Tutorial3.3

Binary Arithmetic
Assignment2

Lei Wang
lei.wang2@ucalgary.ca

UNIVERSITY OF CALGARY

Modulus Arithmetic

- Any carry out is ignored when using ordinary arithmetic instructions
  - Eg: 9 + 8 should give 17
    - but actually gives (9 + 8) mod 16 on a 4-bit CPU

$$
\begin{array}{rr}
 & 1001 \\
+ & \underline{1000} \\
1 & 0001
\end{array}
$$

- Instructions like adds or subs do set the carry flag
  - can be used to do extended precision arithmetic

UNIVERSITY OF CALGARY

Signed number Branching Conditions

- subs
  - alias for *cmp*
- ands
  - *ands Xzr, Xn, Xm*
  - alias for *tst*
  - The *tst* instruction performs a bitwise AND operation on the value in Xn and the value of Xm. This is the same as an ANDS instruction, except that the result is discarded, updates the N and Z flags according to the result.
  - eg
    - *tst x20, 0x8*
    - *b.eq bitclear*
- subs and ands set the flags

Signed number Branching Conditions

- Recall that signed branch instructions use the N, Z, V flags

Name	Meaning	C equivalent	Flags
eq	equal	==	Z == 1
ne	not equal	!=	Z == 0
gt	greater than	>	Z == 0 && N == V
ge	greater than or equal	>=	N == V
lt	less than	<	N != V
le	less than or equal	<=	!(Z == 0 && N == V)

ands (tst)

- ands wzr, product, 0x1   or tst product, 0x1
  - bitwise AND of product and 0x1
- b.eq productEndsWithZero
  - b.eq doesn't mean true, but rather means the Z flag is set.
  - The Z flag is set here if the ANDS operation results in zero (0).
  - That means the Z flag is set (b.eq) when bit-0 of product is 0.
- b.ne productEndsWithOne
  - This is not necessary if you just let your code fall through
  - b.ne (Z flag is not set) when bit-0 of product is 1
  - If bit-0 of product is 1, then ANDS product, 0x1 will product 1 (Z flag is not set).

Multiplication

- Can be done with an iterated procedure
  - Repeated for every bit in the source registers
- Each step consists of two sub-steps:
  - A conditional addition
  - An arithmetic Shift right
- The product may require twice as many bits as for the source registers
  - the result is put into 2 concatenated registers
- if the original multiplier is negative, an extra step is needed:
  - Subtract the multiplicand from the high-order part of the result(i.e.from the product register)

UNIVERSITY OF CALGARY

why extra step?

- Negating a number is done by:
  1. Taking the one's complement
     - Toggle all 0's to 1's, and vice versa
  2. Adding 1 to the result
  - Eg: find the bit pattern for -5 in a 4-bit register
    - +5 is 0101
    - One's complement: 1010
    - Add 1: 1011
- The result equals 2^{N+1}-positive value
  - Eg:
    - $2^5(10000) - 5(0101) = 1011$
- $a*(neg.b) = a*(2^{n+1}-b) = a*2^{n+1}-a*b = a*2^{n+1}+a*(-b)$

  so $a*(-b) = a*(neg.b) - a*2^{n+1}$ = product $- a*2^{n+1}$

Assignment2

- Create an ARMv8 assembly language that implenment the following integer multiplication progarm

conditional addition
using *tst* in assembly

shift the result to multiplier

arithmetic shift right(asr)

extra step for negative

extend the product to 64bit
using *sxtw* in assembly

```c
#include <stdio.h>
#define FALSE 0
#define TRUE  1

int main()
{
  int multiplier, multiplicand, product, i, negative;
  long int result, temp1, temp2;

  // Initialize variables
  multiplicand = -16843010;
  multiplier = 70;
  product = 0;

  // Print out initial values of variables
  printf("multiplier = 0x%08x (%d)  multiplicand = 0x%08x (%d)\n\n",
   multiplier, multiplier, multiplicand, multiplicand);

  // Determine if multiplier is negative
  negative = multiplier < 0 ? TRUE : FALSE;

  // Do repeated add and shift
  for (i = 0; i < 32; i++) {
    if (multiplier & 0x1) {
      product = product + multiplicand;
    }

    // Arithmetic shift right the combined product and multiplier
    multiplier = multiplier >> 1;
    if (product & 0x1) {
      multiplier = multiplier | 0x80000000;
    } else {
      multiplier = multiplier & 0x7FFFFFFF;
    }
    product = product >> 1;
  }

  // Adjust product register if multiplier is negative
  if (negative) {
    product = product - multiplicand;
  }

  // Print out product and multiplier
  printf("product = 0x%08x  multiplier = 0x%08x\n",
   product, multiplier);

  // Combine product and multiplier together
  temp1 = (long int)product & 0xFFFFFFFF;
  temp1 = temp1 << 32;
  temp2 = (long int)multiplier & 0xFFFFFFFF;
  result = temp1 + temp2;

  // Print out 64-bit result
  printf("64-bit result = 0x%016lx (%ld)\n", result, result);

  return 0;
}
```

Assignment2 requirement

- use 32-bit registers for variables declared using int, 64-bit for long int
- use m4 macros
- debug using gdb, capture it using script
- name requirement: assign2a.asm
  assign2b.asm
  assign2c.asm
- Don’t use gcc -S

UNIVERSITY OF CALGARY