Circuits for Arithmetic; Assembly Language: A First Look

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

2-Bit Binary Addition

- The truth table is at right.
  - 4 bits of input
  - 3 bits of output
- In theory, we could use the minterm-expansion approach.
  - How many circuits do we need?
  - How many minterms do we need?

binary inputs **A** and **B**		output, **A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110
A	**B**	

2-Bit Binary Addition

- The truth table is at right.
  - 4 bits of input
  - 3 bits of output
- In theory, we could use the minterm-expansion approach.
  - How many circuits do we need?
    - **3 circuits as there are three bits of output**
  - How many minterms do we need?

binary inputs **A** and **B**		output, **A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110
A	**B**	

2-Bit Binary Addition

- The truth table is at right.
  - 4 bits of input
  - 3 bits of output
- In theory, we could use the minterm-expansion approach.
  - How many circuits do we need?
    - 3 circuits as there are three bits of output
  - How many minterms do we need?
    - 22 minterms to cover all 1s in the output

binary inputs **A** and **B**		output, **A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110
A	**B**	

2-Bit Binary Addition

- The truth table is at right.
  - 4 bits of input
  - 3 bits of output
- In theory, we could use the minterm-expansion approach to create ***3 circuits***.
  - one for each output bit
- It ends up being overly complicated.
  - more gates than are really needed
- Instead, we'll take advantage of two things:
  - our elementary-school bitwise-addition algorithm
  - modular design!

binary inputs **A** and **B**		output, **A+B**
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110
A	**B**	

A Full Adder

- Recall our bitwise algorithm:

```
        0 1 1
      101101
  +   001110
  ----------
      111011
```

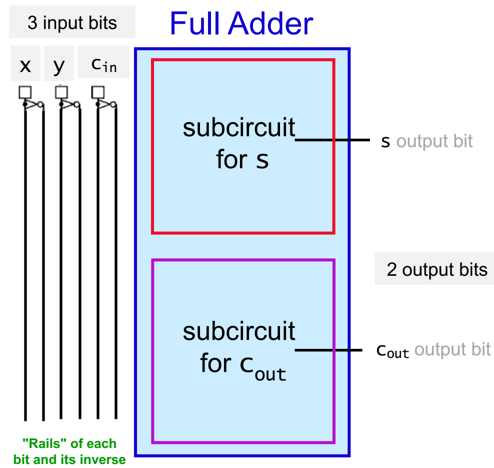
- A *full adder* adds only one column.
- It takes 3 bits of input:
  - x and y – one bit from each number being added
  - c_{in} – the carry bit *into* the current column
- It produces 2 bits of output:
  - s – the bit from the sum that goes at the bottom of the column
  - c_{out} – the carry bit *out of* the current column
    - it becomes the c_{in} of the next column!

inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	**0**	**1**	**0**	**1**
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Building a Full Adder

Create a separate minterm expansion/circuit **for each output bit**!

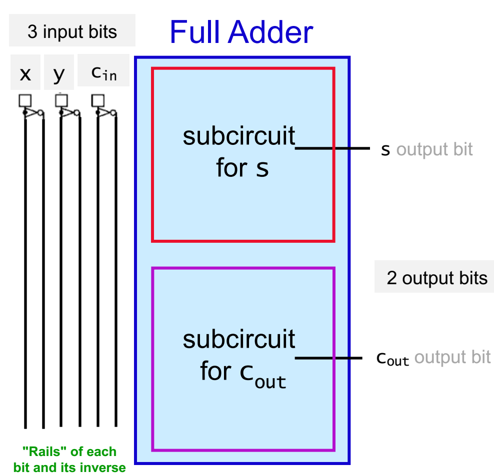
inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



How many AND gates will you need in all?

Create a **separate** minterm expansion/circuit **for each output bit**!

inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

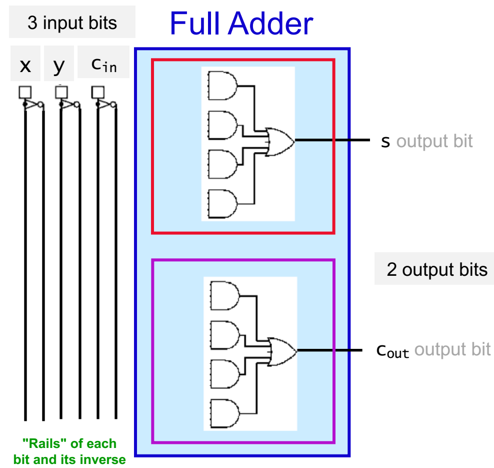


A. 4 B. 7 C. 8 D. 16

How many AND gates will you need in all?

Create a **separate** minterm expansion/circuit **for each output bit**!

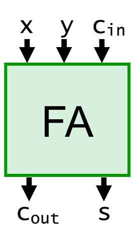
inputs			outputs	
x	y	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



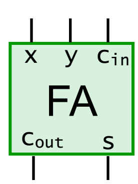
A. 4   B. 7   C. **8**   D. 16

Modular Design

- Once we have a full adder, we can treat it as an *abstraction* – a "black box" with 3 inputs and two outputs.

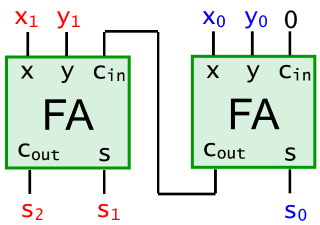


- Here's another way to draw it:



Modular Design (cont.)

- To add 2-bit numbers, combine two full adders!



$$\begin{array}{r} x_1 x_0 \\ +\ y_1 y_0 \\ \hline s_2 s_1 s_0 \end{array}$$

- Produces what is known as a *2-bit ripple-carry adder.*

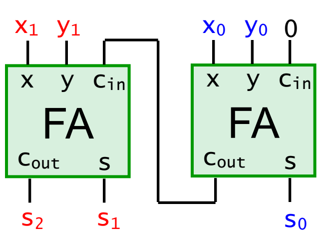
In PS 5, you'll build a 4-bit version!

- To add larger numbers, combine even more FAs!

- More efficient than an adder built using minterm expansion.
  - 16-bit minterm-based adder: need several *billion* gates
  - 16-bit ripple-carry adder: only need *hundreds* of gates

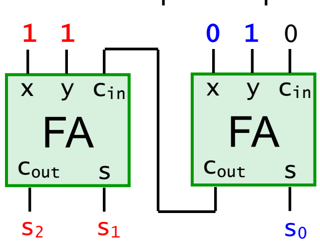
2-Bit Ripple-Carry Adder

- Schematic:



$$\begin{array}{r} x_1 x_0 \\ +\ y_1 y_0 \\ \hline s_2 s_1 s_0 \end{array}$$

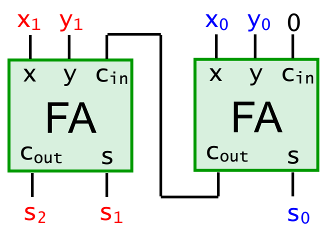
- Here's an example computation:



$$\begin{array}{r} 10 \\ +\ 11 \\ \hline \end{array}$$

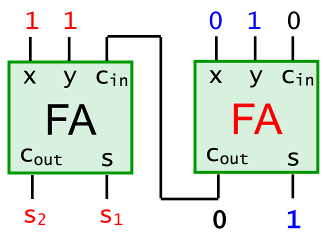
2-Bit Ripple-Carry Adder

- Schematic:



$$\begin{array}{r} x_1 x_0 \\ +\ y_1 y_0 \\ \hline s_2 s_1 s_0 \end{array}$$

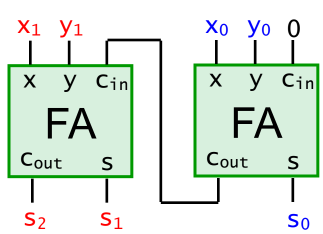
- Here's an example computation:



$$\begin{array}{r} 0\ \ \\ 10 \\ +\ \ 11 \\ \hline 1 \end{array}$$

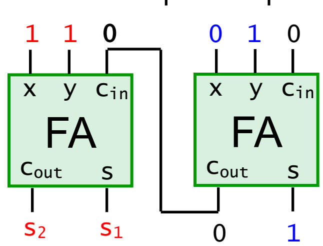
2-Bit Ripple-Carry Adder

- Schematic:



$$\begin{array}{r} x_1 x_0 \\ +\ y_1 y_0 \\ \hline s_2 s_1 s_0 \end{array}$$

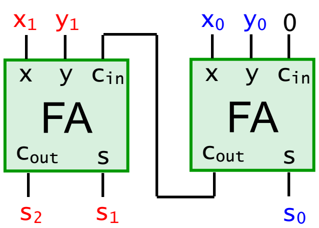
- Here's an example computation:



$$\begin{array}{r} 0\ \ \\ 10 \\ +\ \ 11 \\ \hline 1 \end{array}$$

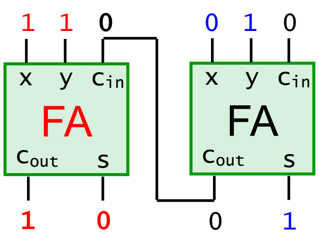
2-Bit Ripple-Carry Adder

- Schematic:



$$\begin{array}{r} x_1x_0 \\ +\; y_1y_0 \\ \hline s_2s_1s_0 \end{array}$$

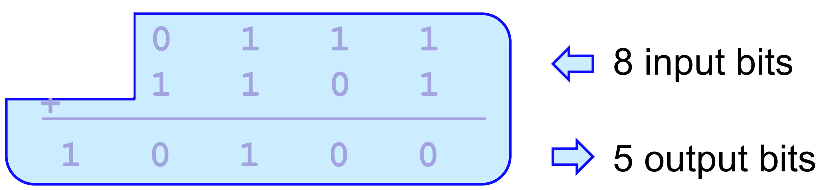
- Here's an example computation:



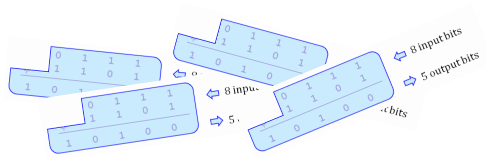
$$\begin{array}{r} \scriptstyle 0 \\ 10 \\ +\quad 11 \\ \hline 101 \end{array}$$

More Modular Design!

- Once you build a 4-bit ripple-carry adder, you can treat it as a "black box".



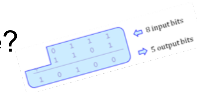
- Use these boxes to build other circuits!



Also in PS 5: Building a 4x2 Multiplier

```
      1   1   0   1     first factor (4 bits)
x             1   0     second factor (2 bits)
---------------------
  0   0   0   0   0     2 partial products
  1   1   0   1
---------------------
  1   1   0   1   0     final answer
```

- How could you use a 4-bit ripple-carry adder here?
  to combine the partial products (which bits?)



- What other smaller circuit might we want to build first so that we can use it as part of the 4 x 2 multiplier?
  - a 4x1 multiplier!
  - use two of them – one to compute each partial product

Two Key Components of a Computer

program + data

CPU ⟷ **RAM**

central processing unit — random access memory

CPU:
- all computation happens here
- adders, multipliers, etc.
- small number of *registers* for storing values

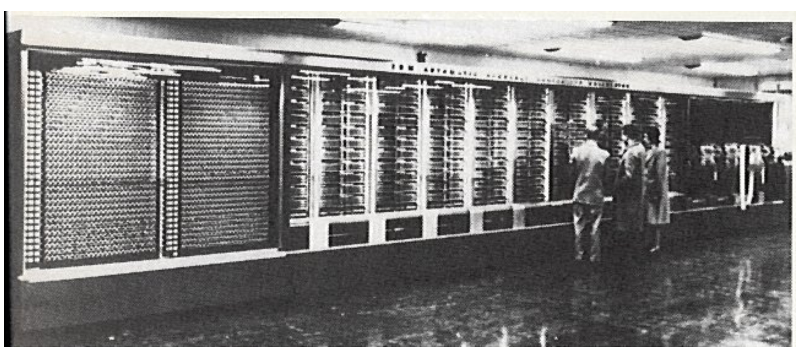
RAM:
- lots of room for storage
- no computation happens here

- Program instructions are stored *with the data* in RAM.
- Instructions and data are transferred back and forth between RAM and the CPU.

von Neumann Architecture

- John von Neumann was the one who proposed storing programs in memory.

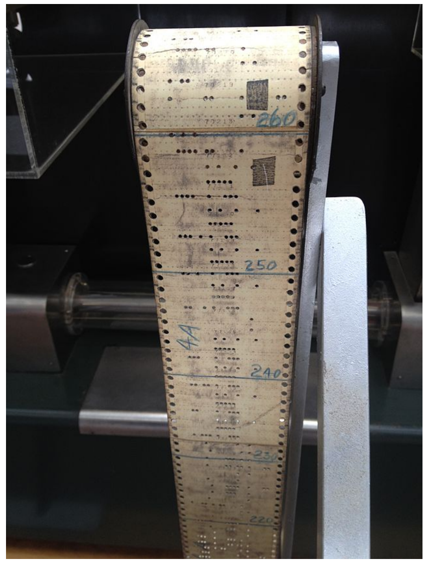
Early Computers



The Mark I: Howard Aiken, Grace Hopper, et al.; Harvard, the 1940s/50s

- In the first computers, programs were stored *separately* from the data.

Early Computers (cont.)



A sample program in machine language:

0000 0001 0000 0001
1000 0010 0001 0001
0110 0010 0010 0001
0000 0010 0000 0010
0000 0000 0000 0000

an external program tape for the Mark I

Assembly Language

processing — **CPU** — central processing unit ⟷ program + data — **RAM** — random access memory

0000 0001 0000 0001	0	`read r1`
1000 0010 0001 0001	1	`mul r2 r1 r1`
0110 0010 0010 0001	2	`add r2 r2 r1`
0000 0010 0000 0010	3	`write r2`
0000 0000 0000 0000	4	`halt`
	5	
	6	

- Assembly language is *human-readable* machine language.
  - use mnemonics instead of the actual bits
  - Each assembly language is specific to a particular computer architecture and operating system.

Hmmm (Harvey Mudd Miniature Machine)

processing		program + data
CPU	⟷	**RAM**
central processing unit		random access memory

r1: (general-purpose register r1)

r2: (general-purpose register r2)

...

0	read r1
1	mul r2 r1 r1
2	add r2 r2 r1
3	write r2
4	halt
5	
6	

...

15 general-purpose registers (r1-r15)

256 memory locations

quick reference and full documentation:
http://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html

Hmmm Assembly: Basic Instructions

instruction	**what it does**	**example**
`read` rX	reads from keyboard into rX	`read r1`
`write` rX	writes value of rX to screen	`write r2`
`add` rX rY rZ	rX = rY + rZ	`add r1 r1 r3`
`sub` rX rY rZ	rX = rY – rZ	`sub r4 r3 r2`
`mul` rX rY rZ	rX = rY * rZ	`mul r3 r1 r2`
`div` rX rY rZ	rX = rY // rZ	`div r2 r5 r6`
`mod` rX rY rZ	rX = rY % rZ	`mod r2 r1 r3`
`setn` rX **n**	rX = **n**	`setn r1 7`
`addn` rX **n**	rX = rX + **n**	`addn r1 -1`

Notation:

- rX, rY, rZ is any register name (`r1-r15`)
- **n is any integer from -128 to 127**

What Does This Program Output?

Screen: 42 (input)

RAM

random access memory

A. 2
B. 3
C. 2.6666666666666665
D. 3.6666666666666665
E. none of these

```
0  read r1
1  setn r2 9
2  sub r3 r1 r2
3  div r3 r3 r2
4  addn r3 -1
5  write r3
6  halt
```

What Does This Program Output?

Screen: 42 (input)

CPU

central processing unit

RAM

random access memory

r1	42
r2	9
r3	33

```
0  read r1
1  setn r2 9
2  sub r3 r1 r2
3  div r3 r3 r2
4  addn r3 -1
5  write r3
6  halt
```

equivalent to:

```
r3 = r1 - r2
     42 - 9
      33
```

What Does This Program Output?

Screen: **42** (input)

CPU (central processing unit) ⟷ **RAM** (random access memory)

r1	42
r2	9
r3	3

Address	Instruction
0	`read r1`
1	`setn r2 9`
2	`sub r3 r1 r2`
3	`div r3 r3 r2`
4	`addn r3 -1`
5	`write r3`
6	`halt`

equivalent to:

```
r3 = r3 // r2
     33 // 9
     3
```

What Does This Program Output?

Screen: **42** (input)

CPU (central processing unit) ⟷ **RAM** (random access memory)

r1	42
r2	9
r3	2

Address	Instruction
0	`read r1`
1	`setn r2 9`
2	`sub r3 r1 r2`
3	`div r3 r3 r2`
4	`addn r3 -1`
5	`write r3`
6	`halt`

equivalent to:

```
r3 = r3 + (-1)
     3  + (-1)
     2
```

What Does This Program Output?

Screen

42 (input)

2 (output)

CPU — central processing unit

r1	42
r2	9
r3	2

RAM — random access memory

```
0  read r1
1  setn r2 9
2  sub r3 r1 r2
3  div r3 r3 r2
4  addn r3 -1
5  write r3
6  halt
```

What Does This Program Output?

Screen

42 (input)

RAM — random access memory

```
0  read r1
1  setn r2 9
2  sub r3 r1 r2
3  div r3 r3 r2
4  addn r3 -1
5  write r3
6  halt
```

A. 2

B. 3

C. 2.6666666666666665

D. 3.6666666666666665

E. none of these

Recall: Binary Data

- All values in the computer are stored as binary numbers.
  - sequences of *bits* (0s and 1s)
- With n bits, we can represent 2^n different values.
  - 2 bits give $2^2 = 4$ different values
    00, 01, 10, 11
  - 3 bits give $2^3 = 8$ different values
    000, 001, 010, 011, 100, 101, 110, 111

Data in Hmmm

- Hmmm only works with integers.
- Each register and memory address is 16 bits wide.
  - 16 bits give 2^{16} = 65,536 different values

A sample program in machine language:

0000 0001 0000 0001
1000 0010 0001 0001
0110 0010 0010 0001
0000 0010 0000 0010
0000 0000 0000 0000

Data in Hmmm

- Hmmm only works with integers.
- Each register and memory address is 16 bits wide.
    - 16 bits give 2^{16} = 65,536 different values
- Thus, we can store any integer from -32,768 to 32,767.
    - the user can enter integers in this range
    - our computations can produce results in this range

Data in Hmmm (cont.)

- We're more limited when we include an integer in an instruction:

```
setn r1 7
addn r2 1
jeqz r2 25
```

- Hmmm instructions are also 16 bits wide.
    - example: **`addn r2 1`**

      corresponds to this Hmmm machine-language instruction:

      `0101001000000001`

Data in Hmmm (cont.)

- We're more limited when we include an integer in an instruction:

```
setn r1 7
addn r2 1
jeqz r2 25
```

- Hmmm instructions are also 16 bits wide.
  - example: **addn r2 1**

    corresponds to this Hmmm machine-language instruction:

    `0101001000000001`

    - first 4 bits say it is **addn**

Data in Hmmm (cont.)

- We're more limited when we include an integer in an instruction:

```
setn r1 7
addn r2 1
jeqz r2 25
```

- Hmmm instructions are also 16 bits wide.
  - example: **addn r2 1**

    corresponds to this Hmmm machine-language instruction:

    `0101001000000001`

    - first 4 bits say it is **addn**
    - next 4 bits specify the register **r2**

Data in Hmmm (cont.)

- We're more limited when we include an integer in an instruction:

```
setn r1 7
addn r2 1
jeqz r2 25
```

- Hmmm instructions are also 16 bits wide.
  - example: `addn r2 1`

    corresponds to this Hmmm machine-language instruction:

    `0101001000000001`

    - first 4 bits say it is `addn`
    - next 4 bits specify the register `r2`
    - the last 8 bits specify the number
      → only 2^8 = 256 possible values!
      - -128 to 127 for arithmetic instructions
      - 0 to 255 for jumps

Jumps in Hmmm

instruction	**what it does**	**example**
`jeqz` rX L	jumps to line L if rX == 0	`jeqz r1 12`

Notation:

- rX is any register name (`r1-r15`)
- L is the line number of an instruction

Jumps in Hmmm

instruction	what it does	example
`jeqz` rX L	jumps to line L if rX == 0	`jeqz r1 12`
`jgtz` rX L	jumps to line L if rX > 0	`jgtz r2 4`
`jltz` rX L	jumps to line L if rX < 0	`jltz r3 15`
`jnez` rX L	jumps to line L if rX != 0	`jnez r1 7`

Notation:

- rX is any register name (**`r1-r15`**)
- L is the line number of an instruction

Jumps in Hmmm

instruction	what it does	example
`jeqz` rX L	jumps to line L if rX == 0	`jeqz r1 12`
`jgtz` rX L	jumps to line L if rX > 0	`jgtz r2 4`
`jltz` rX L	jumps to line L if rX < 0	`jltz r3 15`
`jnez` rX L	jumps to line L if rX != 0	`jnez r1 7`

| `jumpn` L | jumps to line L | `jumpn 6` |

Notation:

- rX is any register name (**`r1-r15`**)
- L is the line number of an instruction

Jumps in Hmmm

instruction	what it does	example
jeqz rX L	jumps to line L if rX == 0	**jeqz r1 12**
jgtz rX L	jumps to line L if rX > 0	**jgtz r2 4**
jltz rX L	jumps to line L if rX < 0	**jltz r3 15**
jnez rX L	jumps to line L if rX != 0	**jnez r1 7**
jumpn L	jumps to line L	**jumpn 6**
jumpr rX (more on this later)	jumps to line # stored in rX	**jumpr r2**

Notation:

- rX is any register name (**r1-r15**)
- L is the line number of an instruction

Hmmm

the complete reference

www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html

Instruction	Description	Aliases
System instructions		
halt	Stop!	
read rX	Place user input in register rX	
write rX	Print contents of register rX	
nop	Do nothing	
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	
addn rX N	Add integer N (-128 to 127) to register rX	
copy rX rY	Set rX = rY	mov
Arithmetic		
add rX rY rZ	Set rX = rY + rZ	
sub rX rY rZ	Set rX = rY - rZ	
neg rX rY	Set rX = -rY	
mul rX rY rZ	Set rX = rY * rZ	
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)	
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	
Jumps!		
jumpn N	Set program counter to address N	
jumpr rX	Set program counter to address in rX	jump
jeqzn rX N	If rX == 0, then jump to line N	jeqz
jnezn rX N	If rX != 0, then jump to line N	jnez
jgtzn rX N	If rX > 0, then jump to line N	jgtz
jltzn rX N	If rX < 0, then jump to line N	jltz
calln rX N	Copy the next address into rX and then jump to mem. addr. N	call
Interacting with memory (RAM)		
loadn rX N	Load register rX with the contents of memory address N	
storen rX N	Store contents of register rX into memory address N	
loadr rX rY	Load register rX with data from the address location held in reg. rY	loadi, load
storer rX rY	Store contents of register rX into memory address held in reg. rY	storei, store