnd
  113:
  114:         // printf("%ld\n", alNums[lIndex])
  115:         adr     x0, printfFormatStr
  116:         adr     x1, alNums
  117:         adr     x2, lIndex
  118:         ldr     x2, [x2]
  119:         ldr     x1, [x1, x2, lsl 3] // Scaled register offset addressing
  120:         bl      printf
  121:
  122:         // lIndex--
  123:         adr     x0, lIndex
  124:         ldr     x1, [x0]
  125:         sub     x1, x1, 1
  126:         str     x1, [x0]
```

revscaledregoffset.s (Page 3 of 3)

```
127:
128:        // goto writeLoop
129:        b       writeLoop
130:
131: writeLoopEnd:
132:
133:        // Epilog and return 0
134:        mov     w0, 0
135:        ldr     x30, [sp]
136:        add     sp, sp, MAIN_STACK_BYTECOUNT
137:        ret
138:
139:        .size   main, (. - main)
```

Princeton University
COS 217: Introduction to Programming Systems
GDB Tutorial and Reference
for ARMv8 Assembly Language

Part 1: Tutorial

Motivation

Suppose you're composing the `power.s` program. Further suppose that the program assembles and links cleanly, but is producing incorrect results at runtime. What can you do to debug the program?

One approach is temporarily to insert calls of `printf(...)` throughout the code to get a sense of the flow of control and the values of variables at critical points. That's fine, but often is inconvenient. It is especially inconvenient in assembly language: the calls of `printf()` will change the values of registers, and thus may corrupt the very data that you wish to view.

An alternative is to use `gdb`. `gdb` allows you to set breakpoints in your code, step through your executing program one line at a time, examine the contents of registers and memory at breakpoints, etc.

Editing for gdb

To prepare your assembly language code to use `gdb`, make sure that the definition of each function ends with a `.size` directive indicating the size of that function. For example, in `power.s` the `main()` function should end with this `.size` directive:

```
.size main, (. - main)
```

Building for `gdb`

To prepare to use `gdb`, build the program with `gcc217` using the `-g` option:

```
$ gcc217 -g power.s -o power
```

Running GDB

The next step is to run `gdb`. You can run `gdb` directly from the shell. But it's much handier to run it from within `emacs`. So launch `emacs`, with no command-line arguments:

```
$ emacs
```

Now call the `emacs gdb` function via these keystrokes:

```
<Esc key> x gdb <Enter Key> power <Enter key>
```

At this point you are executing `gdb` from within `emacs`. `gdb` is displaying its `(gdb)` prompt.

Running Your Program

Issue the `run` command to run the program:

```
(gdb) run
```

`gdb` runs the program to completion, indicating that the process "exited normally."

`gdb` also displays the cryptic message "Missing separate debuginfos..." That message is innocuous; ignore it.

Command-line arguments and file redirection can be specified as part of the `run` command. For example the command `run 1 2 3` runs the program with command-line arguments 1, 2, and 3, and the command `run < myfile` runs the program with its `stdin` redirected to `myfile`.

Using Breakpoints

Set a breakpoint near the beginning of the `main()` function using the `break` command:

```
(gdb) break main
```

Run the program:

```
(gdb) run
```

`gdb` pauses execution at the beginning of the `main()` function. It opens a second window in which it displays your source code, with the about-to-be-executed line of code highlighted.

Issue the `continue` command to tell command `gdb` to continue execution past the breakpoint:

```
(gdb) continue
```

`gdb` continues past the breakpoint at the beginning of `main()`, and executes the program to completion.

Stepping Through the Program

Run the program again:

```
(gdb) run
```

Execution pauses at the beginning of the `main()` function. Issue the `next` command to execute the next instruction of your program:

```
(gdb) next
```

Continue issuing the `next` command repeatedly until the next instruction to be executed is the `bl printf` that appears near the end of the program.

Characters that are written to `stdout` do not necessarily appear in your terminal window immediately. As described in the *Debugging: Part 1* lecture, for efficiency characters written to `stdout` often are buffered; the characters are flushed from the buffer to your terminal window at some later time.

The `step` command is the same as the `next` command, except that it commands `gdb` to step into a called function which you have defined.

The `step` command does not cause `gdb` to step into a standard C function. The `stepi` ("step instruction") command causes `gdb` to step into any function, including a standard C function.

Examining Registers

Issue the `info registers` command to examine the contents of the registers:

```
(gdb) info registers
```

Issue the `print` command to examine the contents of any given register. Some examples:

```
(gdb) print/d $x1      Print as a decimal integer the 8 bytes
                       which are the contents of register X1
(gdb) print/a $x0      Print as a hexadecimal address the 8 bytes
                       which are the contents of register X0
```

Note that you must precede the name of the register with `$`.

Examining Memory

Issue the `x` command to examine the contents of memory at any given address. Some examples:

```
(gdb) x/gd &lBase              Examine as a "giant" decimal integer
                               the 8 bytes of memory at lBase
(gdb) x/gd 0x420035            Examine as a "giant" decimal integer
                               the 8 bytes of memory at 0x420035
(gdb) x/c &printfFormatStr     Examine as a char the 1 byte of
                               memory at printfFormatStr
(gdb) x/30c &printfFormatStr   Examine as 30 chars the bytes of
                               memory beginning at printfFormatStr
(gdb) x/s &printfFormatStr     Examine as a string the bytes of
                               memory beginning at printfFormatStr
(gdb) x/s $x0                  Examine as a string the bytes of
                               memory beginning at the address
                               contained in register X0
```

Quitting GDB

As usual, type:

```
<Ctrl-x> <Ctrl-c>
```

to exit `emacs`.

Command Abbreviations

The most commonly used `gdb` commands have one-letter abbreviations (`r`, `b`, `c`, `n`, `s`, `p`). Also, pressing the Enter key without typing a command tells `gdb` to reissue the previous command.

Part 2: Reference

gcc217 -g ... -o *program*	Assemble and link with debugging information
gdb [-d *sourcefiledir*] [-d *sourcefiledir*] ... *program* [*corefile*]	Run `gdb` from a shell
ESC x gdb [-d *sourcefiledir*] [-d *sourcefiledir*] ... *program* [*corefile*]	Run `gdb` from Emacs

Miscellaneous	
quit	Exit `gdb`.
directory [*dir1*] [*dir2*] ...	Add directories *dir1*, *dir2*, ... to the list of directories searched for source files, or clear the directory list.
help [*cmd*]	Print a description command *cmd*

Running the Program	
run [*arg1*],[*arg2*] …	Run the program with command-line arguments *arg1*, *arg2*, ...
set args *arg1 arg2* ...	Set program's the command-line arguments to *arg1*, *arg2*, ...
show args	Print the program's command-line arguments.

Using Breakpoints	
info breakpoints	Print a list of all breakpoints.
break *addr*	Set a breakpoint at memory address *addr*. The address can be denoted by a label.
condition *bpnum expr*	Add a condition to breakpoint *bpnum* such that the break occurs if and only if expression *expr* is non-zero (TRUE).
commands [*bpnum*] *cmd1 cmd2* ...	Execute commands *cmd1*, *cmd2*, ... whenever breakpoint *bpnum* (or the current breakpoint) is hit.
continue	Continue executing the program.
kill	Stop executing the program.
delete [*bpnum1*][,*bpnum2*]...	Delete breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.
clear [*addr*]	Clear the breakpoint at memory address *addr*. The address can be denoted by a label. Or clear the current breakpoint.
disable [*bpnum1*][,*bpnum2*]...	Disable breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.
enable [*bpnum1*][,*bpnum2*]...	Enable breakpoints *bpnum1*, *bpnum2*, ..., or all breakpoints.

Stepping through the Program	
next	"Step over" the next instruction.
step	"Step into" the next instruction.
finish	"Step out" of the current function.

Examining Registers and Memory	
info registers	Print the contents of all registers.
print/*f* $*reg*	Print the contents of register *reg* using format *f*. The format is typically 'd' (decimal), 'a' (address), 'x' (hexadecimal), 'c' (character), or 'i' (instruction); it defaults to 'd'.
x/*rsf addr*	Examine the contents of memory at address *addr*. The repeat count *r* is optional; it defaults to 1. The size *s* is typically 'h' (two bytes), 'w' (four bytes), or 'g' (eight bytes); its default varies based upon format *f*.
x/*rsf* &*label*	Examine the contents of memory at the address denoted by *label*.
x/*rsf* $*reg*	Examine the contents of memory at the address contained in register *reg*.
info display	Print the display list.
display/*f* $*reg*	Add an entry to the display list; at each break, print the contents of register *reg*.
display/*rsf addr*	Add an entry to the display list; at each break, print the contents of memory at address *addr*.
display/*rsf* &*label*	Add an entry to the display list; at each break, print the contents of memory at the address denoted by *label*.
undisplay *displaynum*	Remove entry with number *displaynum* from the display list.

Examining the Call Stack	
where	Print the call stack.
frame	Print the top of the call stack.
up	Move the context toward the bottom of the call stack.
down	Move the context toward the top of the call stack