ARMv4 Memory Instructions

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

Memory

- A computer's memory is controlled by address lines, data lines, a read/write control line, and a clock line (Lec 17)
- The address lines selects the memory location to access
- The data lines can be used to send or receive data to/from the memory
- The read/write control line determine the direction data is being sent, and the clock line controls when the memory operation occurs

Memory

- Memory can be treated like a byte array
- The address lines act as an index into the array
- Thus, memory reads and writes can be described by:

```
r0 = mem[add]; // read to r0 from memory using add as the index

mem[add] = r0; // write to memory from r0 using add as the index
```

ARMv4 Memory Load Instructions

- Memory load instructions transfer the contents of a memory location to a CPU register
- A byte or a 32-bit word can be transferred
- Load instructions have the following format:

```
ldr Rd, [Rn, #immed]            // R[d] <= M[ R[n] + #immed]

ldr Rd, [Rn, Rm, sh #immed]     // R[d] <= M[ R[n] + (R[m] sh #immed)]

ldrb Rd, [Rn, #immed]           // R[d] <= M[ R[n] + #immed]

ldrb Rd, [Rn, Rm, sh #immed]    // R[d] <= M[ R[n] + (R[m] sh #immed)]
```

Memory Load Instructions

- ldr loads 32 bits from memory into a 32 bit register
- ldrb loads 8 bits from memory into a 32 bit register
  - The upper 24 bits are set to zero
- Similar to the data processing instruction has second operand:
  - an immediate value
  - a register with an optional shift/rotate
- In the immediate offset, the address is calculated by adding Rn with the immediate value
- In the register offset, Rn and Rm are added

Memory Load Instructions

- Rm can also be
  - left shifted
  - right shifted
  - arithmetic right shifted
  - right rotated before the addition
- The Rd register will contain the data, and the Rn and Rm registers are used to calculate the address
- The address is form by either Rn + Rm, or Rn - Rm
- An immediate value can be added or subtracted to Rn to form the address

Memory Store Instructions

- str stores 32 bits to memory from a register
- strb stores 8 bits to memory from the lower 8 bits of a register
- Similar to the data processing and load instructions, an immediate value or a register with an optional shift/rotate is available as the second operand

```
str Rd, [Rn, #immed]              // M[ R[n] + #immed] <= R[d]

str Rd, [Rn, Rm, sh #immed]       // M[ R[n] + (R[m] sh #immed] <= R[d]

strb Rd, [Rn, #immed]             // M[ R[n] + #immed] <= R[d]

strb Rd, [Rn, Rm, sh #immed]      // M[ R[n] + (R[m] sh #immed] <= R[d]
```

Memory Address Calculations

- The way the memory address is calculated is often called the addressing mode
- The load and store instructions provide four different ways of calculating the memory address
- The simplest is:

```
ldr r2, [r3]

str r2, [r3]
```

- The value of r3 contains the address
- If r3 is 0x2000, then the location accessed is 0x2000
  - r2 <= M[0x2000]

Memory Address Calculations

- A fixed constant can be added to the register to give the location
- The constant is between -4095 and 4095 for byte and 32-bit memory accesses

```
ldr r2, [r3,#20]
str r2, [r3,#20]
```

- The location accessed is calculated by adding the contents of r3 with the constant 20 or 0x14 in hexadecimal
- If r3 is 0x400, then the address is 0x414
  - r2 <= M[0x414]
  - M[0x414] <= r2

Memory Address Calculations

- This method and the first are identical
- The first method uses a constant 0
- Thus the first method could have been written as:

```
ldr r2, [r3,#0]

str r2, [r3,#0]
```

Memory Address Calculations

- Two registers can be used in the address calculation

```
ldr r2, [r3,r1]

str r2, [r3,r1]
```

- The contents of r3 and r1 are added to give the location to access
- If r3 is 0x2000 and r1 is 0x100, then the location is 0x2100

Memory Address Calculations

- Subtraction can be specified with:

```
ldr r2, [r3,-r1]

str r2, [r3,-r1]
```

- The r1 register is subtracted from r3 to yield the address
- If r3 is 0x2000 and r1 is 0x100, then the location is 0x1f00

Memory Address Calculations

- In the fourth method, the second register is shifted left, shifted right, or rotated by a fixed amount and the added to the first register
- Some examples are:

```
ldr r2, [r3, r1, lsl #2]

str r2, [r3, r1, lsl #2]
```

Memory Address Calculations

```
ldr r2, [r3, r1, lsl #2]

str r2, [r3, r1, lsl #2]
```

- Assuming
  - r3 contains 0x800
  - r1 is 0x8
- Location accessed is calculated by shifting "r1" 2 position to the left (i.e., multiplying by 4)
- Then adding r3
- The value is 0x800 + 4*0x8, which is 0x820
- r2 <= M[0x820] or M[0x820] <= r2
- This last method is often used to implement array access.

Sum Array With 32-bit Integers (sum_array.S)

- r0 = address to the start of the array
- r1 = number of elements to sum
- Moves r0 to r2
  - r2 now contains the start of the array
- r0 is used to hold the sum
- r0 is set to zero with eor r0, r0, r0
- r3 holds the value loaded from memory
- The loop sums the array entries from the end index to the start

```
sum_array:
        mov   r2, r0
        eor   r0, r0, r0    // r0 = 0
top:
        subs  r1, r1, #1
        blt   done
        ldr   r3, [r2, r1, lsl #2]
        add   r0, r0, r3
        b     top
done:
        mov   pc, lr
```

Sum Array With 32-bit Integers (sum_array.S)

- subs r1, r1, #1 subtracts 1 from the count
- If the count is less than 0
  - loop is terminated by blt done
- An entry in the array is loaded with:

  ldr r3, [r2, r1, lsl #2]

```
sum_array:
        mov   r2, r0
        eor   r0, r0, r0    // r0 = 0
top:
        subs  r1, r1, #1
        blt   done
        ldr   r3, [r2, r1, lsl #2]
        add   r0, r0, r3
        b     top
done:
        mov   pc, lr
```

Sum Array With 32-bit Integers (sum_array.S)

- The index, r1 is scaled by 4 to account for the 32 bits
  - 4 bytes per each integer in the array
  - lsl #2 scales by 4
- The address is calculated by r2 + r1*4
  - r2 is base address
  - r1*4 is next number address
- r3 is added to r0 to form the sum
- The code then branches to the top of the loop to load the next entry
- Test with "sum_array_main.c"

```
arm-linux-gnueabi-gcc --static
sum_array_main.c sum_array.S
```

```
sum_array:
        mov   r2, r0
        eor   r0, r0, r0    // r0 = 0
top:
        subs  r1, r1, #1
        blt   done
        ldr   r3, [r2, r1, lsl #2]
        add   r0, r0, r3
        b     top
done:
        mov   pc, lr
```

Sum Array With 32-bit Integers

- A C-language version of sum_array
- Hand written assembler is very similar to the compiler generated code

```
arm-linux-gnueabi-gcc -march=armv4 -Os -S sum_array.c
```

```
int sum_array( int arr[], int num ) {
    int sum = 0;
    while( --num >= 0 ) {
        sum += arr[num];
    }
    return sum;
}
```

Sum Array With 32-bit Integers

- More straight forward C solution

```
arm-linux-gnueabi-gcc -march=armv4 -Os -S sum_array_v1.c
```

```
int sum_array( int arr[], int num ) {
    int sum = 0;
    for( int i = 0; i < num; i++ ) {
        sum += arr[i];
    }
    return sum;
}
```

Initializing A Sequence (init_seq.S)

- An array is initialized with the sequence 0, 1, 2, ...
- r0 contains start of the array
- r1 number of entries
- str r2, [r0, r2, lsl #2]
  - M[r0 + 4*r2] <= r2
- movhs pc, lr return from the routine
- Conditional
  - r2 is higher or the same as r1
- The condition codes are set by cmp r2, r1

```
init_seq:
        mov   r2, #0
top:
        cmp   r2, r1
        movhs pc, lr
        str   r2, [r0, r2, lsl #2]
        add   r2, r2, #1
        b     top
```

Dot Product (dot_product.S)

$$[1, 2, 3] \cdot [4, 5, 6] = 1\times4 + 2\times5 + 3\times6 = 32$$

- This function accepts three arguments
  - The location of the array a
  - The location of the array b
  - The number of elements
- Since this function uses r4, r5, and r6, these registers are saved with push {r4,r5,r6}
- At the end of this function the values are restored with pop {r4,r5,r6}

Dot Product (dot_product.S)

- A loop from top to done is used the calculate the dot product
- Each element is read with a ldr instruction
- The mla r0, r5,r6, r0 calculates r0 = (r5*r6) + r0
- mla stands for multiply and accumulate
- r3 is used to index into the array

```
top:
        cmp   r3, r2
        bhs   done
        ldr   r5, [r4, r3, lsl #2]
        ldr   r6, [r1, r3, lsl #2]
        mla   r0, r5, r6, r0
        add   r3, r3, #1
        b     top
done:
        pop  {r4,r5,r6}
        mov   pc, lr
```

String Capitalize (capitalize.S)

- Strings in C are represented by byte arrays
- The end of the string is indicated by a byte with a value of zero
- This zero byte is called a NUL
- Code in capitalize.S copies a string to another array and any lower case letters are converted to uppercase

String Capitalize (capitalize.S)

- Bytes a loaded from memory with the ldrb instruction
- A byte is stored with the strb instruction
- The teq instruction tests if loaded bytes is zero
- The loop is terminated if it is zero
- The byte is then checked to see if it is lower case

```
top:
        ldrb    r3, [r1, r2]
        teq     r3, #0
        beq     done
        cmp     r3, #'a
        blt     skip
        cmp     r3, #'z
        bgt     skip
        bic     r3, r3, #0x20
skip:
        strb    r3, [r0, r2]
        add     r2, r2, #1
        b       top
done:
        strb    r3, [r0, r2]
        mov     pc, lr
```

String Capitalize (capitalize.S)

- The bic r3, r3, #0x20 (bit clear) instruction converts a lower case character to upper case
  - A = 0100 0001
  - a = 0110 0001 AND ~(0010 0000) = 0110 0001 AND 1101 1111 = 0100 0001 = A
- The last strb instruction before mov pc, lr places a zero byte in the destination string

```
top:
        ldrb    r3, [r1, r2]
        teq     r3, #0
        beq     done
        cmp     r3, #'a
        blt     skip
        cmp     r3, #'z
        bgt     skip
        bic     r3, r3, #0x20
skip:
        strb    r3, [r0, r2]
        add     r2, r2, #1
        b       top
done:
        strb    r3, [r0, r2]
        mov     pc, lr
```

Unit 5 done

Next: Micro-Architecture of ARM

Questions?