The Simple Decimal Computer

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- A simple decimal example illustrates how von Neumann computers work without worrying about binary representation and more complicated instruction sets.

B. Outcomes

After this lecture, you should know

- How to trace instruction execution for a simple computer.

C. Simple Decimal Computer

- Eventually we'll be looking at the textbook's LC-3 computer, but before that, let's look at a simpler computer. To make it simple, we'll use decimal data, the instruction set will be very limited and we'll use absolute addressing.
- The SDC (a Simple Decimal Computer) has a hundred memory locations numbered `00` – `99` and ten arithmetic registers named `R0`, `R1`, …, `R9`. Each memory location and register holds a signed 4-digit number. Below, we indicate the contents of register *R* using `Reg`[*R*] and the contents of memory location *MM* using `Mem`[*MM*].
- Instructions are 4 digit numbers and have the form $\pm NRMM$, where *N* is an opcode (0-9), *R* is a register number (`0-9`), and *MM* is a memory location (`00-99`). The sign is generally ignored (except for "immediate" instructions; see opcodes 5 and 6 below).
- It's important to differentiate between opcodes 1 (`LD`) and 5 (`LDM`). Both set `Reg`[*R*] to a value: *Load* (`LD`) uses `Mem`[*MM*], the value at memory location *MM*, but *Load Immediate* (`LDM`) uses *MM* literally (we say it's an **immediate operand** when it's part of the instruction this way). Opcodes 3 (`ADD`) and 6 (`ADDM`) are similar: Add sets `Reg`[*R*] ← `Reg`[*R*] + `Mem`[*MM*] but *Add Immediate* sets `Reg`[*R*] ← `Reg`[*R*] + *MM*.

    - Note[1]: For `LDM` and `ADDM`, we take the sign of *MM* to be the sign of the overall instruction: `5123` says to load immediate `R1` with `23`; `-5123` says to load immediate `R1` with `-23`. This difference is handy if you want to decrement a register. (E.g., `-6201` sets `R2` ← `R2 - 1`.)

- For I/O, instead of using *R* as a register, we use it to specify the kind of I/O to do. For operations 90, 91, 93, and 94, we ignore *MM*.
    - Operation `90` reads a character from the keyboard and copies its ASCII numeric representation to `R0`.
    - Operation `91` is the opposite of operation 90: It prints the character whose ASCII numeric representation is in `R0`.
    - Operation `92` prints out a string. The string should be represented as a sequence of characters in memory (one character per location) with one extra trailing memory location containing `0`. The location of the first character is specified by *MM*. E.g., for instruction `9225`, if locations 25 – 28 contain `65`, `66`, `67`, `0`, then we print `ABC`.
    - Operation `93` dumps the contents of the control unit (`PC`, `IR`, and data registers) to the display.
    - Operation `94` dumps the contents of memory to the display.
- **SDC Execution**: When the SDC powers up, it reads a sequence of 4-digit numbers into memory locations `00`, `01`, …. When it reaches the end of the numbers, it sets all registers to `0`, sets the `PC` to location `00`, sets the *Running* flag to true, and begins the instruction cycle, which is a microcode loop:

    while *Running* {
        Fetch instruction: `IR` ← `Mem`[`PC`]*;* `PC++` [2]
        Decode instruction: Set Op, R, and MM to `IR`[3]*,* `IR`[2]*,* `IR`[1:0]*.*
        Get operands, execute instruction, store results:
            if *Op* = `0` **then** *Running* ← *False*
            else if *Op* = `1` **then** ….
    }

---

[1] This is a change from previous semesters.

[2] If the `PC` is < 0 or > 99, the machine halts. This is a change from previous semesters.

Opcode	***Meaning***	***Microcode***
`0`	`HALT` execution. (Ignore R and MM.)	*Running* ← *false*
`1`	`LD` (Load) `Reg`[R] with the value of memory location MM.	`Reg`[R] ← `Mem`[MM]
`2`	`ST` (Store) Copy the value of `Reg`[R] into memory location MM	`Mem`[MM] ← `Reg`[R]
`3`	`ADD` contents of location MM to `Reg`[R]	`Reg`[R] `+=` `Mem`[MM]
`4`	`NEG`: Set `Reg`[R] to its arithmetic negative (ignore MM)	`Reg`[R] ← (– `Reg`[R])
`5`	`LDM` (Load immediate): Load `Reg`[R] with MM (not the contents of MM)**.	`Reg`[R] ← MM
`6`	`ADDM` (Add immediate): Add MM to `Reg`[R].**	`Reg`[R] `+=` MM
`7`	`BR` (branch unconditionally): Go to location MM (ignore R)	`PC` ← MM
`8`	`BRP` (branch if positive): If `Reg`[R] is > 0, go to location MM. If not, just continue on to the next instruction.	**if** `Reg`[R] > 0 **then** `PC` ← MM
`90`	`GETC` Read a character and copy its ASCII representation into `R0`. (`A` sets `R0 = 65`, etc.)	*Reg*[0] ← *Keyboard*
`91`	`OUT` ~~`PUTC`~~ Print the character whose ASCII representation is in `R0`. (`A` for `65`, etc.)	*Print Char*(*Reg*[0])
`92`	`PUTS` Print a string (i.e., the characters) at locations MM, $MM+1$, …. Stop (and don't print) when we get to a location that contains 0. (Don't print the zero.)	`temp` ← MM **while** `Mem`[`temp`] ≠ 0 *Print* `Mem`[`temp`]; `temp++`
`93`	`DMP` Print out the values of the control unit registers (the `PC`, `IR`, and `R0` – `R9`)	*Dump Control Unit*
`94`	`MEM` Print the values in memory in a 10×10 table.	*Dump Memory*
`95-99`	Ignore instruction	

SDC Instruction Set Architecture[3]

[3] If the sign of the overall instruction is negative, take MM to be negative.

- **Example 1**: Add `1` to the contents of memory location `99`, using `R3` as temporary storage:

  `1399 ; R3 ← Mem[99]`
  (Load `R3` from contents of memory location `99`)
  `6301 ; R3 ← R3 + 1`
  `2399 ; Mem[99] ← R3` (Store `R3` into memory location `99`)

- **Example 2**: Set `R1` to `-1`
  - Version (a): Use load immediate

    `-5101 ; R1 ← -1` (Load immediate `R1` with `-1`)
  - Version (b): Load it with `1` and taking its negative.

    ```
    5101  ; R1 ← 1 (Load immediate R1 with 1)
    4100  ; R1 ← -R1
    ```
  - Version (c): Load `R1` with a location that contains `-1`

    ```
    4199  ; R1 ← 1 (Load immediate R1 with 1)
    ```
    where location `99 = -1`

- **Example 3**: Add `R1` to `R0` and branch to location `57` if the result is `> 0`; branch to location `62` if the result is ≤ 0. We use location `98` as temporary storage.

  `2198 ; Mem[98] ← R1` (Temporarily save `R1` into location `98`)
  `3098 ; R0 ← R0 + Mem[98] = R0 + R1`
  `8057 ; Branch to location 57 if` current `R0 > 0`
  `;` (i.e., if old `R0 + R1 > 0`)
  `7062 ; (else) Branch to location 62`

- The branch to location `62` is unconditional, but we get to location `62` only if we don't do the branch if `R0` > `0`, so we branch to `62` only if `R0` is ≤ `0`.

- **Example 4**: Print "`Hello, world!`" Assume the first instruction is at location `00`.

  ```
  9202  ; PRINT string starting at location 02.
  0000  ; HALT
  0072  ; 'H'
  0101  ; 'e'
  0108  ; 'l'
  ```

```
0108  ; 'l'
0111  ; 'o'
0044  ; ','
0032  ; ' '
0119  ; 'w'
0111  ; 'o'
0114  ; 'r'
0108  ; 'l'
0100  ; 'd'
0033  ; '!'
0000  ; end of string
```

- The address of the string is part of the `PRINT` instruction, so it's **really** hard-coded in there.
  - If we ever change the location of the string (e.g., by making the program longer), we'll need to update the `PRINT` instruction to use the new location of the string.
  - A partial workaround is to put the string at an address far away from the program so that lengthening the program is less likely to cause a problem.

D. Instructions vs Data

- When we write low-level programs, we may think of some memory locations as having instructions and other locations as having data, but the CPU treats everything as data except for the value inside the **Instruction Register**.
  - E.g., in Example 4, we intend the `0000` at location 1 to be a `HALT` instruction, but in a different context it can be treated as data.
    - If we were to print the string starting at location `01` (by making the `PRINT` instruction `9201` instead of `9202`), we would print no characters because we'd see the `0000` at location `01` as an end-of-string.
- On the other hand, we can also treat what we think of as data as instructions ("execute our data"):
  - E.g., still in Example 4, if we take out the `HALT 0000` at location `01` so that the '`H`' is now at location `01`, the '`e`' is at location `02`, etc., then the `PRINT 9202` instruction will print "`ello, world!`".

- Then we'll execute the next instruction (the one at location `01`). Location `01` contains `0072` (the letter '`H`') but as an instruction, it means **`HALT`**, so our program will stop.

The Simple Decimal Computer

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- A simple decimal example illustrates how von Neumann computers work without worrying about binary representation and more complicated instruction sets.

B. Outcomes

After this activity, you should be able to

- Describe the basic design of a von Neumman computer and discuss how it differs from other architectures.
- Describe the parts of the instruction cycle and what happens during them.
- Trace instruction execution for a simple computer.

C. Questions

Questions 1 – 7 refer to the Simple Decimal Computer from the notes. Feel free to use `R9`, `R8`, … as temporary registers if you need them. If you need temporary memory locations, use locations `99`, `98`, …. Unless otherwise specified, assume the code for each question starts at location `00`.

1. Write code that takes the contents of memory location `90`, doubles it, and stores it back in location `90`.
2. Write some code that sets `R0 ← R0 – R1`. Use memory location `99` as temporary storage. To subtract `R1`, you'll need to take its negative: `R0 ← R0 + NEG R1`
3. (a) Write a branch instruction that branches to location `12` if memory location `95` is positive.

   (b) Write code that uses a branch instruction(s) to go to location `12` if `M[95]` is non-positive (i.e., ≤ 0). Assume your code starts at location `30`. (To do a branch-on-not-positive to location `12`, first write an unconditional branch to location `12`; then just before it, put a branch-on-positive that jumps over the unconditional branch.)

(c) Write code that goes to location `12` if `M[95]` ≥ 0. Assume your code starts at location 30. (Hint: `M[95]` ≥ `0` iff `M[95]+1` ≥ `1`.)

4. Say location `00` contains `1200`. What happens if we execute the instruction at location `00`?

5. Write code to implement the following loop; use **`BRP`** to jump to the top of the loop. (Version (a): To decrement `R0`, assume `M[90]` = **`-1`**. Version (b): To decrement `R0`, use `ADDM` with **`-1`**.)

   `R0 ← M[20];` **`{… ; --R0; }`** **`while`** `(R0 > 0);`

6. Write code to implement the following loop; use **`BRP`** to exit the loop and **`BR`** to jump to the top of the loop. Let **`XX`** be the first location after the loop.

   `R0 ← M[20];` **`while`** `(R0 ≤ 0)` **`{… ; ++R0; }`** ;

7. Write code to implement the following loop; use a **`BRP`** and **`BR`** to exit the loop and a second **`BR`** to jump to the top of the loop. Let **`XX`** be the first location after the loop.

   `R0 ← M[20];` **`while`** `(R0 > 0)` **`{… ; --R0; }`** ;

Solution

1. Set `M[90]` ← `2*M[90]`: (Uses `R9` as a temporary register.)

```
1990 ; R9 ← M[90]
3990 ; R9 ← R9 + M[90]
2990 ; M[90] ← R9
```

2. Set `R0` ← `R0 – R1` (uses location `99` as temporary storage; destroys `R1`):

```
4100 ; R1 ← - R1
2199 ; M[99] ← R1
3099 ; R0 ← R0 - (original value of) R1
```

3. (a) Go to location `12` if memory location `95` is positive:

```
1095 ; R0 ← M[95]
8095 ; go to 95 if M[95] > 0
```

(b) Go to location `12` if `M[95] ≤ 0`:

```
00: 1995 ; R9 ← M[95]
01: 8903 ; go to 03 if M[95] > 0
02: 7912 ; go to 12 if M[95] ≤ 0
03: (next instruction to execute, if M[95] > 0)
```

(c) Go to location `12` if `M[95] ≥ 0`:

```
00: 1995 ; R0 ← M[95]
01: 6901 ; R0 ← M[95]+1
02: 7912 ; go to 12 if M[95]+1 > 0 (if M[95] ≥ 0)
03: (next instruction to execute, if M[95] < 0)
```

4. Since `M[00]` = `1200`, executing the instruction at location `00` means we execute `1200` as an instruction, so we loads `R2` with the value of `M[00]`, namely `1200`.

5. The loop

```
R0 ← M[20]; {
    …
```

```
        --R0;
    } while (R0 > 0) {
```

is

```
00: 1020  ; R0 ← M[20]
01: …
…
```

(Version a: … : `3090 ; R0 ← R0-M[90] = R0-1`)

(Version b: … : `-6001 ; R0 ← R0-1`)

```
… : 8001  ; continue loop if R0 > 0
…
90:   -1  ; a constant
```

6. The loop

```
R0 ← M[20];
while (R0 ≤ 0) {
    … ;
    ++R0;
} // assume instruction after loop is at XX
```

is

```
00: 1020   ; R0 ← M[20]
01: 80XX   ; Exit loop if R0 > 0
…
… : 6001   ; R0++
… : 7001   ; continue loop
XX: (first location after the loop)
```

7. The loop

```
R0 ← M[20];
while (R0 > 0) {
    … ;
    --R0;
} // assume instruction after loop is at XX
```

is

```
00: 1020  ; R0 ← M[20]
01: 8003  ; continue loop if R0 > 0
02: 7030  ; exit loop if R0 ≤ 0
03: …     ; (top of loop body)
… : …
… :-6001   ; R0 ← R0-1
… : 7002   ; continue loop
XX: (first location after the loop)
```

