SPRING 2016

CMPE 364

Microprocessor Based Design

Dr. Mohamed Al-Meer

Lecture Expectations

- Expected to achieve:
  - Writing assembly code
  - Cycle Counting
  - Instruction scheduling
    - Load instructions
    - Preloading
    - Loop unrolling
  - Register Allocation
  - Allocating variables to registers

Lecture Expectations

- Expected to achieve:
  - Conditional execution
  - Looping Constructs
    - Decremented Counted Loops
  - Efficient Switches

Allocating Variables to Register Numbers

- When you write an assembly routine, it is best to start by using **names for the variables**.
  - This allows you to **change the allocation of vari**ables to register numbers easily
  - use **different register names for the same physical register** number
  - **Register names increase the clarity and read**ability of optimized code

Allocating Variables to Register Numbers - EXAMPLE

- we want to shift an **array of *N* bits upwards** in memory by ***k* bits**.
- For simplicity assume that ***N*** is large and a **multiple** of **256**.
- Also assume that **$0 \leq k < 32$** and that
- The input and output pointers are word aligned.
- **See next C routine:**
  - The **loop is unrolled 8 times** for maximum efficiency

Allocating Variables to Register Numbers - EXAMPLE

```
unsigned int shift_bits(unsigned int *out, unsigned int *in,
                        unsigned int N, unsigned int k)
{
  unsigned int carry=0, x;

  do
  {
    x = *in++;
    *out++ = (x<<k) | carry;
    carry = x>>(32-k);
    N -= 32;
  } while (N);

  return carry;
}
```

- **Original C code.**

Allocating Variables to Register Numbers - EXAMPLE

```
shift_bits
      STMFD   sp!, {r4-r11, lr}       ; save registers
      RSB     kr, k, #32              ; kr = 32-k;
      MOV     carry, #0
loop
      LDMIA   in!, {x_0-x_7}          ; load 8 words
      ORR     y_0, carry, x_0, LSL k  ; shift the 8 words
      MOV     carry, x_0, LSR kr
      ORR     y_1, carry, x_1, LSL k
      MOV     carry, x_1, LSR kr
      ORR     y_2, carry, x_2, LSL k
      MOV     carry, x_2, LSR kr
      ORR     y_3, carry, x_3, LSL k
      MOV     carry, x_3, LSR kr
      ORR     y_4, carry, x_4, LSL k
      MOV     carry, x_4, LSR kr
      ORR     y_5, carry, x_5, LSL k
      MOV     carry, x_5, LSR kr
      ORR     y_6, carry, x_6, LSL k
      MOV     carry, x_6, LSR kr
      ORR     y_7, carry, x_7, LSL k
      MOV     carry, x_7, LSR kr
      STMIA   out!, {y_0-y_7}         ; store 8 words
      SUBS    N, N, #256              ; N -= (8 words * 32 bits)
      BNE     loop                    ; if (N!=0) goto loop;
      MOV     r0, carry               ; return carry;
      LDMFD   sp!, {r4-r11, pc}
```

Allocating Variables to Register Numbers - EXAMPLE

- Now see the register naming and allocation**?**
- ***x_ array*** and ***y_ array*** are occupying the same register range? **Why?**
- For ***carry*** and ***kr***, you can use the stack since there are no remaining registers

```
out     RN 0
in      RN 1

N       RN 2
k       RN 3
```

```
x_0     RN 5
x_1     RN 6
x_2     RN 7
x_3     RN 8
x_4     RN 9
x_5     RN 10
x_6     RN 11
x_7     RN 12
y_0     RN 4
y_1     RN x_0
y_2     RN x_1
y_3     RN x_2
y_4     RN x_3
y_5     RN x_4
y_6     RN x_5
y_7     RN x_6
```

Conditional Execution

- The processor core can conditionally execute most ARM instructions.
- Based on one of **15 condition codes**.
- 14 conditions split into seven pairs of complements.
- By default, ARM instructions do not update the ***N***, ***Z***, ***C***, ***V*** flags in the ARM *cpsr*. Except with using "**S**".
- By combining **conditional execution** and **conditional setting** of the flags, you can **implement simple if statements** without any need for branches.
  - improves efficiency since branches can take many cycles and also reduces code size.

Conditional Execution Exampl-1

EXAMPLE 6.17 The following C code converts an unsigned integer $0 \leq i \leq 15$ to a hexadecimal character c:

```
if (i<10)
{
  c = i + '0';
}
else
{
  c = i + 'A'-10;
}
```

We can write this in assembly using conditional execution rather than conditional branches:

```
CMP     i, #10
ADDLO   c, i, #'0'
ADDHS   c, i, #'A'-10
```

Conditional Execution Example 2

EXAMPLE 6.18 The following C code identifies if c is a vowel:

```
if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
{
    vowel++;
}
```

In assembly you can write this using conditional comparisons:

```
TEQ    c, #'a'
TEQNE  c, #'e'
TEQNE  c, #'i'
TEQNE  c, #'o'
TEQNE  c, #'u'
ADDEQ  vowel, vowel, #1
```

Conditional Execution Example 3

EXAMPLE 6.19 Consider the following code that detects if c is a letter:

```
if ((c>='A' && c<='Z') || (c>='a' && c<='z'))
{
    letter++;
}
```

```
SUB     temp, c, #'A'
CMP     temp, #'Z'-'A'
SUBHI   temp, c, #'a'
CMPHI   temp, #'z'-'a'
ADDLS   letter, letter, #1
```

Looping Constructs Decremented Counted Loops

- On the ARM loops are fastest when they count down towards zero. **Why?**
- How to implement these loops efficiently in assembly
- For a decrementing loop of *N* iterations, the loop counter *i* counts down from *N* to 1 inclusive.
- The loop terminates with *i* = 0. An efficient implementation is:

Looping Constructs Decremented Counted Loops

- **One Solution**
- Count from **N to 1 inclusive**.
- The loop overhead is 1-Subtraction setting the condition codes and 2-Conditional branch
- On ARM7 and ARM9 this overhead costs **four cycles** per loop
  - **3 for BGT and 1 for SUB**

```
      MOV i, N
loop
      ; loop body goes here and i=N,N-1,...,1
      SUBS i, i, #1
      BGT  loop
```

Looping Constructs Decremented Counted Loops

- **Another Solution**
- If i is an index (**N-1 to 0 inclusive**)
- Good for access array element zero.
- Using a different conditional branch:
- ***Z* flag is set** on the **last iteration** of the loop and cleared for other iterations

```
        SUBS i, N, #1
loop
        ; loop body goes here and i=N-1,N-2,...,0
        SUBS i, i, #1
        BGE  loop
```