Boolean Arithmetic

Counting systems

quantity	decimal	binary	3-bit register
	0	0	000
✱	1	1	001
✱✱	2	10	010
✱✱✱	3	11	011
✱✱✱✱	4	100	100
✱✱✱✱✱	5	101	101
✱✱✱✱✱✱	6	110	110
✱✱✱✱✱✱✱	7	111	111
✱✱✱✱✱✱✱✱	8	1000	overflow
✱✱✱✱✱✱✱✱✱	9	1001	overflow
✱✱✱✱✱✱✱✱✱✱	10	1010	overflow

Rationale

$$(9038)_{ten} = 9 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 8 \cdot 10^0 = 9038$$

$$(10011)_{two} = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Binary addition

- Assuming a 4-bit system:

$$\begin{array}{ccccc} 0 & 0 & 0 & 1 & \\ \hline & 1 & 0 & 0 & 1 \\ & 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{ccccc} 1 & 1 & 1 & 1 & \\ \hline & 1 & 0 & 1 & 1 \\ & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 \end{array}$$

no overflow　　　　overflow

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) has to be dealt with.

Representing negative numbers (4-bit system)

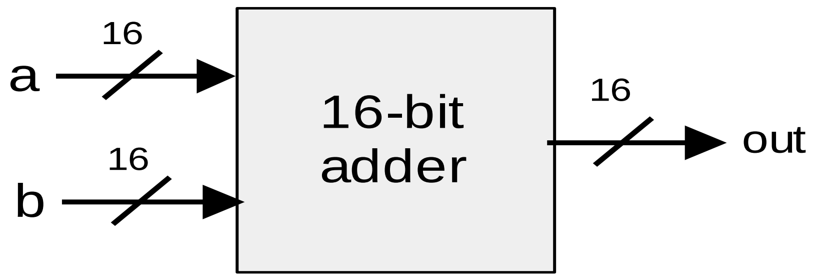
0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

- The codes of all positive numbers begin with a "0"
- The codes of all negative numbers begin with a "1"
- To convert a number:
  leave all trailing 0's and first 1 intact,
  and flip all the remaining bits

Example: 2 - 5 = 2 + (-5) =

```
  0 0 1 0
+ 1 0 1 1
---------
  1 1 0 1     =   -3
```

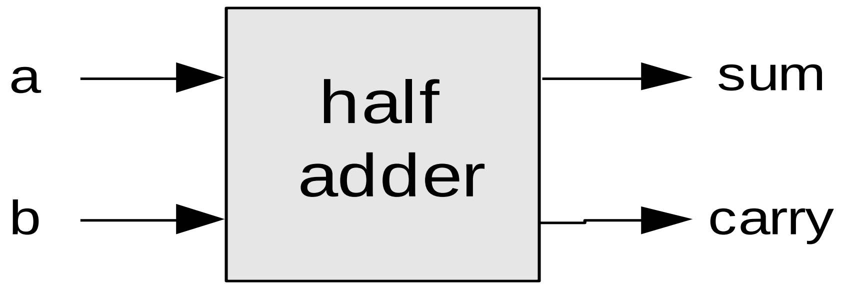
Building an Adder chip



- Adder: a chip designed to add two integers
- Proposed implementation:
  - Half adder: designed to add 2 bits
  - Full adder: designed to add 3 bits
  - Adder: designed to add two *n*-bit numbers.

Half adder (designed to add 2 bits)

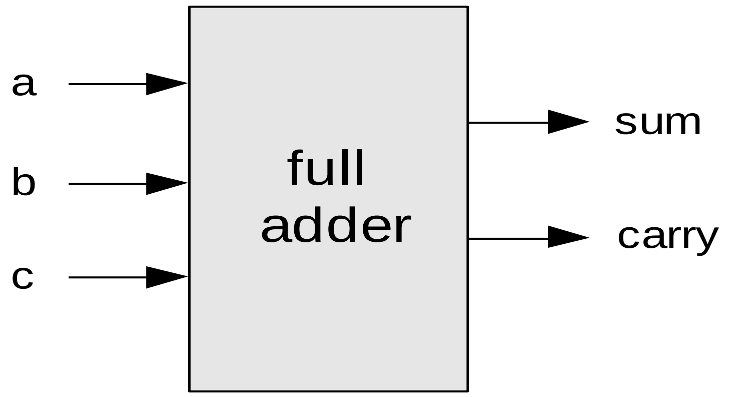
a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



- Implementation: based on two gates that you've seen before.

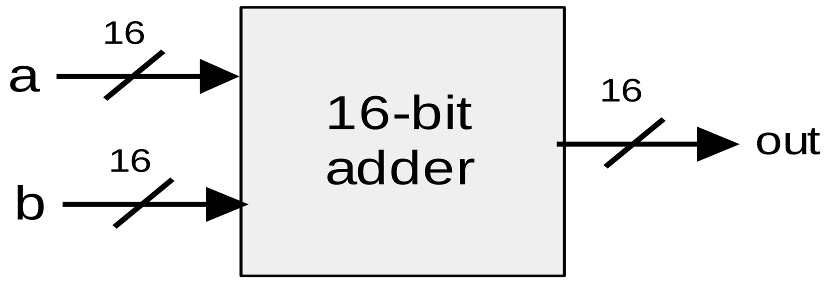
Full adder (designed to add 3 bits)

a	b	c	carry	sum
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



- Implementation: can be based on half-adder gates.

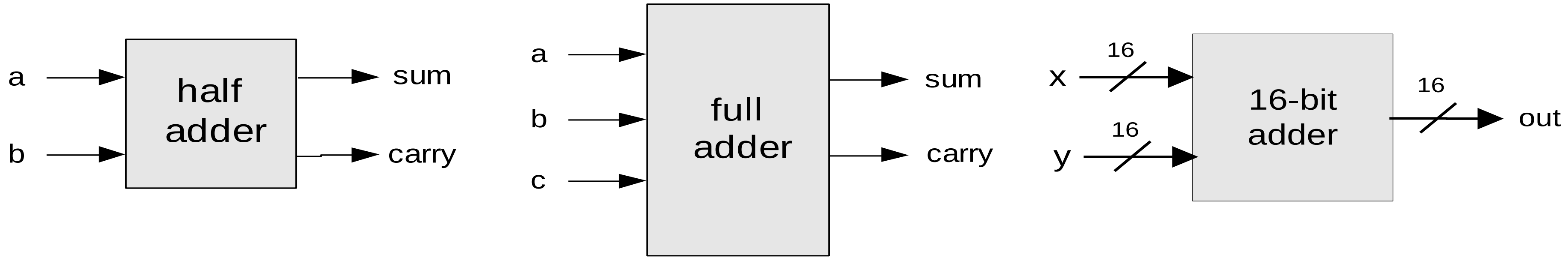
n-bit Adder

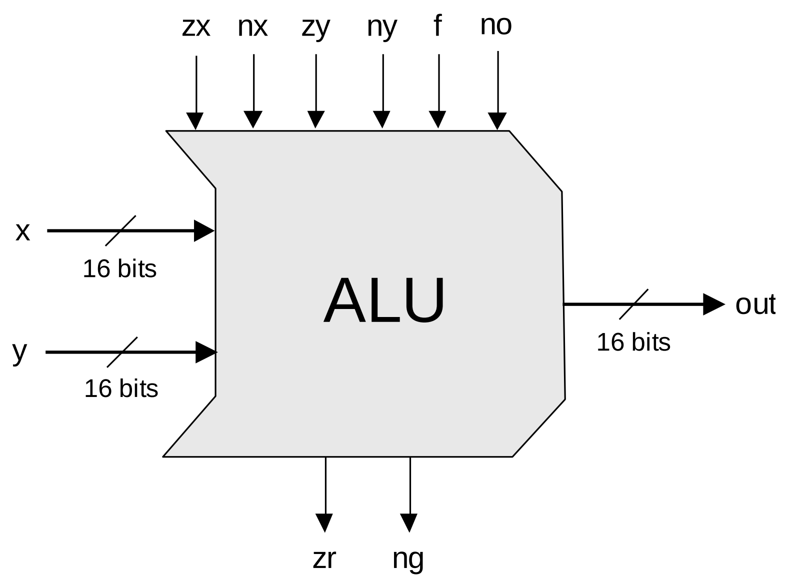


```
...  1   0   1   1     a
                         +
...  0   0   1   0     b
--------------------------
...  1   1   0   1     out
```

- Implementation: array of full-adder gates.

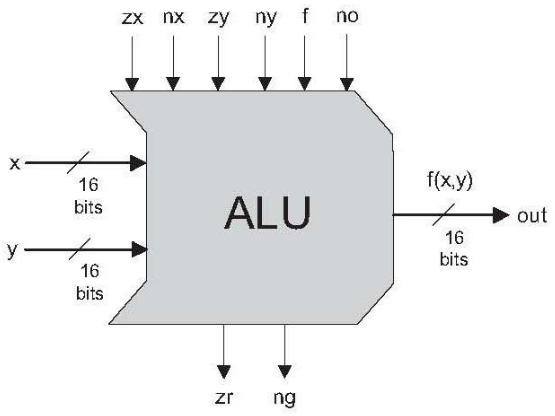
The ALU (of the Hack platform)





out(x, y, control bits**) =**

```
x+y, x-y, y-x,
0, 1, -1,
x, y, -x, -y,
x!, y!,
x+1, y+1, x-1, y-1,
x&y, x|y
```



```
Chip name: ALU
Inputs:    x[16], y[16],    // Two 16-bit data inputs
           zx,              // Zero the x input
           nx,              // Negate the x input
           zy,              // Zero the y input
           ny,              // Negate the y input
           f,               // Function code: 1 for Add, 0 for And
           no               // Negate the out output
Outputs:   out[16],         // 16-bit output
           zr,              // True iff out=0
           ng               // True iff out<0
Function:  if zx then x = 0       // 16-bit zero constant
           if nx then x = !x      // Bit-wise negation
           if zy then y = 0       // 16-bit zero constant
           if ny then y = !y      // Bit-wise negation
           if f then out = x + y  // Integer 2's complement addition
                else out = x & y  // Bit-wise And
           if no then out = !out  // Bit-wise negation
           if out=0 then zr = 1 else zr = 0  // 16-bit eq. comparison
           if out<0 then ng = 1 else ng = 0  // 16-bit neg. comparison
Comment:   Overflow is neither detected nor handled.
```

These bits instruct how to preset the x input		These bits instruct how to preset the y input		This bit selects between + / And	This bit inst. how to postset out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x&y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

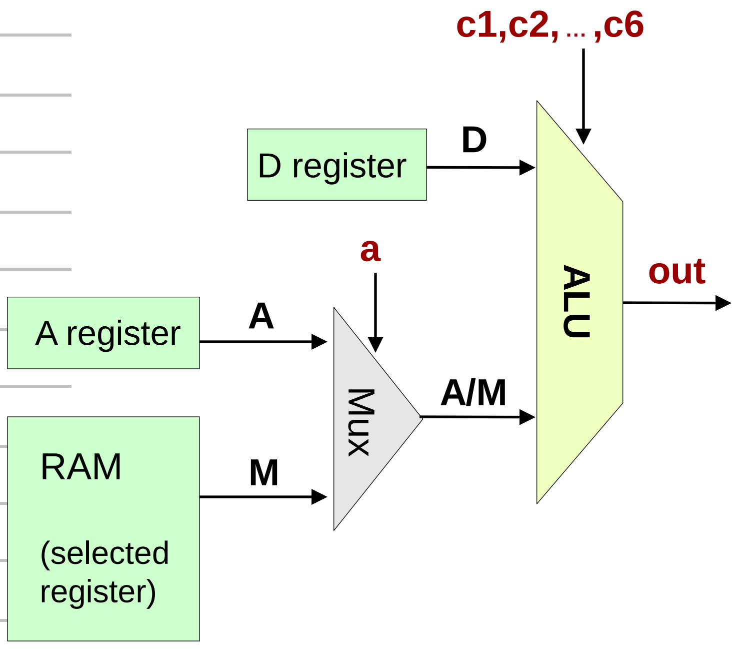
ALU logic (Hack platform)

These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1						y
0						!x
1						!y
0						-x
1						-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Implementation: build a logic gate architecture that "reads" each control bit and does what the table specifies: if zx=1 then set x to 0, etc.

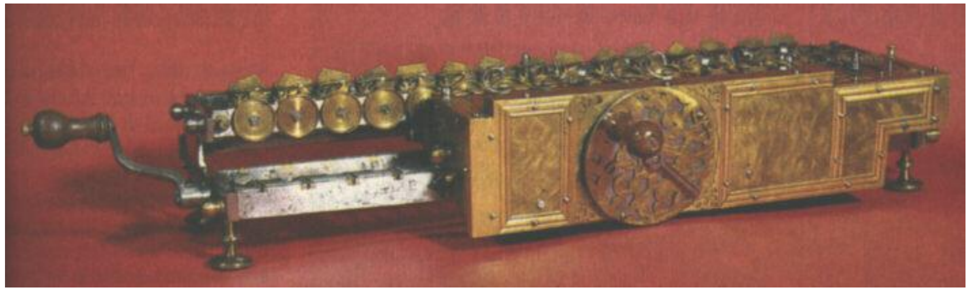
The ALU in the CPU context (Hack platform)

out (when a=0)	c1	c2	c3	c4	c5	c6	out (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

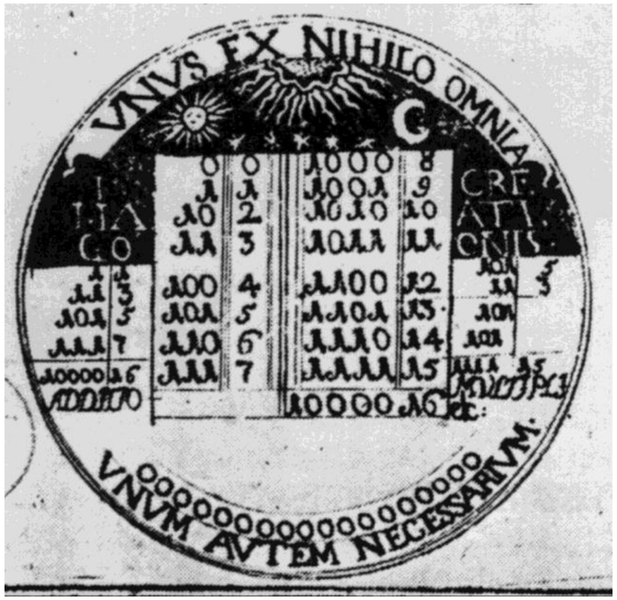


Historical note: Leibnitz (1646-1716)

- "The binary system may be used in place of the decimal system; express all numbers by unity and by nothing"
- 1679: built a mechanical calculator (+, -, *, /)



- CHALLENGE: "All who are occupied with the reading or writing of scientific literature have assuredly very often felt the want of a common scientific language, and regretted the great loss of time and trouble caused by the multiplicity of languages employed in scientific literature:
- SOLUTION: *"Characteristica Universalis"*: a universal, formal, language of reasoning
- The dream's end: Turing and Goedl in 1930's.



Leibniz's medallion for the Duke of Brunswick

Sequential Logic

Sequential VS combinational logic

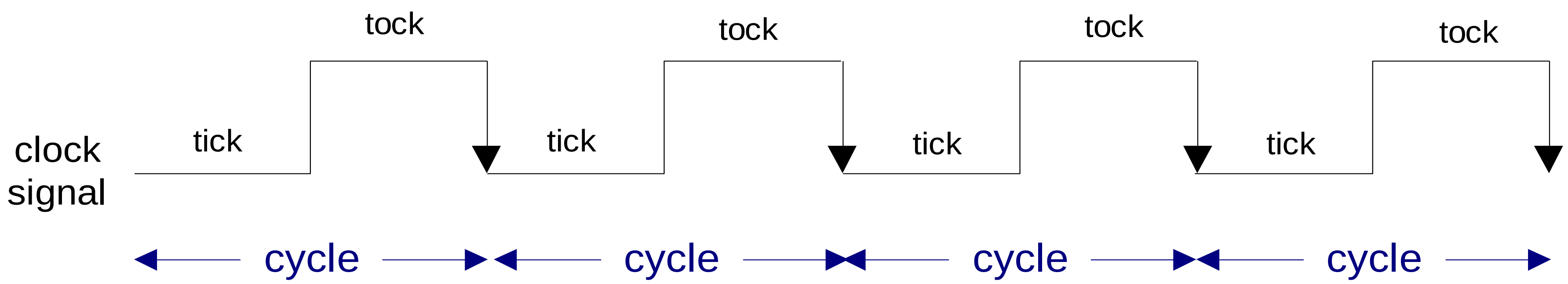
- Combinational devices: operate on data only;
  provide calculation services (e.g. Nand … ALU)
- Sequential devices: operate on data and a clock signal;
  as such, contain *state* and can ve made to provide storage and synchronization services
- Sequential devices are sometimes called “clocked devices”

Lecture plan

- Clock
- A hierarchy of memory chips:
  - Flip-flop gates
  - Binary cells
  - Registers
  - RAM
- Counters
- Perspective.

The Clock

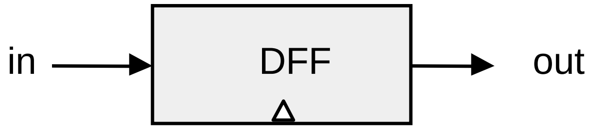
HW simulator demo



- In our jargon, a clock cycle = *tick*-phase (low), followed by a *tock*-phase (high)
- In real hardware, the clock is implemented by an oscillator
- In our hardware simulator, clock cycles can be simulated either
  - Manually, by the user, or
  - "Automatically," by a test script.

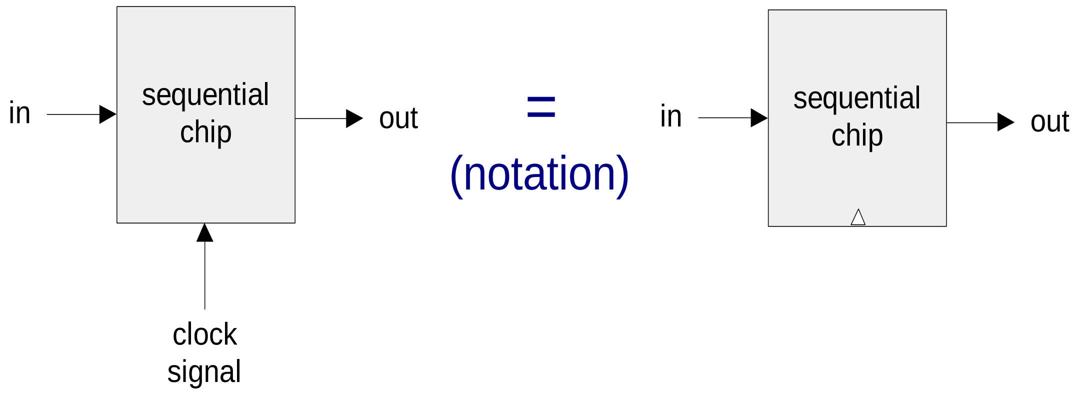
Flip-flop

HW simulator demo



out(t) = in(t-1)

- A fundamental state-keeping device
- For now, let us not worry about the DFF *implementation*
- Memory devices are made from numerous flip-flops, all regulated by the same master clock signal
- Notational convention:

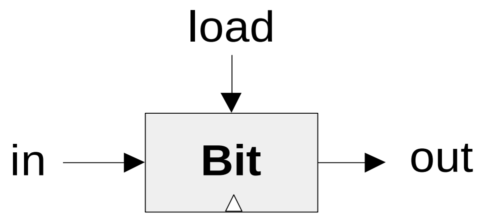


1-bit register (we call it "Bit")

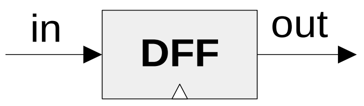
Objective: build a storage unit that can:

(a) Change its state to a given input

(b) Maintain its state over time (until changed)

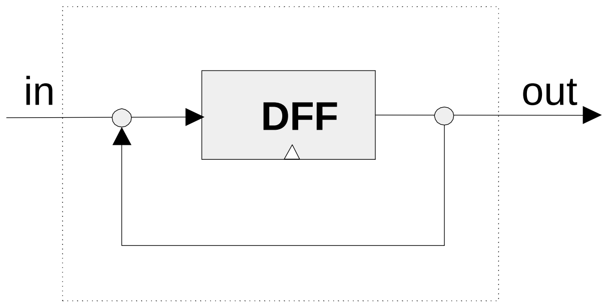


if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)



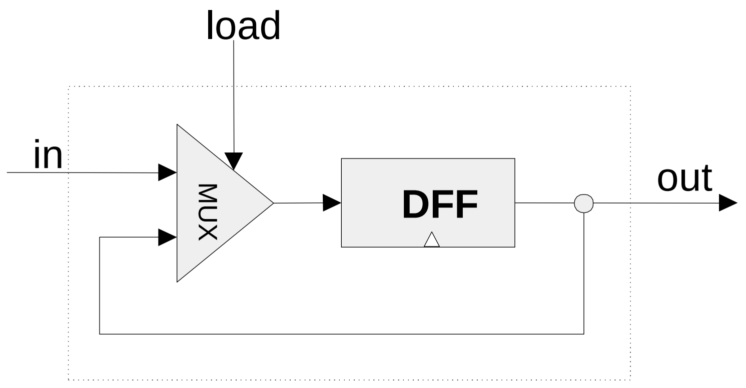
out(t) = in(t-1)

Basic building block



out(t) = out(t-1) ?
out(t) = in(t-1) ?

Won't work



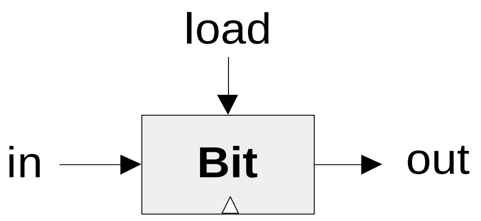
if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

OK

Bit (cont.)

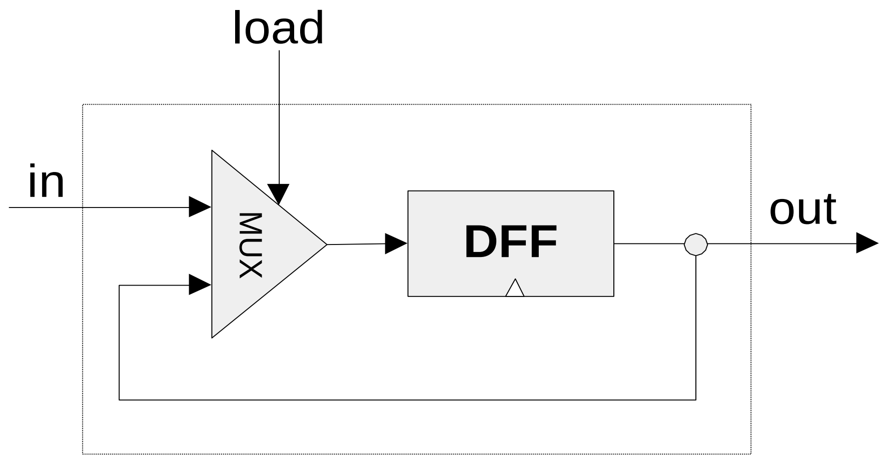
HW simulator demo

Interface



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

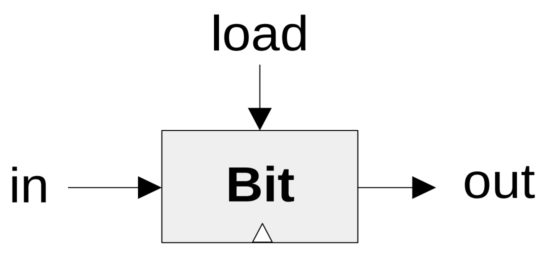
Implementation



- Load bit
- Read logic
- Write logic

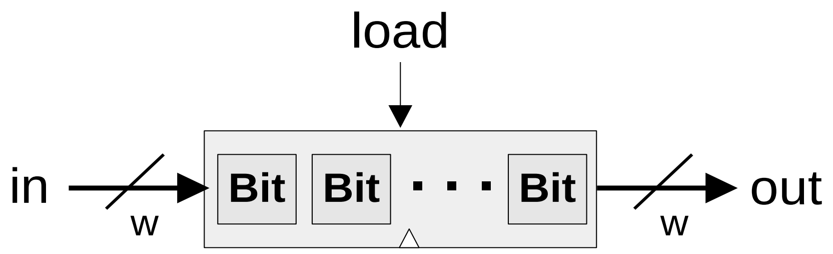
Multi-bit registers

HW simulator demo



if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

1-bit register



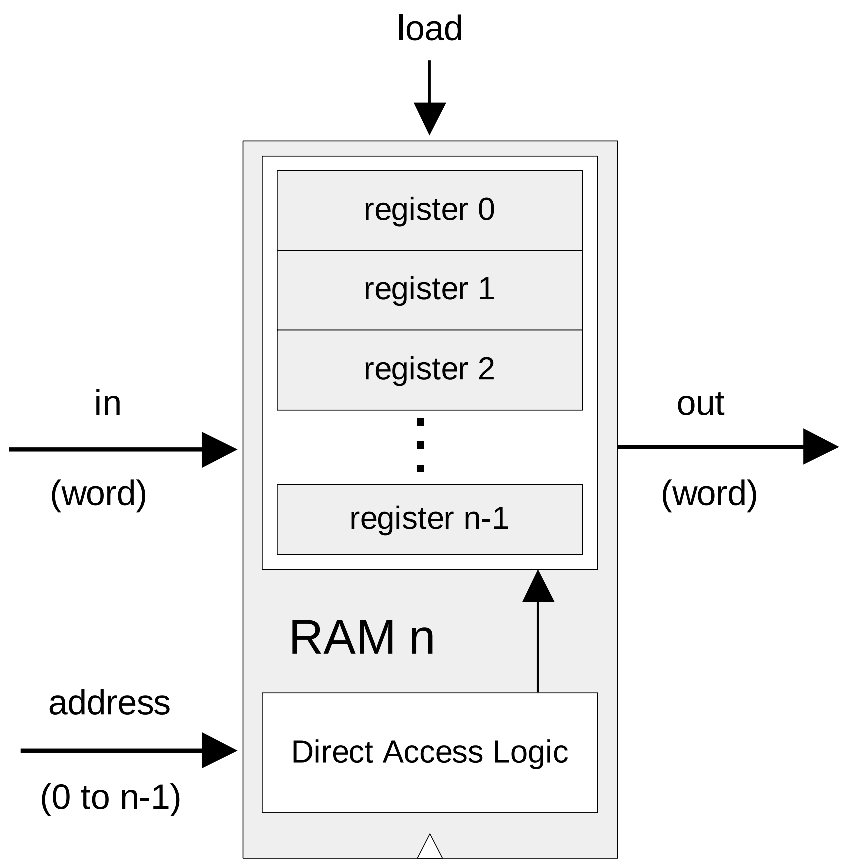
if load(t-1) then out(t)=in(t-1)
else out(t)=out(t-1)

w-bit register

- Register's width: a trivial parameter
- Read logic
- Write logic

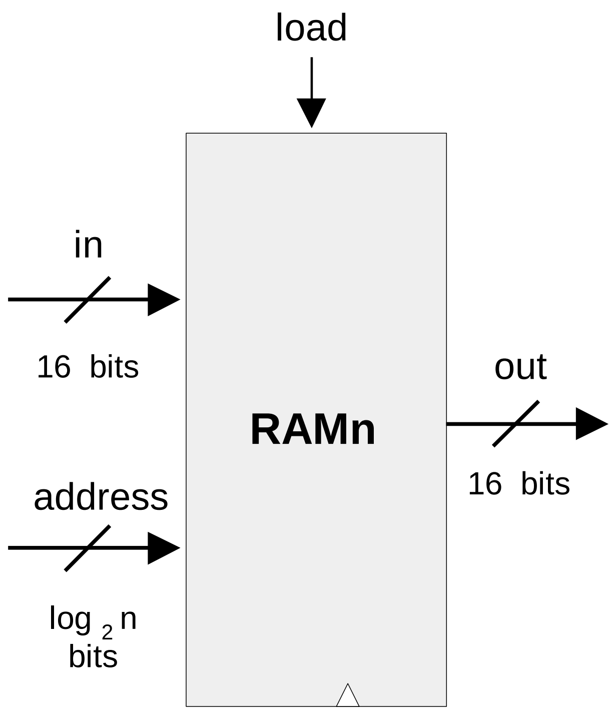
Random Access Memory (RAM)

HW simulator demo



- Read logic
- Write logic.

RAM interface

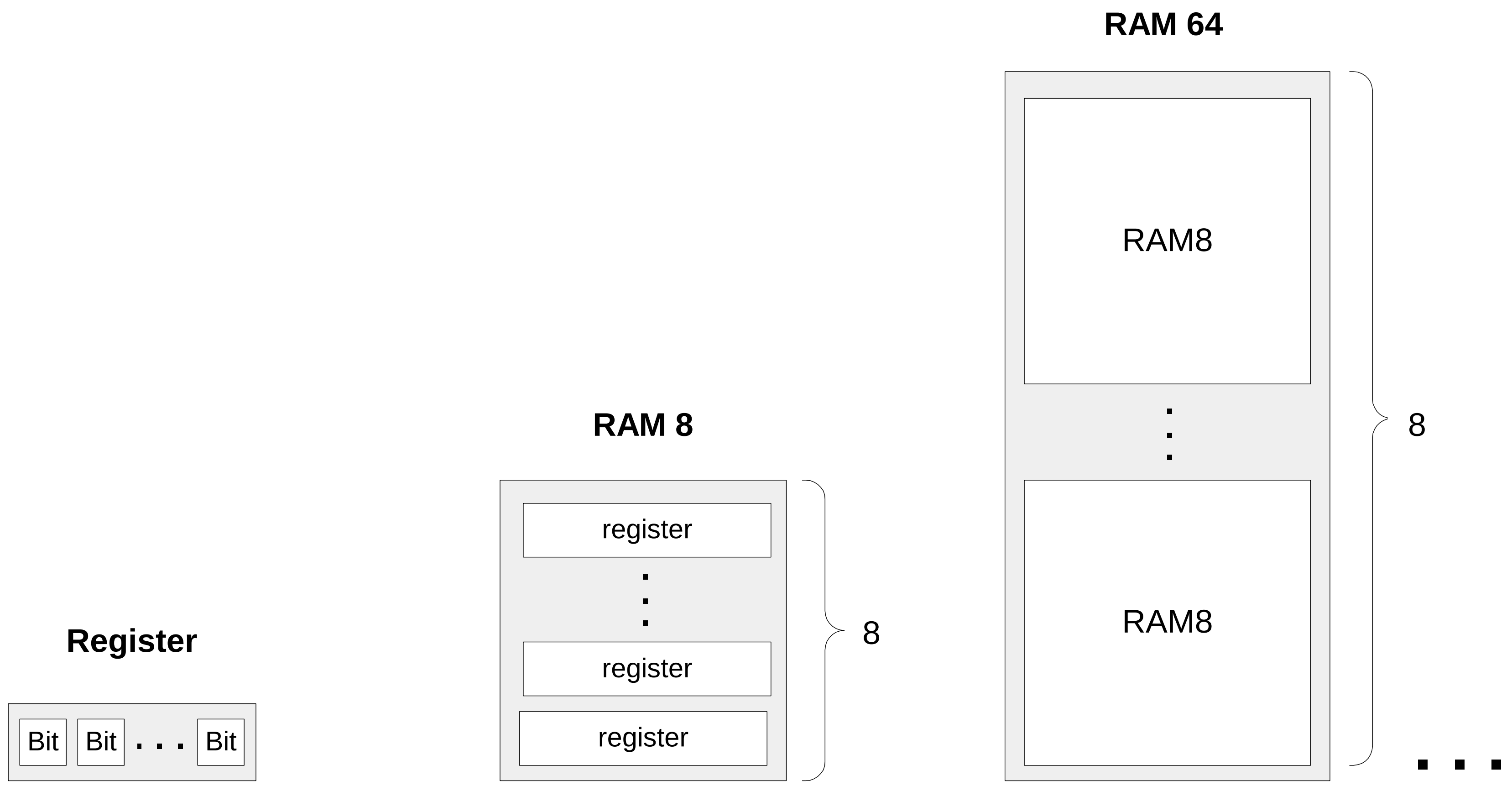


```
Chip name:   RAMn   // n and k are listed below
Inputs:      in[16], address[k], load
Outputs:     out[16]
Function:    out(t)=RAM[address(t)](t)
             If load(t-1) then
                RAM[address(t-1)](t)=in(t-1)
Comment:     "=" is a 16-bit operation.
```

The specific RAM chips needed for the Hack platform are:

Chip name	n	K
RAM8	8	3
RAM64	64	6
RAM512	512	9
RAM4K	4096	12
RAM16K	16384	14

RAM anatomy


RAM 64
RAM8
RAM8
8
RAM 8
register
register
register
8
Register
Bit
Bit
. . .
Bit
. . .

Counter

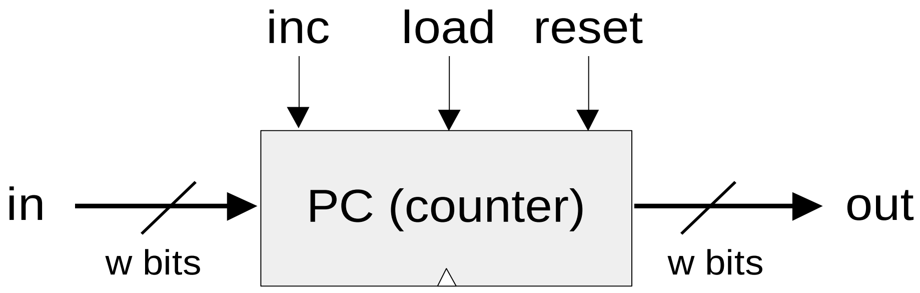
<u>Needed:</u> a storage device that can:

(a) set its state to some base value

(b) increment the state in every clock cycle

(c) maintain its state (stop incrementing) over clock cycles

(d) reset its state

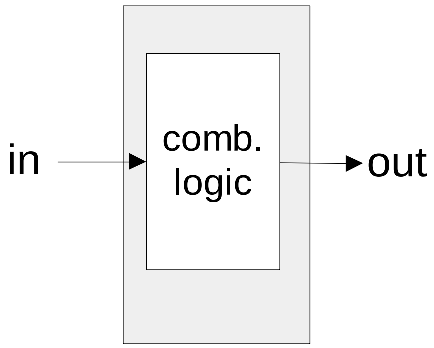


```
If reset(t-1) then out(t)=0
   else if load(t-1) then out(t)=in(t-1)
        else if inc(t-1) then out(t)=out(t-1)+1
             else out(t)=out(t-1)
```

- Typical function: *program counter*
- Implementation: register chip + some combinational logic.

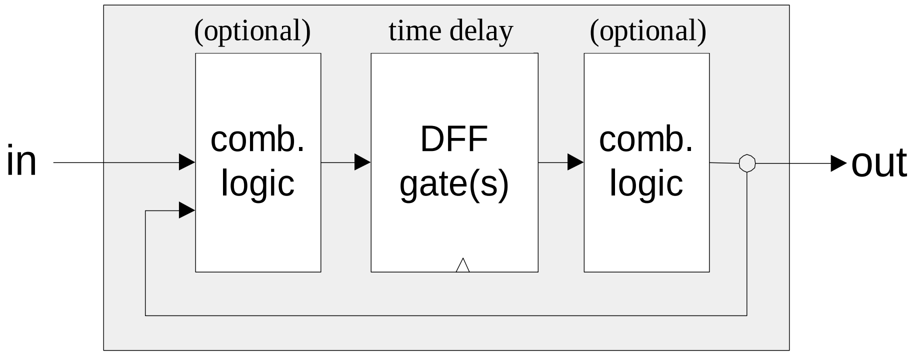
Recap: Sequential VS combinational logic

Combinational chip



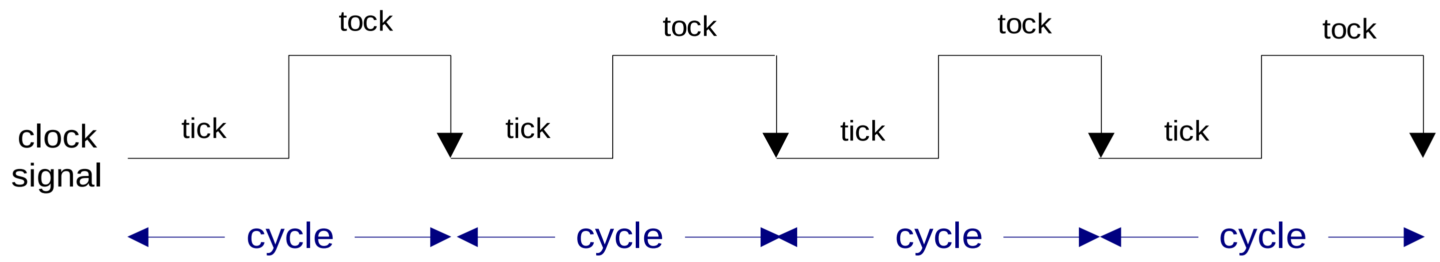
out = *some function of* (in)

Sequential chip

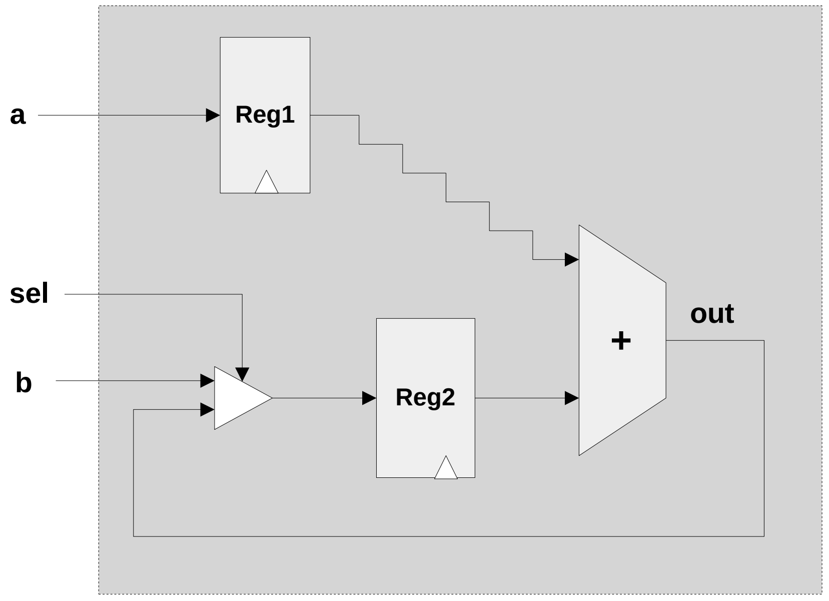


out(t) = *some function of* (in(t-1), out(t-1))

Time matters



- During a tick-tock cycle, the internal states of all the clocked chips are allowed to change, but their outputs are "latched"
- At the beginning of the next cycle, the outputs of all the clocked chips in the architecture commit to the new values.



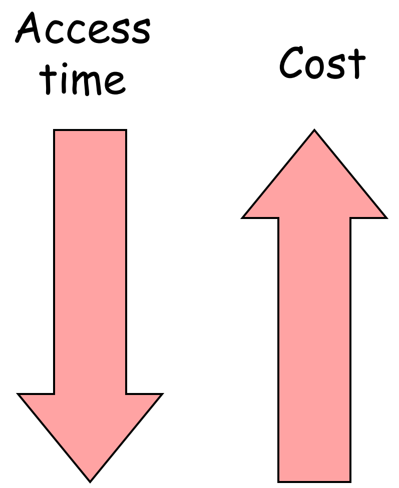
<u>Implications:</u>

- Challenge: propagation delays
- Solution: clock synchronization
- Cycle length and processing speed.

Perspective

- All the memory units described in this lecture are standard
- Typical memory hierarchy
  - SRAM ("static"), typically used for the cache
  - DRAM ("dynamic"), typically used for main memory
  - Disk

  (Elaborate caching / paging algorithms)
- A Flip-flop can be built from Nand gates
- But … real memory units are highly optimized, using a great variety of storage technologies.


Access time
Cost