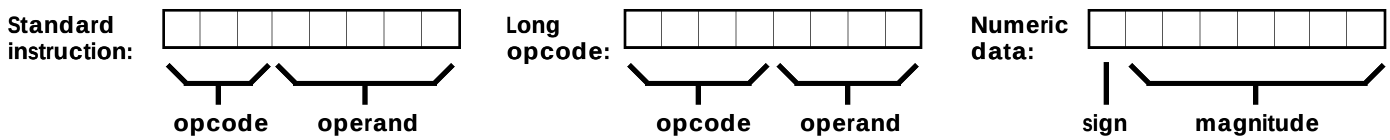
PC 101 Machine Description

Architecture and storage formats

The PC 101 machine has 4 main registers, an accumulator, an instruction register, a program counter and 32 RAM memory locations, numbered from 0 to 31 (for more detail and a graphic overview, see the *PC 101 Machine Architecture* handout from lecture). Each of these various storage areas holds exactly 8 bits. Machine instructions are structured as a 3 bit opcode plus up to 5 bits of operand specification (the standard format) or 4 bits of opcode and up to 4 bits of operand specification (a “long opcode” used to accommodate a larger instruction set).



The operand portion may be 2 bits long (to specify one of the 4 registers) or 5 bits long (to specify a memory location); the operand bits are always taken to be the right-most ones in the instruction. Unused bits (those which are neither part of the opcode nor part of an operand specification) are ignored by the machine.

Numeric data values are represented as follows: the first bit represents the sign of the number, 0 for positive numbers and 1 for negative numbers. The last 7 bits represent a magnitude as a binary number. Thus, for example, `00000011` represents positive 3 whereas `10000101` represents negative 5.

Instruction set *(bits marked with `xxx` are ignored by the machine)*

Shorthand	**Opcode**	**Format**	**Explanation**
`PRINT`	`0000`	`0000xxxx`	Print (or display) the value in the accumulator
`READ`	`0010`	`0010xxxx`	Read a value from the keyboard into the accumulator
`ADD`	`010`	`010xxxRR`	Add the value in register `RR` to the accumulator
`SUB`	`011`	`011xxxRR`	Subtract the value in register `RR` from the accumulator
`LOAD`	`100`	`100nnnnn`	Load from RAM location `nnnnn` into the accumulator
`STORE`	`101`	`101nnnnn`	Store from the accumulator into RAM location `nnnnn`
`PJUMP`	`110`	`110nnnnn`	Jump to memory location `nnnnn` *if* the value in the accumulator is positive (greater than 0)
`STOP`	`111`	`111xxxxx`	Stop the computer
`COPY`	`0001`	`0001xxRR`	Copy the value in register `RR` into the accumulator
`MOVE`	`0011`	`0011xxRR`	Move a value from the accumulator into register `RR`

Basic operation

We will assume that the machine starts with a program (and any data values needed) loaded into RAM. All unspecified values in the RAM will start out with contents `00000000`. All registers, the accumulator and the program counter start out with contents `00000000` (in particular, the program counter will therefore refer to the first instruction in memory). Once the machine is started up, it operates as follows: the contents of a memory location are loaded into the instruction register (based on the address in the last 5 bits of the program counter); the program counter is incremented by 1; the instruction is decoded and performed (NB: possibly changing the program counter, in the case of a jump!); finally, this whole process is repeated until a `STOP` instruction is executed.

Some sample programs

❶ *Read in a number; print out the value of the number multiplied by 5:*

Machine code	**Explanation**
`00100000`	Read the input number into the accumulator
`00110001`	Move the input number from the accumulator into register 1
`01000001`	Add register 1 to the accumulator (four times)
`01000001`	Add register 1 to the accumulator (four times)
`01000001`	Add register 1 to the accumulator (four times)
`01000001`	Add register 1 to the accumulator (four times)
`00000000`	Print the sum from the accumulator
`11100000`	Stop machine

❷ *Read in two numbers and print out their sum:*

Machine code	**Explanation**
`00100000`	Read first number into the accumulator
`00110001`	Move first number from the accumulator into register 1
`00100000`	Read second number into the accumulator
`01000001`	Add register 1 to the accumulator
`00000000`	Print sum from the accumulator
`11100000`	Stop machine

❸ *Read in two numbers and print out their product:*

Our overall approach will be to repeatedly add the first number (the *multiplicand)* into a running sum, as many times as indicated by the second number (the *multiplier).* We will decrease the multiplier by 1 each time we add, and stop when it reaches 0. Since we have only a single accumulator in which to do the arithmetic, and several values to store over time, we will need to use the registers as temporary storage. Our plan will be to store the multiplicand in register 1 (R1), the multiplier in register 2 (R2) and the current sum (which will be built up in the accumulator) in register 3 (R3). In each step of the problem, we will increase the current sum by the multiplicand and decrease the multiplier by 1. This leads to the following solution sketch:

```
            read the multiplicand, R1, and the multiplier, R2
test:       if done, jump to the exit (we are done when the multiplier, R2, equals 0)
            decrease the multiplier by a constant value of 1 (R2=R2-1)
            increase the current sum by the multiplicand (R3=R3+R1)
            jump back to the test
exit:       print the current sum and stop
```

As we try to fill out this basic sketch, we will need to overcome several difficulties:

- how do we resolve the named "labels", such as test and exit, that we use above as the targets of jumps?
  Solution: we need to number our instructions, and figure out the numeric address of each "label", then build that address into any corresponding jump instruction.

- how do we subtract a constant value 1 from the multiplier?
  Solution: we can use register 0 (R0) to store a constant value; we can load it in from RAM when the program starts.

- how do we *just* jump, without checking any condition (as needed in the step just before the exit label)?
  Solution: we can `LOAD` the constant 1 and use a `PJUMP`; since 1 is positive, it will always jump.

- how do we jump when R2 = 0 (at the "test" step) rather than jumping on a positive value (using `PJUMP`)?
  Solution: We can use a *two-step* jump: first use `PJUMP` to jump on positive to the normal continuation of the program, then jump from right after the first `PJUMP` to the program exit point

Read in two numbers and print out their product (the completed program):

Step	Machine code		Explanation
0	`10010100`		load constant 1 from RAM
1	`00110000`		move constant 1 into R0
2	`00100000`		read multiplicand into the accumulator
3	`00110001`		move multiplicand into R1
4	`00100000`		read multiplier into the accumulator
5	`00110010`		move multiplier into R2
6	`00010010`	test:	copy multiplier (R2) into the accumulator (redundant the 1st time around)
7	`11001010`		pjump to more
8	`00010000`		copy constant 1 (R0) into the accumulator (just for the jump)
9	`11010001`		pjump to exit
10	`01100000`	more:	subtract constant 1 (R0) from the accumulator (currently holds multiplier)
11	`00110010`		move the accumulator to R2
12	`00010011`		copy current sum (R3) into the accumulator
13	`01000001`		add multiplicand (R1) to the accumulator
14	`00110011`		move the accumulator to current sum (R3)
15	`00010000`		copy constant 1 (R0) into the accumulator
16	`11000110`		pjump to test
17	`00010011`	exit:	copy current sum (R3) into the accumulator
18	`00000000`		print the accumulator
19	`11100000`		stop
20	`00000001`		the constant 1

Some challenge problems

If you are interested in a challenge, try writing a program to solve one of the following problems using the PC 101 machine:

- read in two numbers and *divide* the first by the second; print out an integer dividend and an integer remainder.

  Suggestions: your basic approach can be similar to the multiplication problem above, but you will need to repeatedly subtract instead of repeatedly adding. Also, rather than holding a fixed multiplier, you will repeatedly increase the dividend by 1. In many cases, you will pass 0 using this strategy, so you will need to “back up” a step to determine the proper remainder.

- read in 10 numbers and then print them out in reverse order.

  Suggestions: your basic approach would be to store the numbers into RAM, one at a time into successive storage locations, and then print them back out, loading them in one at a time in reverse order. Note that you will need to keep track of your current position in the RAM and either increase or decrease it by 1, depending on which direction you are going. In order to access RAM using this “number”, you will need to keep the number in the operand specification part of a `LOAD` or `STORE` instruction; this in turn means that you will need to *perform arithmetic on the instruction itself* in order to change the location it specifies.

Read in ten numbers and print them out in reverse order:

In this first attempt, we use two loops, one to read in the numbers and one to print them out … but the overhead of loop maintainence and comparisons, etc., takes up too many instructions: we can’t even fit in the program, much less the 10 RAM locations needed for values.

Step	Machine code		Explanation
0	00000000		load constant 1 into the accumulator
1	00000000		move constant 1 into R0
2	00000000		load initial constant 10 into the accumulator
3	00000000		move initial constant 10 into counter (R1)
4	00000000	test 1:	copy counter from R0 into the accumulator (redundant 1st time)
5	00000000		jump to step1 if accumulator is positive
6	00000000		set up for unconditional jump
7	00000000		jump to part 2
8	00000000	step 2:	read number from input to the accumulator
9	00000000	store:	store number from accumulator into RAM
10	00000000		load the store instruction into the accumulator
11	00000000		add 1 to the store instruction, increasing its address
12	00000000		store modified instruction into RAM
13	00000000		load counter (R1) into the accumulator
14	00000000		subtract constant 1 (R0) from the counter
15	00000000		move counter back to R1
16	00000000		set up for unconditional jump
17	00000000		jump to test 1
18	00000000	part 2:	load initial constant 10 into the accumulator
19	00000000		move initial constant 10 into counter (R1)
20	00000000	test 2:	copy counter from R0 into the accumulator (redundant 1st time)
21	00000000		jump to step2 if accumulator is positive
22	00000000		set up for unconditional jump
23	00000000		jump to exit
24	00000000	step 2:	load number from RAM into the accumulator
25	00000000		print number from accumulator
26	00000000		load the load instruction into the accumulator
27	00000000		subtract 1 from the load instruction, decreasing its address
28	00000000		store modified instruction into RAM
29	00000000		load counter (R1) into the accumulator
30	00000000		subtract constant 1 (R0) from the counter
31	00000000		move counter back to R1
32	00000000		set up for unconditional jump
33	00000000		jump to test 2
34	00000000	exit:	stop the machine

In this second attempt, we use two loops, as before, but now we don't use a counter from 10; rather, we directly compare the modified instruction value to see if it meets some completion condition. If we do the comparisons right, we can avoid the two-step jumps used above.

Step	**Machine code**		**Explanation**
0	`00000000`		load constant 1 into the accumulator
1	`00000000`		move constant 1 into R1
2	`00000000`		load constant FINAL-STORE into the accumulator
3	`00000000`		move constant FINAL-STORE into R0
4	`00000000`	test 1:	read number from input to the accumulator
5	`00000000`	store:	store number from accumulator into RAM
6	`00000000`		load current store instruction into the accumulator
7	`00000000`		add 1 to the store instruction, increasing its address
8	`00000000`		store modified instruction into RAM
9	`00000000`		subtract constant FINAL-STORE from store instruction
10	`00000000`		jump back to test 1 if accumulator is positive
11	`00000000`	part 2:	load constant FINAL-LOAD into the accumulator
12	`00000000`		move constant FINAL-LOAD into R0
13	`00000000`	load:	load number from RAM to the accumulator
14	`00000000`		print number from the accumulator
15	`00000000`		load current load instruction into the accumulator
16	`00000000`		subtract 1 from the load instruction, decreasing its address
17	`00000000`		store modified instruction into RAM
18	`00000000`		subtract constant FINAL-LOAD from store instruction
19	`00000000`		jump back to test 2 if accumulator is positive
20	`00000000`	exit:	stop the machine
21	`00000000`	F-S:	the constant FINAL-STORE
22	`00000000`	F-L:	the constant FINAL-LOAD
23	`00000000`	one:	the constant 1

In this next attempt, we use two loops and compare modified instruction values, as before, but now we keep temporary copies of the instructions in registers to make it easier to get at them.

Step	**Machine code**		**Explanation**
0	`00000000`		load constant 1 into the accumulator
1	`00000000`		move constant 1 into R1
2	`00000000`		load constant FINAL-STORE into the accumulator
3	`00000000`		move constant FINAL-STORE into R0
2	`00000000`		load store1 instruction into the accumulator
3	`00000000`		move store1 instruction into R2
4	`00000000`	test 1:	load current store instruction into the accumulator
5	`00000000`		add 1 to the store instruction, increasing its address
6	`00000000`		move modified instruction into R2
7	`00000000`		subtract constant FINAL-STORE from store instruction
8	`00000000`		jump to part 2 if accumulator is positive
9	`00000000`		read number from input to the accumulator
10	`00000000`	store:	store number from accumulator into RAM
11	`00000000`		jump to test 1 (unconditional, assumes only positive inputs)
12	`00000000`	part 2:	load constant FINAL-LOAD into the accumulator
13	`00000000`		move constant FINAL-LOAD into R0
14	`00000000`	test 2:	load current load instruction into the accumulator
15	`00000000`		subtract 1 from the load instruction, decreasing its address
16	`00000000`		store modified instruction into RAM
17	`00000000`		subtract constant FINAL-LOAD from store instruction
18	`00000000`		jump to exit if accumulator is positive
19	`00000000`	load:	load number from RAM to the accumulator
20	`00000000`		print number from the accumulator
21	`00000000`		jump to test 2 (unconditional, assumes only positive inputs)

22	00000000	exit:	stop the machine
23	00000000	F-S:	the constant FINAL-STORE
24	00000000	F-L:	the constant FINAL-LOAD
25	00000000	one:	the constant 1

In this fourth and final attempt, we use a single loop to serve for both storage and loading: in between the loops, we modify the store instruction to become a load and modify the incrementation constant from 1 to -1.

Step	**Machine code**		**Explanation**
0	00000000		load constant CHECK1 into the accumulator
1	00000000		move constant CHECK1 into R1
2	00000000		load initial increment constant (1) into the accumulator
3	00000000		move increment constant into R0
4	00000000	test 1:	load funny instruction into the accumulator
5	00000000		add increment constant to the funny instruction
6	00000000		store modified instruction into RAM
7	00000000		subtract check constant from funny instruction
8	00000000		jump to part 2 if accumulator is positive
9	00000000		read number from input to the accumulator
10	00000000	funny:	store number from accumulator into RAM
11	00000000		set up for unconditional jump
12	00000000		jump to test 1
13	00000000	part 2:	load constant CHECK2 into the accumulator
14	00000000		move constant CHECK2 into R1
15	00000000		load decrement constant (-1) into the accumulator
16	00000000		move decrement constant into R0
17	00000000		[swap instructions??]
18	00000000		[fix target of jump??]
19	00000000		set up for unconditional jump
20	00000000		jump to test 2
21	00000000	exit:	stop the machine
22	00000000	[VAR]:	various constants

Assuming RAM loaded with zeros after program, is there a better choice for opcode 00000000 than PRINT?

What happens if we change the third instruction of sample program 2 to ... (ignore xx bits)

Write a program to read in a number, add 5 to it, print it out again (5 in binary is 101).

Binary	Explanation
------	-----------
	Load constant 5 from RAM into the accumulator
00110001	Move constant 5 into register 1
00100000	Read first number into the accumulator
01000001	Add register 1 to the accumulator
00000000	Print sum from the accumulator
11100000	Stop machine
00000101	The constant 5