LC-3 ISA
Memory Instructions

Based on slides © McGraw-Hill
Additional material © 2004/2005 Lewis/Martin
Modified by Diana Paletta

Memory Access

Instruction memory access
- IR = MEM[PC]
- Implicit to von Nuemann Model

Data Memory access must done explicitly via instructions
- LOAD: read data from memory into register
  - i.e. reg <- copy of data in mem
- STORE: write data from register to memory
  - i.e. mem[addr] <- copy of reg

LC3/LC4 are RISC style machines
- Separate instruction for data access
- Unlike x86 (CISC) where an instruction where instruction can do mix operations e.g. reg <- reg + mem[addr]
- Trade off: code size vs. fast hardware

PC-Relative Addressing Mode

Want to specify address directly in the instruction
- But an address is 16 bits, and so is an instruction!
- After subtracting 4 bits for opcode and 3 bits for register, we have 9 bits available for address

Possible Solution
- Take 9 bits [8:0] in instruction
- Sign extended to 16 bits
- Add it to the PC (of next instruction) to form address

Limitations
- What is the max range of addresses we span from PC?

Base + Offset Addressing Mode

Problem
- With PC-relative mode, can only address memory locations “near” the instruction
- What about the rest of memory?

Solution
- Use a register to generate a full 16-bit address

Logistics
- 4 bits for opcode, 3 for src/dest register, 3 bits for base register
  - Base Register is setup using LEA instruction
- Remaining 6 bits are used as a signed offset
- Offset is sign-extended before adding to base register
  - i.e., Instead of adding offset to PC, add it to base register

Example: LDR: R1 <- M[R2+SEXT(IR[5:0])]


Load Effective Address

Problem
- How can we compute address without also LD/ST-ing to it?

Solution
- Load Effective Address (LEA) instruction
- Used to generate address
  - initializes a register with an address very close to the initializing instruction

Logistics
- Store address in destination register (not data at that address)
- Does not access memory
- E.g. LEA: R1 <- PC + SEXT(IR[8:0])
- For LC4 LEA is pseudo instruction
  - implemented using CONST instruction - more on it later!

Global Variable

Managed by compiler
- or assembly programmer

Variables assigned static addresses
- i.e. known from beginning and location is constant through the execution

Declaring and Initializing global variable

```
int global_counter = 1; // in C language

;In LC3/4 Assembly
.DATA .following labels/directives are data
.ADDR x2000 ;following data is at this static addr

;label (name of the variable) followed by init value
.FILL is pseudo instr
global_counter .FILL x0001
```
**Reading & Writing global variable**

```c
int global_counter = 1;
void inc_global_counter()
{
    global_counter++;
}
```

Here is assembly translation:

- LEA R4, global_counter ; R4 = &global_counter (like in C)
- LDR R3, R4, #0 ; R3 = global_counter (like in C)
- ADD R3, R3, #1 ; increment by 1
- STR R3, R4, #0 ; store the updated value back

---

**Global Array Declaration & Initialization**

```c
int global_array[10]; //C declaration
void initialize()
{
    int i = 0;
    for(i = 0; i < 10; ++i)
    {
        global_array[i] = i;
    }
}
```

Assembly

**Assembly**

```
INITIALIZE
    AND R4, R4, #0 ; R4 = i = 0
    LEA R5, GLOBAL_ARRAY ; R5 = addr of first element of the array
    LOOP
        STR R4, R5, #0 ; store value of i at the ith array location
        CMP R4, #4 ; if i == 4 then DONE
        BRz DONE
        ADD R4, R4, #1 ; i++
        ADD R5, R5, #1 ; next location in array
        BRnzp LOOP
    DONE
```

---

**Other LC3 memory operations**

**PC-relative**

- Addr = PC + 1 + SEXT(IMM9)
- LD, ST

**Memory Indirect**

- addr = mem[PC+1+SEXT(IMM9)]
- Instructions: LDI, STI

**LC4 doesn’t have these**

- Mainly because PC-relative jumps are limiting, which why there is a need for Memory Indirect Operation