Introduction to Hack Assembly Language

Jump to

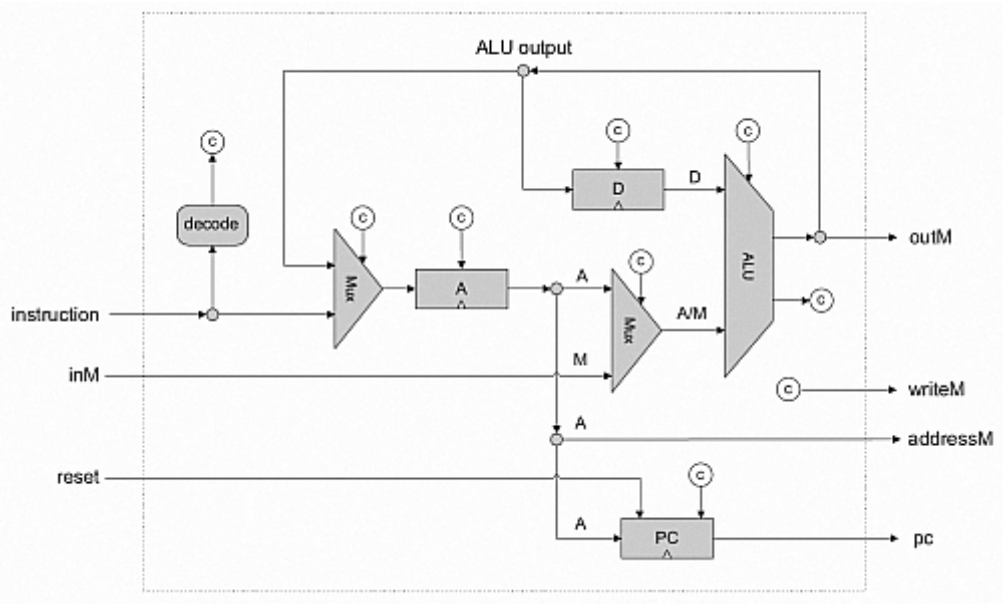

Figure 5.9: Proposed CPU Implementation

The Hack CPU has two registers, A and D. The Hack Assembly Language adds a pseudo-register, M, that refers to transferring data between the CPU and RAM. The A, D, and M register values can be sent to the ALU. The output of the ALU can be stored into the A, D, and M registers. The A register value is used as the RAM address when reading or writing the M register. The A register value is also used to set the program counter (PC) when a program jump occurs.

The Hack CPU has two types of instructions, A-instructions and C-instructions. The Hack Assembly Language adds a Label pseudo-instruction.

A-Instructions

A-instructions load the A register with a constant value between 0 and 32767, inclusive. A-instructions never read from RAM. The Assembly Language syntax for A-instructions is:

"@"constant	Load a non-negative decimal constant into the A-register.
"@"symbol	Load the address of the code label or RAM variable symbol into the A-register.

There are several built-in RAM variables available. For chapter 4 you may want to use the variables SCREEN and KBD whose address are 13684 and 24576, respectively. See 6.2.3 for the complete list of built-in variables.

Warning: A, D and M are valid symbol names and can be used with A-instructions, as in "@D". This will load the A register with the address of a variable named "D" or a code label "(D)"; it has nothing to do with the D register. Using A, D or M as symbols is highly discouraged.

> Software Bug: As of January 2014, there is a bug in the Assembler tool and the assembler built into the Hardware Simulator and CPU Emulator. The @ instruction accepts numbers outside the range $0 \leqslant n \leqslant 32767$, including negative numbers,

without issuing any error message. The machine code generated by these erroneous instructions results is undefined behavior in the Hardware Simulator and CPU Emulator.

The most common coding error that hits this bug is "@-1". The correct way to do this is "A=-1".

The most common manifestation of this bug in the Hardware Simulator and CPU Emulator is that there is no instruction shown for a ROM address when the ROM is showing the ASM view. For most of the erroneous instructions, the Dec/Hex/Bin views will show the erroneous machine code.

C-Instructions

C-instructions do computations, data moves, and conditional jumps. The Assembly Language syntax for C-instructions is:

[destination"="] computation [";" jump-condition]	Note that destination and jump-condition are optional.

Computation

All C-instructions do a computation. The computation may be as simple as returning a register's value unchanged, or returning a constant 0, 1, or -1. More complex computations available are the arithmetic operations add, subtract, increment, decrement and negate, and the bitwise logical operations and, or and invert. (Invert is sometimes referred to as logical negation.)

For computations involving two registers, the registers must be D and one of A or M. The registers may occur in either order. Register names must be upper case.

0	The constant value 0.	A+1	A register plus 1.
A	The A register.	M-D	Subtract D register from RAM[A].
-1	The constant value -1.	D+D	Illegal - Must use D and A or M.
-D	Arithmetic negation of D register.	A+M	Illegal - Must use D and A or M.

Destination

A C-instruction may optionally store the result of the computation into one or more of the A, D, or M registers. It is possible (and sometimes useful) to store into the A and M registers simultaneously. The assembler is picky. The letter order is important - the only values of destination registers it accepts are:

A	M	AD	ADM
D	MD	AM	

Putting the letters in the wrong order gives a "Destination expected" error message.

Jump condition

A C-instruction may optionally jump to the program address in the A register. The result of the computation is compared to 0 and the jump occurs if the specified jump

condition is satisfied. The jump conditions are:

JLT	Jump if result < 0.	JEQ	Jump if result = 0.	JGT	Jump if result > 0.
JLE	Jump if result ⩽ 0.	JNE	Jump if result ≠ 0.	JGE	Jump if result ⩾ 0.
JMP	Always jump.				

Warning: The behavior of the CPU is undefined if the destination includes the A register and a jump occurs. Depending on CPU implementation, the PC may be loaded with either the original or the updated A-register value.

Example C-instructions

D=A	Set D register to A register.
AD=A+1	Increment A register and set D register to incremented value.
M=~M	Invert RAM[A].
M=M-D	Subtract D register from RAM[A].
AMD=0	Set RAM[A] and A and D registers to zero.
0;JMP	Unconditional jump to ROM address in A register. Note that there must be a computation; the convention is to use 0 for unconditional jumps.
D;JNE	Jump to ROM address in A register if D register is non-zero.
D=D-1;JGT	Decrement D register, then jump to ROM address in A register if D register is greater than zero.
M=M-1;JGT	Not useful because the A register must be used for both the RAM address and the target jump address.

Labels

The Hack Assembly Language syntax for labels is:

"(" symbol ")"

Labels do not generate any instructions, nor do they take up any ROM space. They simply provide a target for jump instructions. The label's value is the address of the next A- or C-instruction.

Register usage

The "A" in A-register stands for address. The A-register is primarily used for three things:

- Constants used in computations and variable initializations,
- RAM addresses for variable accesses, and
- ROM addresses for jumps.

Note that the A-register may only be used for one of these things in any given C-instruction.

The "D" in D-register stands for data. The D-register is primarily used for two things:

- Data used in variable computations, and
- Computing memory addresses for array elements.

Here are some examples showing how the registers are used:

Pseudo-code	ASM	Pseudo-code	ASM	Pseudo-code	ASM
n = 1	@n M=1	n = 17	@17 D=A @n M=D	n = -2	@2 D=-A @n M=D
a = -a	@a M=-M	a = a+7	@7 D=A @a M=D+M	a = b+c	@b D=M @c D=D+M @a M=D

goto LOOP	@LOOP
	0;JMP
		if (x==0)
		  goto BREAK
			@x
			D=M
			@BREAK
			D;JEQ
				if (y<=5)
				  goto REPEAT
					@y
					D=M
					@5
					D=D-A
					@REPEAT
					D;JLE

a[123] = 0	@123 D=A @a A=D+A M=0	a[i] = 1	@i D=M @a A=D+A M=1	a[j] = b	This is tricky and you will not need it until chapter 7. I leave it for you to figure out then.

Writing a program

The sample program will place the numbers 0, 11, 22, ..., 99 int RAM locations starting at address 100, and then hang in an infinite loop. This program will illustrate several important aspects of Hack assembly language programming:

- Set a variable to a constant
- Perform arithmetic on a variable
- Access RAM through a pointer variable
- Conditional jump

Here's the pseudo-code for the program:

```
n = 0
addr = 100
do {
    RAM[addr] = n
    addr = addr+1
    n = n+11
} while n <= 99
```

Hack program

```
1:      @n      // n = 0
2:      M=0
3:
4:      @100    // addr = 100
5:      D=A
6:      @addr
7:      M=D
8:
9:  (Loop)      // do {
10:     @n      //     RAM[addr] = n
11:     D=M
12:     @addr
13:     A=M
14:     M=D
15:
```

```
16:     @addr   //      addr = addr+1
17:     M=M+1
18:
19:     @11     //      n = n+11
20:     D=A
21:     @n
22:     MD=M+D  // (tricky: also leaves new 'n' value in D)
23:
24:     @99     // } while n <= 99
25:     D=D-A
26:     @Loop
27:     D;JLE
28:
29:     @Halt   // loop forever
30: (Halt)
31:     0;JMP
```

Notes:

1: n will be assigned address 16.

6: addr will be assigned address 17.

13: Although it looks like this might be a problem because the instruction is changing the A register which is supplying the address of addr, this is OK because the A register is changed when the instruction executes which happens after addr's value is read from RAM.

22: Stores n+11 into n and into D where it will be used for the following comparison.

30: Halt is assigned the ROM address of the 0;JMP instruction. This is the preferred way to write a halt loop so that the 0;JMP jumps to itself.