NAND2Tet - Exam Preparation

February 2011

The following is a short document containing details I found important when preparing for the exam. It may contain errors! Use at your own responsibility.

Written by Nerya Or.

1 Assembly

Bit indices

Given a 16-bit integer n, it is ordered as follows:

$n[15]$	$n[14]$	$n[13]$	$n[12]$	$n[11]$	$n[10]$	$n[9]$	$n[8]$	$n[7]$	$n[6]$	$n[5]$	$n[4]$	$n[3]$	$n[2]$	$n[1]$	$n[0]$

- Note that even though this is the situation, when working with the screen things are "reversed"!! When one of the screen registers contains the decimal "1" value (0b0000000000000001), it will light up the **left**-most pixel in that register, rather than the right as we would expect based on the index scheme above.

A-instruction: 0XXXXXXXXXXXXXXX

This type of instruction is equivalent to doing "$@XXXXXXXXXXXXXXX$" (int constant). Sets the A register to contain the first 15 digits.

C-instruction: 1XXXXXXXXXXXXXXX

Also marked: $111\,a\,c_1c_2c_3c_4\,c_5c_6d_1d_2\,d_3j_1j_2j_3$.

- j_i bits are for jumps.
  - j_1 is 1 iff we jump when $out < 0$.
  - j_2 is 1 iff we jump when $out = 0$.
  - j_3 is 1 iff we jump when $out > 0$.
- d_i bits are for the destination of the result.
  - if d_1 is 1, then the result will go to A.
  - if d_2 is 1, then the result will go to D.
  - if d_3 is 1, then the result will go to M.
- c_i bits determine arithmetic operations in the ALU.
  - c_1 is wired to zx in the ALU. If c_1 is 1, x is set to 0.
  - c_2 is wired to nx in the ALU. If c_2 is 1, x is set to $\neg x$ (bitwise).
  - c_3 is wired to zy in the ALU. If c_3 is 1, y is set to 0.
  - c_4 is wired to ny in the ALU. If c_4 is 1, y is set to $\neg y$ (bitwise).
  - c_5 is wired to f in the ALU. If c_5 is 1, the ALU will compute "$op1 + op2$", and if c_5 is 0, it will compute "$op1 \wedge op2$" (bitwise AND).
  - c_6 is wired to no in the ALU. If c_6 is 1, the ALU will set $out = \neg out$.

Code for push, pop operations

- push constant 17:

  @17

  D=A

  @SP

  AM=M+1

  A=A-1

  M=D

- pop:

  @SP

  AM = M-1

  D = M

Addresses

- Predefined in the assembly language:
  - $R0 - R15$ = 0...15, respectively.
  - Within those addresses, some have a special name:
    - SP=0
    - LCL=1
    - ARG=2
    - $THIS$=3
    - $THAT$=4
    - Not related to assembly, but still worth noting:
      - $R5 - R12$ are the *temp* segment in the VM.
      - $R13 - R15$ are general purpose registers to be used by the VM.
  - $SCREEN$=16384 (0x4000)
  - KBD=24576 (0x6000)

2 VM

2.0.1 Segments mapping

- *argument* maps directly to $*ARG$
- *local* maps directly to $*LCL$
- *static* is defined per file, using a naming convention. "*static* 5" maps to $@currentFilename.5$
- *constant* is mapped directly in compile-time to an int constant. Only for non-negative constants: 0...32767.
- *this* is mapped directly to $*THIS$
- *that* is mapped directly to $*THAT$
- *pointer* is mapped directly to $THIS$. Thus, *pointer* 0 is used to set the address in $THIS$ and *pointer* 1 is used to set the address in $THAT$.
- *temp* is mapped to $R5$. By convention, *temp* is restricted to $R5 - R12$, so *temp* 8 is illegal by convention (as it accesses $R13$).

To sum up the addresses:

- 0-15 are the registers described above.
- 16-255 are used as the *static* segment
  - Why? Because we simply map static variables to named assembly variables, and the assembler assigns these variables addresses starting from 16 and counting up.
- 256 - 2047 is the stack.
  - In the VMTranslator we write down code that initializes SP to point at 256. These are the first lines of codes that run, followed by a jump to $Sys.init$.
- 2048 - 16383 is the heap.
  - Determined in the OS.
- 16384 - 24575 is the screen memory.
- 24576 is the keyboard memory.

Order of operations in a function call

- First, the caller pushes n arguments, and then "call"s the function, with n marking the number of arguments he's pushed.
- Now the call procedure starts:
  - Caller pushes the "return address" label.
  - Now we do what may be abbreviated as $LATT$:
    * Caller pushes his LCL address
    * Caller pushes his ARG address
    * Caller pushes his $THIS$ address
    * Caller pushes his $THAT$ address
  - Set $ARG = SP - n - 5$. This makes the ARG pointer for the called function point right at the arguments that the calling function has pushed before starting the call procedure.
  - Set $LCL = SP$. This makes the LCL pointer for the called function point at the current "free" stack space, right after everything that we've pushed just now. Since the called function will never peek beyond the base LCL pointer, this means all the stack history is kept invisible to the called function.
  - Jump to the function. Simply $@functionName$ followed by $0; JMP$.
  - The next instruction we write is in fact a label: the "return address" label. When the called function returns, it will jump here by popping the value we pushed earlier.

Beginning of a function

Whenever a function with n arguments starts running (after it's been called), it **pushes 0** n times. This means the LCL segment will be initialzed with n null-ed variables.

This also means, that the SP will be pointing **after** LCL, as it should. Had we not performed this pushing step, our programs could have accidentally run over some of the local variables in them, while writing things to the stack.

Order of operations when returning from a function

- First, the returning function always pushes some return value. Void functions push 0 by convention.
- Now the return procedure starts:
  - We will work with a temporary variable, nicknamed $FRAME$. In implementation it's probably $R15$ or something.
    - Start with $FRAME = LCL$.
  - Another temporary variable, which we will call RET, will be initialized with $RET = *(FRAME - 5)$.
    - RET contains the return address label that was pushed during the call procedure.
  - We now do: $*ARG = pop()$, meaning we pop the top value of the stack into $ARG[0]$. Since the convention says that when a function returns all of its arguments are popped from the stack, we see that $ARG[0]$ is the position of the stack that's right after where the caller function's stack was pointing to before the call was made. Thus, putting the return value there means the return value from the called function will indeed be the first thing the calling function sees on the stack eventually. All we need is to handle the SP pointer...
  - $SP = ARG + 1$. This is to accomodate what we've just done a moment earlier...
  - Now, we start restoring the 4 registers:
    - $THAT = *(FRAME - 1)$
    - $THIS = *(FRAME - 2)$
    - $ARG = *(FRAME - 3)$
    - $LCL = *(FRAME - 4)$
  - And to finish things off, we will jump to the address we stored in RET.
    - Why do we even need this RET variable? Well, in the case where we have $n = 0$ arguments to the called function, when we write the return value into $ARG[0]$ we would have in fact over-written the return address! Therefore we store it into a temporary variable before we do anything else.

Other implementation details

- The stack pointer SP always points one slot "after" the location of the last valid stack item.
  - When pushing, we **enlarge** the stack pointer. Naive implementation: $@SP$ followed by $M = M + 1$.
    - To push: Set $*SP$ to the new value, and then raise SP by 1.
  - When popping, we **reduce** the stack. Naive implementation: $@SP$ followed by $M = M - 1$.
    - To pop: Reduce SP by 1, and then take $*SP$.
- VM commands:
  - Arithmetic: *add*, *sub*, *neg*, *eq*, *gt*, *lt*, *and*, *or*, *not*
    - *neg* here is **arithmetic negation** (2's complement), rather than bitwise "not". We have a *not* command for that...
  - Program flow:
    - *label* symbol
    - *goto* symbol
    - $if-goto$ symbol
  - Function calling:
    - $function$ functionName nLocals
    - *call* functionName nArgs
      - Note that when declaring a function we declare the number of locals, in contrast to when calling it, where we declare the number of arguments.

      * *return*

- When compiling Jack files, the compiler gives all functions a prefix according to the name of the file (class) they reside in. Therefore, we can assume all function names in different VM files are unique.
  - Assuming we have a function foo in the file Filename.jack, representing the class $Filename$, **the jack compiler** will translate that function to a VM function called $Filename.foo$.
  - Inside Jack functions, the Jack compiler takes care to use unique labels (using a counter).
- When writing an **assembly** label in the process of compiling a VM file to assembly, we give that label a prefix of the current function we're in, in the VM file.
- To conclude: Our final assembly file will have labels of the form $Filename.foo$ to signify start of functions, as well as labels of the form $Filename.foo\$labelName$ for internal function labels (and $labelName$ is unique for every foo-scope).
- Access to the *static* segment in a file $Filename.vm$ at index j, ("$push\,static\,0$") will be translated as access to the assembly variable $Filename.j$.

3 Jack

Constructor implementation

- When writing a constructor's VM code, we:
  1. Push the number of fields (using "count" functionality of the symbol table, for FIELD-kind variables).
  2. Call $Memory.alloc$ with the one argument we've just pushed.
  3. Pop the resulting address into *pointer* 0 ("this"'s address).
- Since the constructor's Jack implementation has to return "this", we will end up returning the address properly.
  - Note that "this" in Jack is in fact parsed as *pointer* 0, i.e. with no dereferencing involved.

Calling an object's method

- To call a method that belongs to a specific object ("$foo.bar()$"), we need to somehow pass the object as an argument. Therefore, by convention, all methods get a first "hidden" argument that is the address of the object on which the method operates. So $foo.bar(int\,blah)$ is in fact implemented as $foo.bar(fooObject * objPointer, int\,blah)$.
- To do this, the Jack compiler takes the following steps:
  - When calling a method, the caller must push the object's address as the 0'th argument:
    * $push\ < object\,variable's\,segment >< object\,variable's\,index >$
    * push any other arguments
    * $call\ < classOfObject > . < methodName >$
  - The actual method code compiled will always start with:
    * $push\,argument\,0$
    * $pop\,pointer\,0$
      - This sets "this" as the object we're operating on.

Handling a subroutine call

We have several options to take into consideration when compiling a call to a subroutine (for example "*do subroutine*").

- If we have $identifier1.subroutineName$:
  - If $identifier1$ is a variable name, then we look up the matching variable's class from the symbol table, and then -
    - Push the variable's address (as arg0, for "this" later), i.e. *push variableSegment variableIndex*
    - Push arguments, if they exist
    - Call $classOfVariable.subroutineName$ with the number of arguments we pushed, including the "this" address.
  - If $identifier1$ is not a variable name, then it must be a class name. In this case -
    - Push arguments, if they exist
    - Call $identifier1.subroutineName$ with the number of arguments we pushed.
- Otherwise, we just have $identifier1$ (no "$.subroutineName$"), and therefore we will expect that $identifier1$ is a **method** (!!!) that is defined within the current class.
  - Push the variable's address (as arg0, for "this" later), i.e. *push pointer* 0
  - Push arguments, if they exist
  - Call $currentCompiledClassName.identifier1$ with the number of arguments we pushed, including the "this" address.