A computer consists of a state: lots of DFF generally organized into a sequence of largely equivalent registers (of size 16, 32, 64 bits depending on the technology; whatever that size is we'll call that a word) that can be modified by an instruction (a binary number inputs, again 16, 32, 64 bits depending on the technology). By this definition a JKFF is just a very simple computer with one one-bit register Q that can be modified by the instruction JK. Normally a sequence of instructions (a program) is stored in RAM. A specialized register, called the program counter or PC, determines the address in RAM of the instruction to be executed. Like any counter, the PC will increment on a clock edge but may also be set to some value defined on its inputs. A non-incremental change in the PC will make a big change in the next instruction to be executed. This is called a branch or jump.

A computer is largely defined by the set of instructions the chip designer has implemented. HH 14.2 and Table 14.1 gives a brief and simplified tour of the instructions available in early Intel chips (x86). For this project I've defined a computer both more complex and more simple than the x86. It has 16 64-bit registers; since a single hex digit can name 16 things the registers are named with a single hex digit: r0–rF. r0 and rF are both specialized: rF is the PC and r0 is automatically connected to whatever word follows the current instruction in RAM, i.e., whatever word is at the address[1] in RAM of $PC + 1$. The instruction is organized to allow operations that involve up to three registers. One register may be changed by the operations: the destination[2] register whose CLK input is controlled by the D-decoder and hence the DST or D nibble (4 bits, a hex digit) of the instruction. (Every register's D inputs are connected to the D-bus but only one will be clocked.) Up to two registers may be used to calculate the value to be placed on the D-bus and hence eventually in the destination register. The binary number held by the two source registers will be placed on the A-bus or B-bus; the A-decoder (and hence the A nibble of the instruction) determines which register is connected to the A-bus; the B-decoder determines which register is connected to the B-bus. The bottom 3/4 (everything except the first 16 bits of the 64-bit instruction) of the instruction determines exactly how the value(s) on the source buses are used to determine the value placed on the destination bus; we call that part of the instruction the 'opcode' (operation code). The bits in the opcode would control exactly which functional units are connected to the D-bus and what those functional units should do. (I.e., we imagine an arithmetic unit that can add, subtract, multiple, divide, etc. according to some input controls that are connected to the opcode bits of the instruction.) The top quarter of the instruction is used to determine which (if any) of the registers are connected to the various buses. Leaving aside the most significant nibble, the next nibble (the A nibble) says which of the 16 registers are connected to the A-bus, the next nibble says which of the 16 registers are connected to the B-bus, the next nibble says which of the 16 registers will be updated using the value on the D-bus. That is these nibbles are directly connected to the respective decoder that selects which registers are connected to which bus (or in the case of the D-decoder, which register is connected to the system clock which will force its DFFs to read and store the value on their D inputs). The top nibble in the instruction, with its three least significant bits, simply determines whether the operation uses the A, B, or D buses. These bits will be connected to the enables of the corresponding decoder. The MSB in the instruction determines if the PC will be incremented by one or two; It is directly connected to the 2x input pin of the PC.

In denoted these instructions all four bits of the high nibble are displayed, followed by the hex digits for the A B D decoders. If a decoder is turned off by the appropriate bit of the first nibble,

[1] This is an unusual feature; generally RAM is organized into bytes not words so the following 64-bit instruction would be at $PC + 8$

[2] The D used to describe this stands for destination not the hex digit D. Similarly the bus names A and B are arbitrary and have nothing to do with the corresponding hex digits.

the corresponding value of the hex digit does not matter and it is displayed as **x**. We don't bother to display the bottom 3/4 of the instruction.

An important feature of operations is the resulting Condition Codes (CC): individual bits that report whether the most recent operation produced a zero result, a negative result, or (for logical operations) a 0. The various CC bits control if the PC is updated, i.e., if a branch will occur.

Note: since 'floating point' numbers (i.e., computer versions of the real numbers) are radically different from integer binary numbers, many instructions come in two versions: one that starts with **f** for floating point numbers and one that doesn't for integer numbers. (e.g., **fneg** negates a floating point number simply by flipping the sign bit (which is the MSB) whereas **neg** on an integer requires twos complement).

Homework

1. Find on the web site the file: `sqr.txt`. This file contains code that should calculate the square root of a number, S, that is stored at a known address in RAM. The method involves iteration: given an approximate version of the square root of S (call that x_n) an improved approximation (which we'll call x_{n+1}) is:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right) \tag{1}$$

   (You can find information on this method at wiki—it's based on Newton's Method of root finding, but an easier approach is to think of x_n and S/x_n as a 'factorization' of S. If x_n is the square root the two factors will be the same. If x_n is a bit smaller (larger) than the square root, the other factor will be a bit larger (smaller). In either case, averaging the two factors will provide an improved guess.) The overall plan is to run this improvement process until an improvement step hardly changes x at all (i.e., the percent change in x is small: $|x_{n+1} - x_n|/x_{n+1} < \epsilon$ where ϵ is a small number you supply.

   Of course we need a starting guess! For a square root, the exponent of the floating point number needs to be cut in half. Unfortunately the exponent is stored in the 11 bits following the sign bit and stored with an offset (so 0 in the exponent bits actually corresponds to an exponent of about $e = -2^{10} = -1024$ and so a multiplier of about $2^e \approx 5 \times 10^{-309}$). It turns out that halving the exponent looks like this in terms of integer operations:

$$x_0 \leftarrow (i - 2^{52})/2 + 2^{61} \tag{2}$$

   where i is the bit pattern of S thought of as an integer and $\div 2$ is implemented as a right shift by one bit. The supplied code has several errors. Find/report three errors! Rewrite Eqs. (1–2) on your answer sheet. Label three operations expressed in the equations with the corresponding code that does that operation.

2. The famous infinite series for e is:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots$$

   Write code to add up this series until the added term is 'small'.

3 register instructions		
f	add	$A + B \rightarrow D$
f	sub	$A - B \rightarrow D$
f	mul	$A \times B \rightarrow D$
f	div	$A \div B \rightarrow D$
	and	$A \,.\, B \rightarrow D$
	or	$A + B \rightarrow D$
	xor	$A \oplus B \rightarrow D$
	asl	shift A: B digits left $\rightarrow D$
	asr	shift A: B digits right $\rightarrow D$
	bic	clear the Bth digit of $A \rightarrow D$
	bis	set Bth digit of $A \rightarrow D$
2 register instructions		
finv		$1/B \rightarrow D$
f	abs	$\lvert B \rvert \rightarrow D$
f	neg	$-B \rightarrow D$
f	cmp	$A - B$ update CC
	i2f	integer B to float in D
	f2i	float B to integer in D
	inc	$B + 1 \rightarrow D$
	dec	$B - 1 \rightarrow D$
	mov	$B \rightarrow D$
	not	$\overline{B} \rightarrow D$
	load	$(B) \rightarrow D$
	store	$A \rightarrow (B)$
	bit	is the Bth digit of A 0 or 1: update CC
PC update instructions		
	br	same as: mov B rF
	beq	if CC=0: mov B rF
	bne	if CC$\neq$0: mov B rF
	bpl	if CC>0: mov B rF
	bmi	if CC<0: mov B rF
	bb1	if CC bit=1: mov B rF
	bb0	if CC bit=0: mov B rF
1 register instructions		
f	tst	update CC based on B
	clr	$0 \rightarrow D$
CC instructions		
	ccc	clear all CC
	clz	clear zero CC
	cln	clear negative CC
	clb	clear bit CC
	scc	set all CC
	sez	set zero CC
	sen	set negative CC
	seb	set bit CC