CSCI 1103 Computer Science 1 Honors

Fall 2022

Robert Muller - Boston College

Lecture Notes

Week 10: A Simple Virtual Machine

Topics:

1. A Simple Virtual Machine
2. Data "Structures"

---

1. A Simple Virtual Machine

Our phones, our laptops, the university's mainframe computers, they are actually quite similar. All are based on a very flexible design developed in the 1940s by the Hungarian-born mathematician John von Neumann (among others). It is remarkable that the design, sometimes referred to as the stored-program computer, or the von Neumann architecture, has persisted in the face of the never-ending revolution in technology.

The heart of the idea is very simple. We all know that computers store information in the form of bits, the ubiquitous binary digits 1 and 0. (A string of 8 of them is called a byte. Computers actually contain two kinds of memory, persistent (or non-volatile) storage that retains information when the computer is off. This is where our files are stored. The other type, ephemeral (or volatile) storage retains information when the computer is powered-up but is erased when the computer is off. This type of memory is sometimes called random-access memory (RAM) because each storage cell (i.e., each byte) has an address and the individual cells can be efficiently accessed in random order.

The other piece to understand is that it's possible to design circuitry to manipulate the stored bits in various ways. We can design an addition circuit for example. When presented with input bits representing say, the integers 6 and 3, it will produce as output the bits representing the integer 9. We can design subtraction circuits, comparison circuits, we can design circuits that will load information from RAM or store information in RAM. And so on for the various sorts of simple operations that a computer can carry out. Most computers have a couple hundred operations.

Let's say we're building a computer and we've designed circuitry for say 8 operations. von Neumann's key idea was to assign a unique pattern of bits to each of the operations. Feel free to consider the pattern to be a numeral or number if you like. For example, we might assign the bit pattern 0010 to the ADD operation and the bit pattern 0110 to the SUB operation. For a machine with 16 instructions, we require 4 bit patterns (because $2^4 = 16$). These patterns (or *opcodes*) can be stored in the computer's RAM along with the data. This is an idea that was inspired by his knowledge of Alan Turing's work which was in turn inspired by the seminal work of Kurt

Godel.

Since operations usually have operands, the opcodes are usually packaged-up in RAM together with representations of their operands:

```
  ADD   opnd1, opnd2, opnd3
```

When executed, the above instruction would add the contents of opnd2 to the contents of opnd3 and store the result in opnd1. We'll call the combination of opcode and operands an instruction. To simplify the whole setup, instructions usually have a fixed size, on many computers, the are packed into 4 consecutive bytes of RAM. (And by the way, 4 consecutive bytes are usually referred to as a word of memory.)

If we know where in the RAM to look for them, we can execute any sequence of instructions that someone might like to load into the memory. This process is called, naturally enough, coding (aka programming). Instead of being fixed and rigid, stored-program computers are the essence of flexibilty! This might seem like a simple idea but it's also a powerful one and, as we said above, it has remained in place for over 70 years. It is the basic design of virtually all computers.

Essential Parts

In order to make this scheme work, it turns out to be most efficient to modularize the design, housing the above described circuitry, the "smarts" of the computer, in a central processing unit (or CPU). Modern CPUs contain many components, including:

- a subcomponent containing the circuits that carry out the instructions on their operands. This subcomponent is known as the the arithmetic and logic unit or ALU,
- a small set (typically 32 or 64) of very high-speed storage cells called registers. These are typically named R0, R1, ... Some are general purpose work areas while others have special purposes:

the PC (or program counter) register holds the address in RAM of the next instruction to be executed, the PSW (or program status word) register holds information about the outcomes of comparisons, etc the RA (or return address) register holds the address of the instruction to return to after a JSR. the Zero register holds the constant 0.

Operands

As we noted above, an instruction includes information about the required operands. For example, in the instruction

```
ADD   R0, R1, R2
```

registers R0, R1 and R2 are operands of the ADD instruction. When executed, this instruction would cause the computer to add the contents of registers R1 and R2 and deposit the sum in register R0. Note that this mutates R0.

Different sorts of instructions will have different sorts of operands. For example, the load instruction:

```
LOD    R0, 12(R1)
```

uses *indirect addressing*. Let base be the value in register R1. Then this instruction loads the bits stored in RAM address (base + 12), into register R0.

The Instruction Cycle

Given the above, a stored-program computer cycles through the following instruction cycle:

1. Fetch the instruction from the text segment location whose address is in the PC register;
2. Decode the instruction and fetch the operands, if any;
3. Increase the PC register to point to the next instruction;
4. Execute the instruction;
5. Go to 1.

The Simple Virtual Machine

SVM is simple because it only has the very basics of the full von Neumann architecture, it has neither a stack nor a heap. It is virtual because we'll actually implement it in (OCaml) software.

SVM has 8 registers and 16 instructions.

Registers

- Register **PC** is the program counter;
- Register **PSW** is the program status word;
- Register **RA** is the return address register;
- Register **Zero** holds the constant 0;
- Registers **R0** through **R3** are general purpose registers.

SVM Instructions

SVM instructions have two representations: in machine-readable binary and in human-readable symbolic form, what is known as assembly language. These generally match up one-for-one: each assembly instruction can be translated to one two-byte word of memory. The machine language form isn't the tack that we're pursuing here, we'll focus on the assembly.

In the descriptions below we'll follow the standard notation: **Rd**, **Rs** and **Rt** refer to one of the general purpose registers, with **Rd** denoting a *destination* of an operation and **Rs** and **Rt** denoting *source* registers. We'll use the symbol RAM to refer to the random access memory, the symbol addr to refer to a non-negative integer address in memory and the notation RAM[addr] to refer to the contents of location addr in the memory. We're not going to distinguish between data stored in the static data segment of an image or in the dynamic heap. We'll use the symbol disp to refer to an integer displacement that may (or may not) be added to the PC register to alter the flow of control.

> To illustrate machine language lets look at a possible representation of the instruction `add r3, r1, r2` which adds the words in registers `r1` and `r2` and places the sum in register `r3`. Since there are 16 instructions, we need $log_2(16) = 4$ bits to have a unique bit pattern for each instruction. Let's say the pattern for `add` is `0110`. The `add` instruction has 3 register operands. Since there are 8 registers, we need 3 bits to uniquely identify a register. Lets go with `000` for `R0`, `001` for `R1` and so forth. Then the instruction above: `add r3, r1, r2` might be represented by the following 16 bits.
>
> ```
> 1
> 5                      8   7                      0
> +------------+------------+------------+------------+
> | 0  1  1  0   x  0  1  1   x  0  0  1   x  0  1  0 |  = 0x6312
> +------------+------------+------------+------------+
>   add           r3           r1           r2
> ```
>
> Note that we've chosen to place the bits specifying the operation in the leftmost 4 bits. The `x`s denote "don't care" bits --- we didn't need 4 bits to identify a register, only 3 were needed. When the machine sees the pattern `0x6312` it is going to fetch the contents of `r1` and `r2`, route these through an add circuit and place the sum in `r3`.

All instructions leave the RA and PSW register alone unless specified otherwise.

0. **Lod Rd, offset(Rs)**: let base be the number in register **Rs**. Then this loads RAM[base + **offset**] into register **Rd**.
1. **Li Rd, number**: loads **number** into register **Rd**.
2. **Sto Rs, offset(Rd)**: let base be the number in register **Rd**, stores the contents of register **Rs** into location RAM[base + **offset**] in the memory.
3. **Mov Rd, Rs**: copies the contents of register **Rs** into register **Rd**.
4. **Add Rd, Rs, Rt**: adds the contents of registers **Rs** and **Rt** and stores the sum in register **Rd**.
5. **Sub Rd, Rs, Rt**: subtracts the contents of register **Rt** from **Rs** and stores the difference in register **Rd**.
6. **Mul Rd, Rs, Rt**: multiplies the contents of register **Rt** from **Rs** and stores the product in register **Rd**.
7. **Div Rd, Rs, Rt**: divides the contents of register **Rs** by **Rt** and stores the integer quotient in register **Rd**.
8. **Cmp Rs, Rt**: sets **PSW** = **Rs - Rt**. Note that if **Rs** > **Rt**, then **PSW** will be positive, if **Rs** == **Rt**, then **PSW** will be 0 and if **Rs** < **Rt**, then **PSW** will be negative.
9. **Jsr disp**: sets **RA** = **PC** and then **PC** = **PC** + **disp**.
10. **R**: sets **PC** = **RA**.
11. **Blt disp**: if **PSW** is negative, causes the new value of **PC** to be the sum **PC** + **disp**. Note that if **disp** is negative, this will cause the program to jump backward in the sequence of instructions. If **PSW** >= 0, this instruction does nothing.
12. **Beq disp**: if **PSW** == 0, causes the new value of PC to be the sum **PC** + **disp**. Note that if **disp** is negative, this will cause the program to jump backward in the sequence of instructions. If **PSW** != 0, this instruction does nothing.
13. **Bgt disp**: if **PSW**, is positive, causes the new value of **PC** to be the sum **PC** + **disp**. Note that if **disp** is negative, this will cause the program to jump backward in the sequence of instructions. If **PSW** <= 0, this instruction does nothing.

14. **Jmp disp**: causes the new value of **PC** to be the sum **PC** + **disp**.
15. **Hlt**: causes the svm machine to print the contents of registers **PC**, **PSW**, **RA**, **R0**, **R1**, **R2** and **R3**. It then stops.

Segments

Many modern computer architectures separate the program and data into different pieces called *segments*. In particular, the parts of program containing (static) data, are called data segments while the parts of program containing instructions are called (oddly enough) text segments.

Example: Remainder

Let the data segment data = [M; N], where M and N are natural numbers. The following SVM program (in a text segment) computes M mod N storing the remainder in register R0. The line numbers on the left are for readability.

```
data = [M, N]

1:     LOD   R0, 0(Zero)    #  R0 <= data[0] aka M
2:     LOD   R1, 1(Zero)    #  R1 <= data[1] aka N
3:     CMP   R0, R1         #  R0 < R1?
4:     BLT   2
5:     SUB   R0, R0, R1     #  R0 <= R0 - R1
6:     JMP   (-4)
7:     HLT                  #  The answer is in R0
```

Example: Factorial

Let the data segment data = [N], where N is a natural number. The following SVM program computes N! storing the result in register R0.

```
data = [N]
```

```
1:     Li    R0, 1          #  R0 <= 1
2:     MOV   R2, R0         #  R2 <= 1
3:     LOD   R1, 0(Zero)    #  R1 <= data[0] = N
4:     CMP   R1, Zero       #  R1 = 0?
5:     BEQ   3
6:     MUL   R0, R0, R1     #  R0 <= R0 * R1
7:     SUB   R1, R1, R2     #  R1 <= R1 - 1
8:     JMP   (-5)
9:     HLT                  #  N! is in R0
```

Example: Count Data

Data items in the data segment terminated by the sentinal -1.

```
data = [30, 40, 50, 60, 70, -1]
```

```
Mov R0, Zero             # start of counting routine
Mov R1, R0               # R1 is the address of a number
Li  R2, 1                # R2 is for incrementing
Lod R3, 0(R1)            # R3 is the number in the data
Cmp R3, Zero
Blt 3
Add R1, R1, R2           # point to the next datum
Add R0, R0, R2           # increment the data counter
Jmp -6                   # and go back to process it
R
```

Example: Sum Data

Data items in the data segment terminated by the sentinal -1.

```
data = [30, 40, 50, 60, 70, -1]
```

```
Mov R1, Zero             # R1 contains the address of data segment
Mov R0, R1               # R0 now used to hold sum
Li  R2, 1                # R2 used for incrementing
Lod R3, 0(R1)            # R3 has the data, NB offset=1!
Cmp R3, Zero
Blt 3
Add R0, R0, R3           # R0 := R0 + data
Add R1, R1, R2           # increment the address
Jmp (-6)                 # go read another data value
Lod R1, 0(Zero)          # R1 := count
Div R0, R0, R1           # R0 := sum / count, i.e., average
R
```

Example: Average Data

Data items in the data segment terminated by the sentinal -1. In this example, we must count the data and sum the data. We first compute the count. Since we need all 4 registers to computer the sum, we'll store the count in location 0 of RAM. Then the data starts at postion 1.

```
data = [0, 30, 40, 50, 60, 70, -1]
```

```
0:    Jsr 14                  # R0 := count(the data items)
1:    Mov R1, Zero            # R1 contains the address of data *)
2:    Sto R0, 0(R1)           # save no of elements in data area *)
3:    Li  R1, 1               # R1 contains the address of data segment
4:    Mov R0, Zero            # R0 now used to hold sum
5:    Li  R2, 1               # R2 used for incrementing
6:    Lod R3, 0(R1)           # R3 := data
7:    Cmp R3, Zero
8:    Blt 3
9:    Add R0, R0, R3          # R0 := R0 + data
10:   Add R1, R1, R2          # increment the address
11:   Jmp (-6)                # go read another data value
12:   Lod R1, 0(Zero)         # R1 := count
13:   Div R0, R0, R1          # R0 := sum / count, i.e., average
14:   R

15:   Mov R0, Zero            # start of counting routine
16:   Mov R1, R0              # R1 is the address of a number
17:   Li  R2, 1               # R2 is for incrementing
18:   Lod R3, 1(R1)           # R3 is the number in the data
19:   Cmp R3, Zero
20:   Blt 3
21:   Add R1, R1, R2          # point to the next datum
22:   Add R0, R0, R2          # increment the data counter
23:   Jmp -6                  # and go back to process it
24:   R
```

Data "Structures"

When data is stored in RAM (either the static data or the dynamic heap), it can be accessed via `Lod` and `Sto` instructions. For example, a register `R3` and a fragment of RAM might contain

```
                      | ... |
   +-------+          +-----+
R3 | 2000 |     2000 |  8  |
   +-------+          +-----+
                2002 |  6  |
                      +-----+
                      | ... |
```

Since `R3` contains the address `2000`, the instruction `Lod R0, 0(R3)` will load `8` into register `R0`.

> Heads up! We usually don't care about the particulars of where things are stored in RAM so we usually use arrows to depict references to RAM.

```
                  | ... |
+-----+           +-----+
R3 |  o--+--------> |  8  |
+-----+           +-----+
                  |  6  |
                  +-----+
                  | ... |
```

> The arrow depicts an address in RAM, naturally enough, these are sometimes called *pointers*.

Block Allocation

In the examples above, we had a sequence or *block* of numbers terminated with a sentinel value `-1`. Instead of using a sentinel value, blocks of data are often preceded by an integer count.

```
data = [5, 30, 40, 50, 60, 70]
```

Sequential Access

Exercise: Write an SVM program to index through the block, placing each value 30, ..., 70 in R0.

```
Li    R3, 2          # The word size AND, by chance, the base address of the block (!)
Lod   R2, 0(Zero)    # R2 := length of block = 5
Li    R1, 0          # the block index i := 0
Cmp   R1, R2         # done?
Beq   6
Mul   R1, R1, R3     # R1 := displacement to ith element
Add   R0, R1, R3     # R0 := the address of element block[i]
Lod   R0, 0(R0)      # MISSION ACCOMPLISHED R0 := block[i]
Li    R0, 1
Add   R1, R1, R0     # i := i + 1;
Jmp   -8
R
```

Randoml Access

Because addresses are natural numbers, we can compute addresses using the arithmetic instructions `Add`, `Sub`, `Mul` and `Div.` Lets call the block in the above example `a` and assume that we want to load the *i*th element of `a` into `R0`. This is usually written as `a[i]`, in OCaml it is written `a.(i)`. We'll assume the word index `i` is contained in `R3`.

```
  Mov R1, Zero
  Add R1, R1, R3     # R1 has the address of the word before a[i]
  Lod R0, 1(R1)      # R0 := a[i], NB: the 1 in 1(R1) skips over the size word
```

The `Add` instruction provides access to the *ith* element in one instruction cycle!

Linked Allocation

Data isn't always stored in contiguous memory words. It's common to have the data spread around in remote locations in RAM but linked together in a chain of cons cells

```
                 +-------+------+
                 | value | next |
                 +-------+------+
```

as in

```

address 0   2  4  6  8 10  12  14 16 18  20  22 24 26 28  30 32  34  36
data = [0, 70, 0, 0, 0, 0, 40, 34, 0, 0, 30, 12, 0, 0, 0, 60, 2, 50, 30]

```

where the first value `30` is at memory address 20, the second value `40` is at memory address 12, and so on. We would draw this as a linked list as

```
     +----+---+    +----+---+    +----+---+    +----+---+    +----+---+
 o--->| 30 | o-+--> | 40 | o-+--> | 50 | o-+--> | 60 | o-+--> | 70 | o-+--+
     +----+---+    +----+---+    +----+---+    +----+---+    +----+---    |
                                                                          =
```

even though these two-word cons blocks are scattered all about.

Code for the equivalent problem above of loading each value: 30, ..., 70 into R0 would be

```
Li   R1, 20      # the address of the first number 30
Cmp  R1, Zero    # quit if finished
Beq  3
Lod  R0, 0(R1)   # MISSION ACCOMPLISHED R0 := next number
Lod  R1, 2(R1)   # R1 := R1.next with 2 being # of bytes
Jmp  -5
R
```

> Heads up! Though this latter code is roughly half the size of the code for the block allocated data, it is generally much slower, in part, because the cost of a `Lod` instruction is relatively high and there are two `Lod` instructions in the loop in the code for the linked structure while there is only one in the code for the block-allocated structure. A second, concern is the issue of *locality* which relates to a processor memory cache. This is a topic for a more advanced course.