Note: The Elements of Computing Systems

September 22, 2020

Preface

This is my study note for The Elements of Computing Systems (ISBN: 978-0262640688) by Noam Nisan and Shimon Schocken.

Chapter 1

Boolean Logic

Boolean Algebra

$$
\begin{aligned}
x \text{ or } y &= \bar{x}y + x\bar{y} + xy \\
&= \bar{x}y + x(y + \bar{y}) \\
&= x + \bar{x}y \\
&= x + \overline{x + \bar{y}} \\
&= x + \overline{\overline{\overline{x + y}}} \\
&= \overline{\bar{x}(x + \bar{y})} \qquad \text{where } \overline{x + y} = \bar{x}\bar{y} \\
&= \overline{x\bar{x} + \bar{x}\bar{y}} \\
&= \overline{\bar{x}\bar{y}} \\
&= \overline{\overline{x + y}} \\
&= x + y \\
\text{if } y \text{ then } x &= \bar{x}\bar{y} + x\bar{y} + xy \\
&= \bar{x}\bar{y} + x(y + \bar{y}) \\
&= x + \bar{x}\bar{y} \\
&= x + \overline{x + y} \\
&= x + \overline{\overline{\overline{x + y}}} \\
&= \overline{\bar{x}(x + y)} \\
&= \overline{x\bar{x} + \bar{x}y} \\
&= \overline{\bar{x}y} \\
&= x + \bar{y} \\
\text{if } x \text{ then } y &= \bar{x}\bar{y} + \bar{x}y + xy \\
&= \bar{x}\bar{y} + (x + \bar{x})y \\
&= \bar{x}\bar{y} + y \\
&= \overline{x + y} + y \\
&= \overline{\overline{\overline{x + y}} + y} \\
&= \overline{(x + y)\bar{y}} \\
&= \overline{x\bar{y} + y\bar{y}} \\
&= \overline{x\bar{y}} \\
&= \bar{x} + y \\
x \text{ nand } y &= \bar{x}\bar{y} + \bar{x}y + x\bar{y} \\
&= \bar{y}(x + \bar{x}) + \bar{x}y \\
&= \bar{y} + \bar{x}y \\
&= \bar{x} + \bar{y} \qquad \text{where } x + y = x + \bar{x}y \\
&= \overline{\overline{\bar{x} + \bar{y}}} \\
&= \overline{xy}
\end{aligned}
$$

Chapter 5

Computer Architecture

5.1 Memory

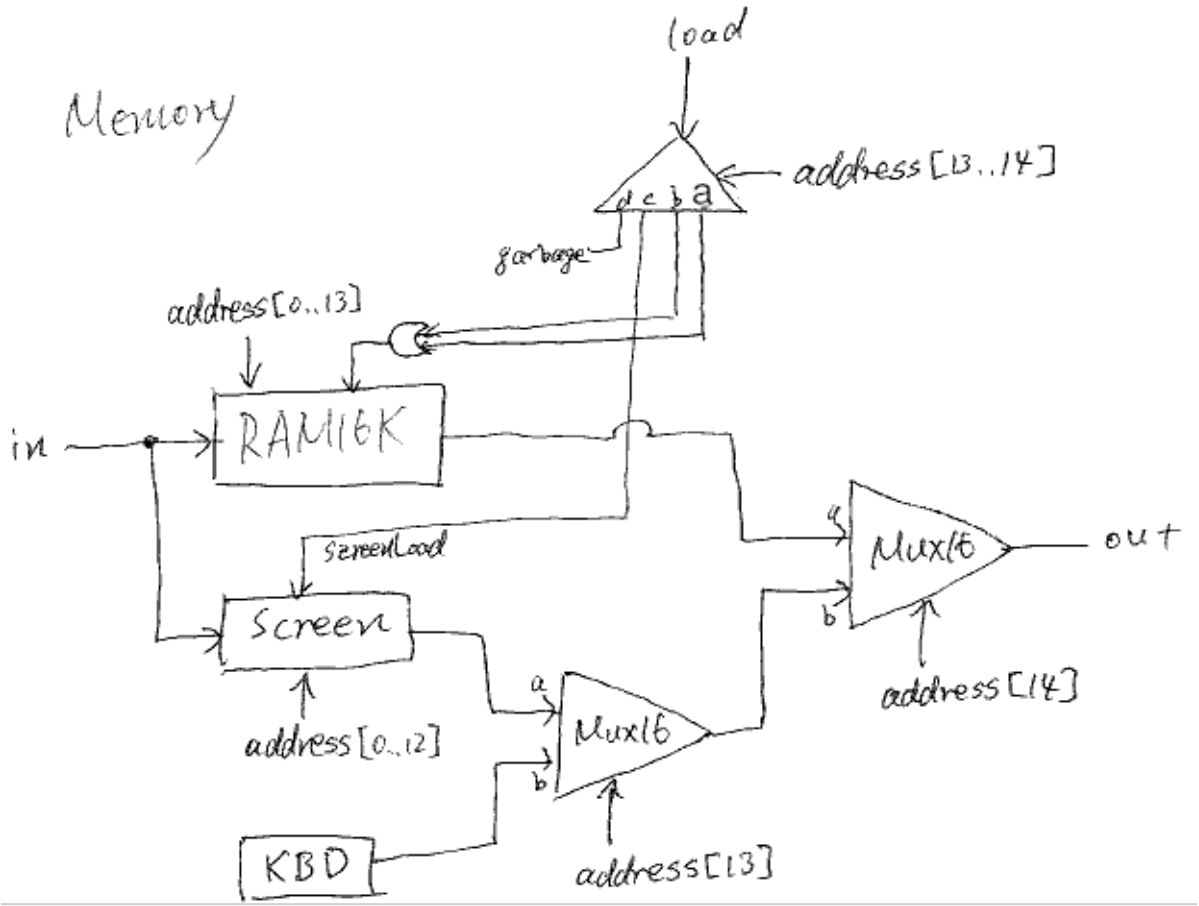


Figure 5.1: memory

5.2 CPU

C-instruction = $111ac_1c_2c_3c_4c_5c_6d_1d_2d_3j_1j_2j_3$

- d_1: destination A
- d_2: destination D
- d_3: destination M

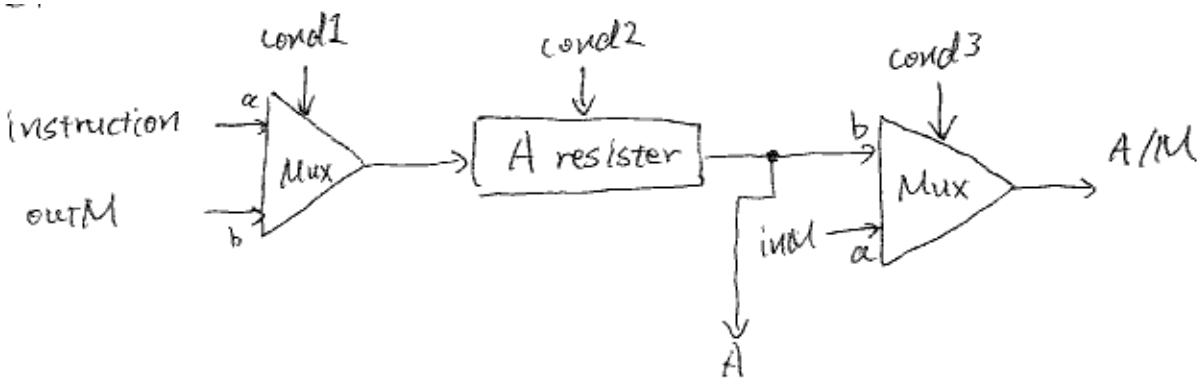


Figure 5.2: memory

$$\begin{aligned}&\text{cond1} = \text{instruction[15]}\\&\text{cond2} = (\text{not instruction[15]}) \text{ or } (\text{instruction[15] and instruction[5]})\\&\text{cond3} = \text{instruction[15] and not instruction[12]}\end{aligned}$$

- instruction[15]: opcode
- instruction[12]: C-instruction's a. If a is 1, comp includes A, otherwise, comp includes M.
- instruction[5]: destination A.

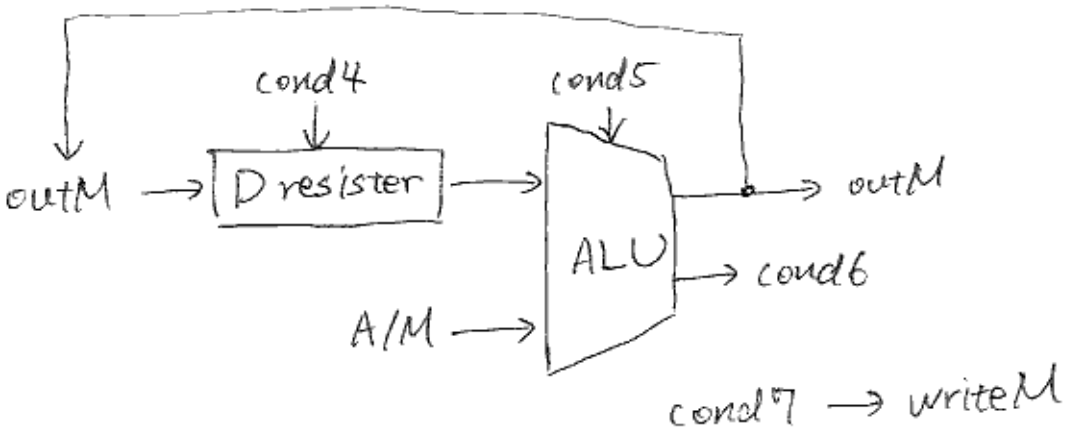


Figure 5.3: memory

$$\begin{aligned}&\text{cond4} = \text{instruction[15] and instruction[4]}\\&\text{cond5} = \begin{cases}\text{zx} = \text{instruction[11]} = c_1\\ \text{nx} = \text{instruction[10]} = c_2\\ \text{zy} = \text{instruction[9]} = c_3\\ \text{ny} = \text{instruction[8]} = c_4\\ \text{f} = \text{instruction[7]} = c_5\\ \text{no} = \text{instruction[6]} = c_6\end{cases}\\&\text{cond6} = (\text{zr}, \text{ng})\\&\text{cond7} = \text{instruction[15] and instruction[3]}\end{aligned}$$

- instruction[3]: d_3, destination M

- instruction[4]: d_2, destination D

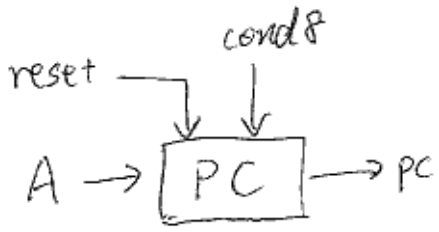


Figure 5.4: memory

$$
\begin{aligned}
\text{cond8} = & \overline{\text{zr}} \cdot \overline{\text{ng}} \cdot \overline{j_1} \cdot \overline{j_2} \cdot \overline{j_3} && \text{(JGT)} \\
& + \overline{\text{zr}} \cdot \overline{j_1} \cdot j_2 \cdot j_3 && \text{(JEQ)} \\
& + \overline{\text{ng}} \cdot \overline{j_1} \cdot j_2 \cdot j_3 && \text{(JGE)} \\
& + \text{ng} \cdot j_1 \cdot \overline{j_2} \cdot \overline{j_3} && \text{(JLT)} \\
& + \overline{\text{zr}} \cdot j_1 \cdot \overline{j_2} \cdot j_3 && \text{(JNE)} \\
& + (\text{zr} + \text{ng}) \cdot j_1 \cdot j_2 \cdot \overline{j_3} && \text{(JLE)} \\
& + j_1 \cdot j_2 \cdot j_3 && \text{(JMP)}
\end{aligned}
$$

Chapter 7

Virtual Machine I: Stack Arithmetic

7.1 Arithmetic

add (sub) ⇒

```
// SP--
@SP
M=M-1

// D = y
A=M
D=M

// SP--
@SP
M=M-1

// *SP = y + x (add) / x - y (sub)
A=M
M=D+M (add) / M=M-D (sub)

// SP++
@SP
M=M+1
```

neg, not ⇒

```
// SP--
@SP
@M=M-1

// -M
A=M
M=-M (neg) / M=!M (not)

// SP++
@SP
M=M+1
```

eq, gt, lt ⇒

```
// SP--
@SP
M=M-1

// D = y
A=M
D=M

// SP--
@SP
M=M-1

// x - y
A = M
D=M-D

// if condition then -1 else 0 end
@then
D;jEQ (eq), D;JGT (gt), D:JLT (lt)
@SP
A=M
M=0
@end
0;JMP
(then)
@SP
A=M
M=-1
(end)

// SP++
@SP
M=M+1
```

```
and, or ⇒ {
    // SP--
    @SP
    M=M-1

    // D = y
    A=M
    D=M

    // SP--
    @SP
    M=M-1

    // *SP = y&x
    A=M
    M=D&M (and), M=D|M (or)

    // SP++
    @SP
    M=M+1
```

7.2 logical command

```
push constant i ⇒ { *SP=i
                    SP++      ⇒ {
                                  // *SP=i
                                  @i
                                  D=A
                                  @SP
                                  A=M
                                  M=D

                                  // SP++
                                  @SP
                                  M=M+1
```

```
push segment i ⇒ { addr = SEG + i
                   *SP = *addr
                   SP++          ⇒ {
                                     // addr = SEG + i
                                     @SEG
                                     D=M
                                     @i
                                     A=D+A

                                     // *addr
                                     D=M

                                     // *SP = *addr
                                     @SP
                                     A=M
                                     M=D

                                     // SP++
                                     @SP
                                     M=M+1
```

where `segment = local, argument, this, that`.

```
push pointer i ⇒ { *SP = *(R3 + i)   ⇒ {  // *(R3 + i)
                                           @R(3 + i)
                                           D=M

                                           // *SP = *(R3 + i)
                                           @SP
                                           A=M
                                           M=D
```

```
push temp i ⇒ { *SP = *(R5 + i)   ⇒ {  // *(R5 + i)
                                        @R(5 + i)
                                        D=M

                                        // *SP = *(R5 + i)
                                        @SP
                                        A=M
                                        M=D
```

```
push static i ⇒ { *SP = *Xxx.i   ⇒ {  @Xxx.i
                                      D=M
                                      @SP
                                      A=M
                                      M=D
                                      @SP
                                      M=M+1
```

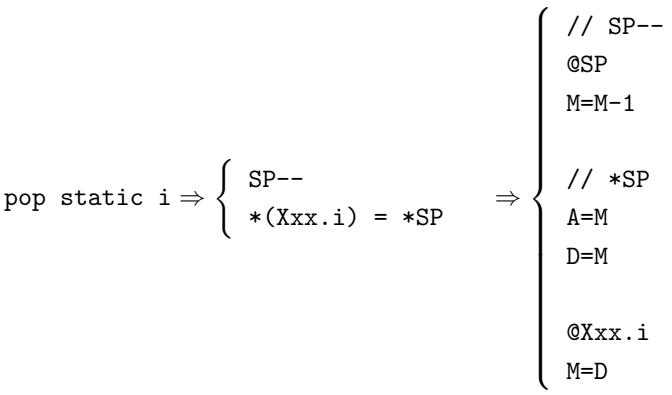
where this vm program name is `Xxx.vm`.

$$\texttt{pop segment i} \Rightarrow \begin{cases} \texttt{addr = SEG + i} \\ \texttt{SP--} \\ \texttt{*addr = *SP} \end{cases} \Rightarrow \begin{cases} \texttt{SEG = SEG + i} \\ \texttt{SP--} \\ \texttt{*SEG = *SP} \\ \texttt{SEG = SEG - i} \end{cases} \Rightarrow \begin{cases} \texttt{// SEG=SEG+i} \\ \texttt{@SEG} \\ \texttt{D=M} \\ \texttt{@i} \\ \texttt{D=D+A} \\ \texttt{@SEG} \\ \texttt{M=D} \\ \\ \texttt{// SP--} \\ \texttt{@SP} \\ \texttt{M=M-1} \\ \\ \texttt{// *SP} \\ \texttt{A=M} \\ \texttt{D=M} \\ \\ \texttt{// *SEG = *SP} \\ \texttt{@SEG} \\ \texttt{A=M} \\ \texttt{M=D} \\ \\ \texttt{// SEG=SEG-i} \\ \texttt{@i} \\ \texttt{D=A} \\ \texttt{@SEG} \\ \texttt{M=M-D} \end{cases}$$

where `segment = local, argument, this, that`.

$$\texttt{pop pointer i} \Rightarrow \begin{cases} \texttt{SP--} \\ \texttt{*SP} \\ \texttt{*(R3 + i) = *SP} \end{cases} \Rightarrow \begin{cases} \texttt{// SP--} \\ \texttt{@SP} \\ \texttt{M=M-1} \\ \\ \texttt{// *SP} \\ \texttt{A=M} \\ \texttt{D=M} \\ \\ \texttt{@R(3 + i)} \\ \texttt{M=D} \end{cases}$$

$$\texttt{pop temp i} \Rightarrow \begin{cases} \texttt{SP--} \\ \texttt{*SP} \\ \texttt{*(R5 + i) = *SP} \end{cases} \Rightarrow \begin{cases} \texttt{// SP--} \\ \texttt{@SP} \\ \texttt{M=M-1} \\ \\ \texttt{// *SP} \\ \texttt{A=M} \\ \texttt{D=M} \\ \\ \texttt{@R(5 + i)} \\ \texttt{M=D} \end{cases}$$


pop static i ⇒
SP--
*(Xxx.i) = *SP
⇒
// SP--
@SP
M=M-1
// *SP
A=M
D=M
@Xxx.i
M=D

Chapter 11

Compiler II: Code Generation

`label` *symbol*
`goto` *symbol*
`if-goto` *symbol*
`function` *functionName nLocals*
`call` *functionName nArgs*

Expression

Integer constant

```
cgen(n) = push constant n
```

where `n` is an integer.

String constant

```
cgen(str) = push constant length(str)
            call String.new 1
            push constant ASCII(str1)
            call String.appendChar 2
            push constant ASCII(str2)
            call String.appendChar 2
              ⋮
            push constant ASCII(strn)
            call String.appendChar 2
```

where `str` is a string constant, `stri` is ith character of str, `ASCII(x)` is ASCII number of char `x` and `n` is the length of `str`.

Keyword constant

```
cgen(true) = push constant 0
             neg

cgen(false) = push constant 0

cgen(null) = push constant 0

cgen(this) = push pointer 0
```

Variable

```
cgen(x) = push segment i
```

where `x` is a variable and `segment i` is related to `x`.

```
cgen( arr[expr] ) = cgen( expr )
                    push segment i
                    add
                    pop pointer 1
                    push that 0
```

where `arr` is an Array and `segment i` is related to `arr`.

Subroutine Call

```
cgen(subroutineName(expressionList)) = push pointer 0
                                       cgen(expressionList)
                                       call ClassName.subroutineName z
```

where `ClassName` is the name of class that *subroutineName* belong to and `z` $= nArgs + 1$.

```
cgen(obj.subroutineName(expressionList)) = cgen(obj)
                                           cgen(expressionList)
                                           call ClassName.subroutineName z
```

where `ClassName` is the class name of *obj* and `z` $= nArgs + 1$.

```
cgen(className.subroutineName(expressionList)) = cgen(expressionList)
                                                 call ClassName.subroutineName nArgs
```

```
cgen(expressionList) = cgen(expr1, expr2, ..., exprn)
                     = cgen(expr1)
                       ⋮
                       cgen(exprn)
```

Parenthesis

```
cgen{ ( expr ) } = cgen( expr )
```

Unary Operator

```
cgen( op term ) = cgen(term)
                  op
```

Operator

```
cgen( term1 op term2 ) = cgen(term1)
                         cgen(term2)
                         op
```

Statement

let

```
cgen(let varName = expr ; )  = cgen(expr)
                               pop segment i
```

where `segment i` is related to `varName`.

```
cgen( let arr[expr1] = expr2 ; ) = cgen(arr)
                                   cgen(expr1)
                                   add
                                   cgen(expr2)
                                   pop temp 0
                                   pop pointer 1
                                   push temp 0
                                   pop that 0
```

if

```
cgen( if ( expr ) { statements } ) = cgen( expr )
                                     not
                                     if-goto label
                                     cgen( statements )
                                     label label
```

```
cgen( if ( expr ) { statements1 } else { statements2 } )  = cgen( expr )
                                                            not
                                                            if-goto else
                                                            cgen( statements1 )
                                                            goto end
                                                            label else
                                                            cgen( statements2 )
                                                            label end
```

`if-goto` jumps when top most stack value is not 0.

while

```
cgen( while (expr) { statements } ) = label begin
                                      cgen( expr )
                                      not
                                      if-goto end
                                      cgen( statements )
                                      goto begin
                                      label end
```

do

```
cgen( do subroutineCall ; ) = cgen( subroutineCall )
                              pop temp 0
```

return

$$\texttt{cgen(return ;)} = \begin{array}{l}\texttt{push constant 0}\\ \texttt{return}\end{array}$$

$$\texttt{cgen(return expr;)} = \begin{array}{l}\texttt{cgen(expr)}\\ \texttt{return}\end{array}$$

Subroutine Declaration

$$\texttt{cgen}\left(\begin{array}{l}\texttt{constructor className subroutineName(parameters) \{}\\ \quad\texttt{varDec*}\\ \quad\texttt{statements}\\ \texttt{\}}\end{array}\right)$$

$$= \begin{array}{l}\texttt{function className.subroutineName nLocals}\\ \texttt{push constant nFields}\\ \texttt{call Memory.alloc 1}\\ \texttt{pop pointer 0}\\ \texttt{cgen(statements)}\end{array}$$

where `className` is the class that `subroutineName` belong to and `nFields` is the number of field variable in the class.

$$\texttt{cgen}\left(\begin{array}{l}\texttt{function type subroutineName(parameters) \{}\\ \quad\texttt{varDec*}\\ \quad\texttt{statements}\\ \texttt{\}}\end{array}\right)$$

$$= \begin{array}{l}\texttt{function className.subroutineName nLocals}\\ \texttt{cgen(statements)}\end{array}$$

where `className` is the class that `subroutineName` belong to.

$$\texttt{cgen}\left(\begin{array}{l}\texttt{method type subroutineName(parameters) \{}\\ \quad\texttt{varDec*}\\ \quad\texttt{statements}\\ \texttt{\}}\end{array}\right)$$

$$= \begin{array}{l}\texttt{function className.subroutineName nLocals}\\ \texttt{push argument 0}\\ \texttt{pop pointer 0}\\ \texttt{cgen(statements)}\end{array}$$

where `className` is the class that `subroutineName` belong to.