Overview

- Memory Access in Assembly
- Data Structures in Assembly

Review: Instruction Set (ARM 7TDMI)

- Set of instruction that a processor can execute
- Instruction Categories
  - Data Processing or Computational (Logical and Arithmetic)
  - Load/Store (Memory Access: or transferring data between memory and registers)
  - Control Flow (Jump and Branch)
  - Floating Point - coprocessor
  - Memory Management

| Registers | r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15 (PC) |
|-----------|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|

Review: ARM Instructions So far

- Data Processing Instructions with shift and rotate
  - lsl, lsr, asr, ror

- Multiplications
  - mul, mla, umull, umlal, smull, smlal
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But ARM arithmetic instructions only operate on registers, never directly on memory.

**Data transfer instructions** transfer data between registers and memory:
- Memory to register
- Register to memory

Data Transfer: Memory to Reg (#1/4)

- To transfer a word of data, we need to specify two things:
  - Register: specify this by number (r0 – r15)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

Data Transfer: Memory to Reg (#2/4)

- To specify a memory address to copy from, specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes), or another register which contains an offset

- The desired memory address is the sum of these two values.

- Example: \([v1, #8]\)
  - specifies the memory address pointed to by the value in \(v1\), plus 8 bytes

- Example: \([v1, v2]\)
  - specifies the memory address pointed to by the value in \(v1\), plus \(v2\)

Data Transfer: Memory to Reg (#3/4)

- Load Instruction Syntax:
  1. \([2, [3, 4]]\)
  - where
    1) operation name
    2) register that will receive value
    3) register containing pointer to memory
    4) numerical offset in bytes, or another shifted index register

- Instruction Name:
  - `ldr` (meaning Load register, so 32 bits or one word are loaded at a time from memory to register)
**Data Transfer: Memory to Reg (#4/4)**

- **Example:** `ldr a1, [v1, #8]

  This instruction will take the pointer in `v1`, add 8 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `a1`.

- **Notes:**
  - `v1` is called the base register.
  - 8 is called the offset.
  - Offset is generally used in accessing elements of array: base reg points to beginning of array.

- **Example:** `ldr a1, [v1, v2]

  This instruction will take the pointer in `v1`, add an index offset in register `v2` to it, and then load the value from the memory pointed to by this calculated sum into register `a1`.

- **Notes:**
  - `v1` is called the base register.
  - `v2` is called the index register.
  - Index is generally used in accessing elements of array using a variable index: base reg points to beginning of array.

**Data Transfer: Other Mem to Reg Variants (#1/2)**

- **Pre Indexed Load Example:**
  
  ```
  ldr a1, [v1, #12]!
  ```

  This instruction will take the pointer in `v1`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `a1`.

  Subsequently, `v1` is updated by computed sum of `v1` and 12, (`v1 ← v1 + 12`).

- **Pre Indexed Load Example:**
  
  ```
  ldr a1, [v1, v2]!
  ```

  This instruction will take the pointer in `v1`, add an index offset in register `v2` to it, and then load the value from the memory pointed to by this calculated sum into register `a1`.

  Subsequently, `v1` is updated by computed sum of `v1` and `v2`, (`v1 ← v1 + v2`).

**Data Transfer: Other Mem to Reg Variants (#2/2)**

- **Post Indexed Load Example:**
  
  ```
  ldr a1, [v1], #12
  ```

  This instruction will load the value from the memory pointed to by value in register `v1` into register `a1`.

  Subsequently, `v1` is updated by computed sum of `v1` and 12, (`v1 ← v1 + 12`).

- **Example:** `ldr a1, [v1], v2

  This instruction will load the value from the memory pointed by value in register `v1`, into register `a1`.

  Subsequently, `v1` is updated by computed sum of `v1` and `v2`, (`v1 ← v1 + v2`).

**Data Transfer: Reg to Memory (1/2)**

- **Also want to store value from a register into memory**

- **Store instruction syntax is identical to Load instruction syntax**

- **Instruction Name:**

  ```
  str (meaning Store from Register, so 32 bits or one word are stored from register to memory at a time)
  ```
Data Transfer: Reg to Memory (2/2)

- Example: `str a1, [v1, #12]`
  This instruction will take the pointer in `v1`, add 12 bytes to it, and then store the value from register `a1` into the memory address pointed to by the calculated sum.

- Example: `str a1, [v1, v2]`
  This instruction will take the pointer in `v1`, adds register `v2` to it, and then store the value from register `a1` into the memory address pointed to by the calculated sum.

Data Transfer: Other Reg to Mem Variants (#1/2)

- Pre Indexed Store Example:
  `str a1, [v1, #12]!`
  This instruction will take the pointer in `v1`, add 12 bytes to it, and then store the value from register `a1` into the memory address pointed to by the calculated sum.
  Subsequently, `v1` is updated by computed sum of `v1` and 12, (`v1 ← v1 + 12`).

- Pre Indexed Store Example:
  `str a1, [v1, v2]!`
  This instruction will take the pointer in `v1`, adds register `v2` to it, and then store the value from register `a1` into the memory address pointed to by the calculated sum.
  Subsequently, `v1` is updated by computed sum of `v1` and `v2` (`v1 ← v1 + v2`).

Data Transfer: Other Reg to Mem Variants (#2/2)

- Post Indexed Store Example:
  `str a1, [v1], #12`
  This instruction will store the value from register `a1` into the memory address pointed to by register `v1`.
  Subsequently, `v1` is updated by computed sum of `v1` and 12, (`v1 ← v1 + 12`).

- Post Indexed Store Example:
  `str a1, [v1], v2`
  This instruction will store the value from register `a1` into the memory address pointed to by register `v1`.
  Subsequently, `v1` is updated by computed sum of `v1` and `v2`, (`v1 ← v1 + v2`).

Pointers v. Values

- Key Concept: A register can hold any 32-bit value. That value can be a (signed) `int`, an unsigned `int`, a pointer (memory address), etc.

- If you write `add v3, v2, v1` then `v1` and `v2` better contain values

- If you write `ldr a1, [v1]` then `v1` better contain a pointer

- Don’t mix these up!
**Addressing: Byte vs. halfword vs. word**

- Every word in memory has an **address**, similar to an index in an array.
- Early computers numbered words like C numbers elements of an array:
  - `Memory[0], Memory[1], Memory[2], ...`
  
  Called the **address** of a word.
- Computers needed to access **8-bit bytes, half words (2 bytes/halfword)** as well as **words (4 bytes/word).**
- Today machines address memory as bytes, hence
  - Half word addresses differ by 2:
    - `Memory[0], Memory[2], Memory[4], ...`
  - Word addresses differ by 4:
    - `Memory[0], Memory[4], Memory[8], ...`

**Compilation with Memory**

- What offset in `ldr` to select `my_Array[8]` in C?
  - `4x8=32` to select `my_Array[8]`: byte v. word
- Compile by hand using registers:
  - `g = h + my_Array[8];`
    - `g`: v1, `h`: v2, v3: base address of `my_Array`
- 1st transfer from memory to register:
  - `ldr v1, [v3,#32]`; `v1` gets `my_Array[8]`
    - Add 32 to v3 to select `my_Array[8]`, put into v1
- Next add it to `h` and place in `g`
  - `add v1,v2,v1`; `v1 = h + my_Array[8]`

**Same thing in pictures**

- `v3` contains the address of the Base of the `my_Array`
  - `ldr v1, [v3,#32]` Adds offset "8 x 4 = 32" to select `my_Array[8]`, and puts into `v1`
  - The value in register `v3` is an address
  - Think of it as a pointer into memory
- `add v1, v2,v1` The value in register `v1` is the sum of `v2` and `v1`.

**Compile with variable index**

- What if array index not a constant?
  - `g = h + my_Array[i];`
  - `g`: v1, `h`: v2, i: v3, v4: base address of `my_Array`
- To load `my_Array[i]` into a register, first turn `i` into a byte address; multiply by 4
- How multiply using adds?
  - `i + i = 2i, 2i + 2i = 4i`
  - `mov a1,v3`; `a1 = i`
  - `add a1,a1`; `a1 = 2*i`
  - `add a1,a1`; `a1 = 4*i`
  
  Better alternative: `mov a1, v3, lsl #2`
Compile with variable index, con’t

- Now load $my\_Array[i] = my\_Array[0] + 4*i$ into v1 register:
  
  ```
  ldr v1, [v4, a1] ; v1= my\_Array[i]
  ```

- Finally add to $h$ to it and put sum in $g$:
  
  ```
  add v1, v1, v2 ; g = h + my\_Array[i]
  ```

Compile with variable index: Summary

- C statement:
  
  ```
  g = h + my\_Array[i];
  ```

- Compiled ARM assembly instructions:
  
  ```
  mov a1, v3, lsl #2 ; a1 = 4*i
  ldr v1, [v4, a1] ; v1= my\_Array[i]
  ```

- Finally add to $h$ to it and put sum in $g$:
  
  ```
  add v1, v1, v2 ; g = h + my\_Array[i]
  ```

Compile with variable index Example

- Compile this into ARM code:
  
  ```
  B\_Array[i] = h + A\_Array[i];
  ```

  - $h$: v1, $i$: v2, v3: base address of $A\_Array$, v4: base address of $B\_Array$

Compile with variable index Example (Solution)

- Compile this C code into ARM:

  ```
  B\_Array[i] = h + A\_Array[i];
  ```

  - $h$: v1, $i$: v2, v3: base address of $A\_Array$, v4: base address of $B\_Array$

  ```
  mov a1, v2, lsl #2 ; a1 = 4*i
  ldr a2, [v3, a1] ; v4 + a1 = ; addrB\_Array[i]
  ; a2= A\_Array[i]
  add a2, a2, v1 ; a2 = h + A\_Array[i];
  ```

  - $v4 + a1 = \text{addrB\_Array[i]}$; $a2= A\_Array[i]$

  ```
  str a2, [v4, al] ; v4 + a1 = ; addrB\_Array[i]
  ; B\_Array[i]= a2
  ```
Notes about Memory

Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.

- Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
- So remember that for both `ldr` and `str`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned).

More Notes about Memory: Alignment (#1/2)

ARM requires that all words start at addresses that are multiples of 4 bytes.

- Called Alignment: objects must fall on address that is multiple of their size.
- Some machines like Intel allow non-aligned accesses.

More Notes about Memory: Alignment (#2/2)

Non-Aligned memory access causes byte rotation in right direction within the word.

- `ldr a1, 0x80 → a1 = 0x0982a2e
- `ldr a1, 0x81 → a1 = 0x2e0982a2
- `ldr a1, 0x82 → a1 = 0xa22e0982
- `ldr a1, 0x83 → a1 = 0x82a2e09`
Role of Registers vs. Memory

° What if more variables than registers?
  • Compiler tries to keep most frequently used variable in registers
  • Writing less common to memory: spilling

° Why not keep all variables in memory?
  • Smaller is faster: registers are faster than memory
  • Registers more versatile:
    - ARM Data Processing instructions can read 2, operate on them, and write 1 per instruction
    - ARM data transfer only read or write 1 operand per instruction, and no operation

“And in Conclusion…” (#1/2)

° In ARM Assembly Language:
  • Registers replace C variables
  • One Instruction (simple operation) per line
  • Simpler is Better
  • Smaller is Faster

  ° Memory is byte-addressable, but ldr and str access one word at a time.

  ° A pointer (used by ldr and str) is just a memory address, so we can add to it or subtract from it (using offset).

  ° Word Addresses in Memory should be word aligned

“And in Conclusion…” (#2/2)

° New Instructions:
  ldr, str