representing HW

last time

RISC
- efficient, simple hardware
- expose what hardware can do well

CISC
- convenience of compiler/assembly programmer

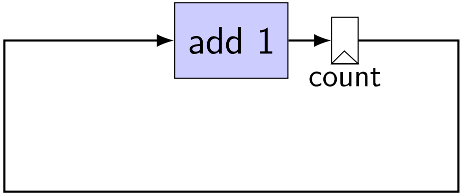
Y86-64
- movq $\rightarrow$ {irmovq, rrmovq, mrmovq, rmmovq}
- cmovXX
- one set of operand types per item

Y86-64 encoding table

describing hardware

how do we describe hardware?

pictures?



circuits with pictures?

yes, something you can do

such commercial tools exist, but…

not commonly used for processors

hardware description language

programming language for hardware

(typically) text-based representation of circuit

often abstracts away details like:

- how to build arithmetic operations from gates
- how to build registers from transistors
- how to build memories from transistors
- how to build MUXes from gates
- …

those details also not a topic in this course

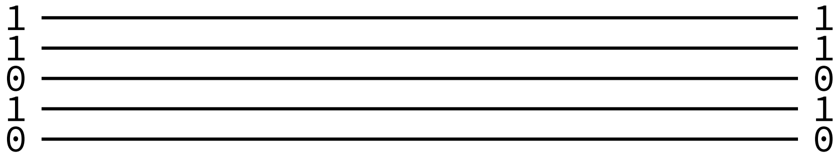
our tool: HCLRS

built for this course

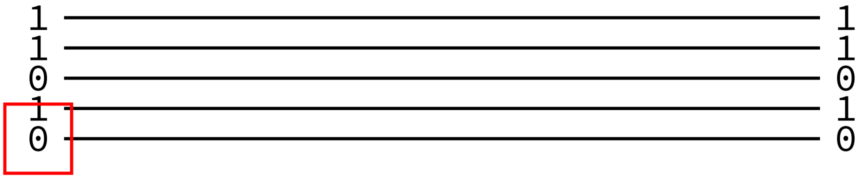
assumes you're making a processor

somewhat different from textbook's HCL

circuits: wires

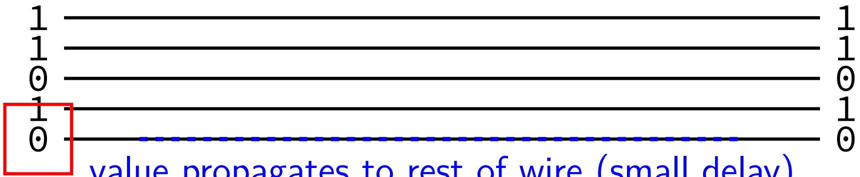


circuits: wires



binary value — actually voltage

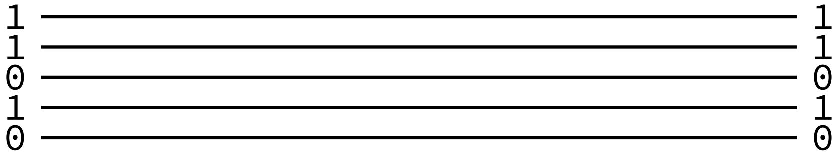
circuits: wires



value propagates to rest of wire (small delay)

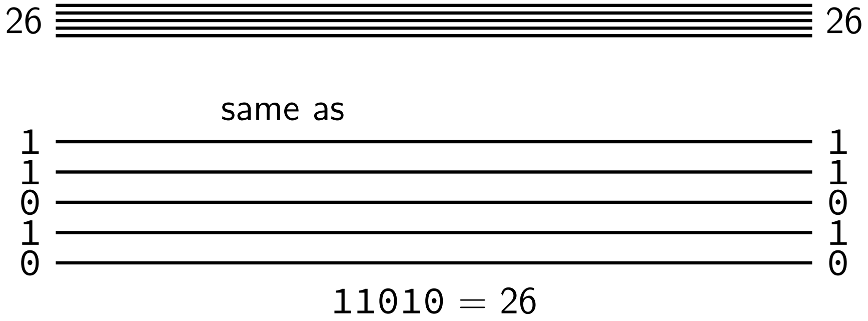
binary value — actually voltage

circuits: wire bundles

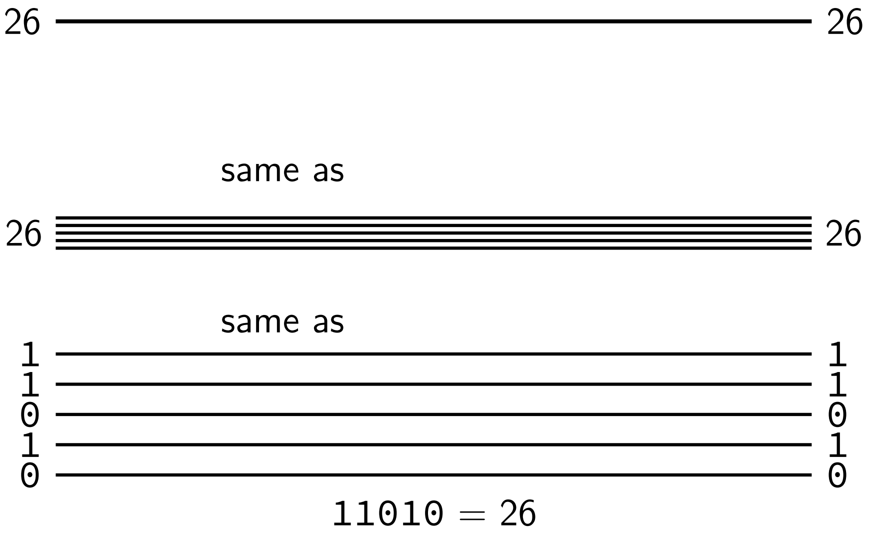


$\texttt{11010} = 26$

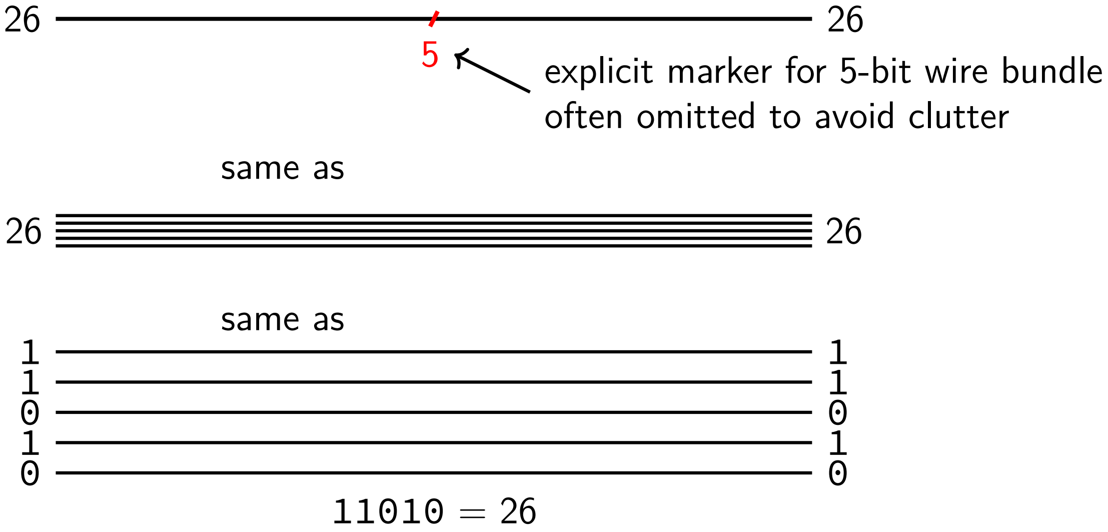
circuits: wire bundles



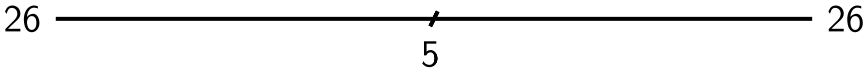
circuits: wire bundles



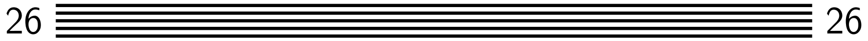
circuits: wire bundles


26
26
5
explicit marker for 5-bit wire bundle
often omitted to avoid clutter
same as
26
26
same as
1 1 0 1 0
1 1 0 1 0
11010 = 26

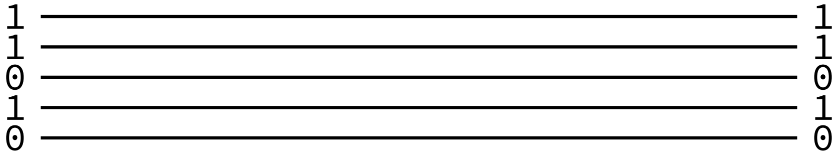
circuits: wire bundles



same as

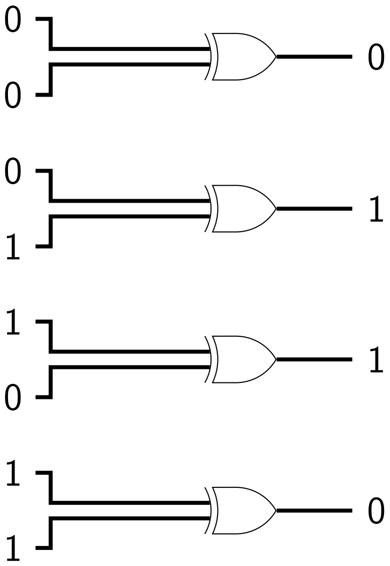


same as



`11010` $= 26$

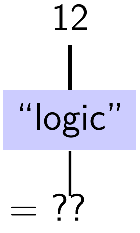
circuits: gates



circuits: logic

want to do calculations?

generalize gates:



function(12) = ??

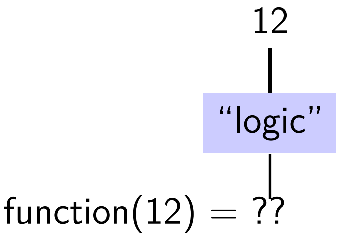
circuits: logic

want to do calculations?

generalize gates:

output wires contain result of function on input

changes as input changes (with delay)



circuits: logic

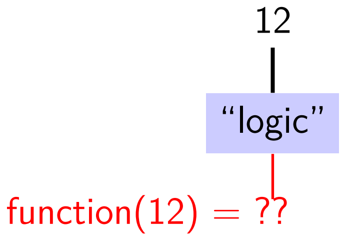
want to do calculations?

generalize gates:

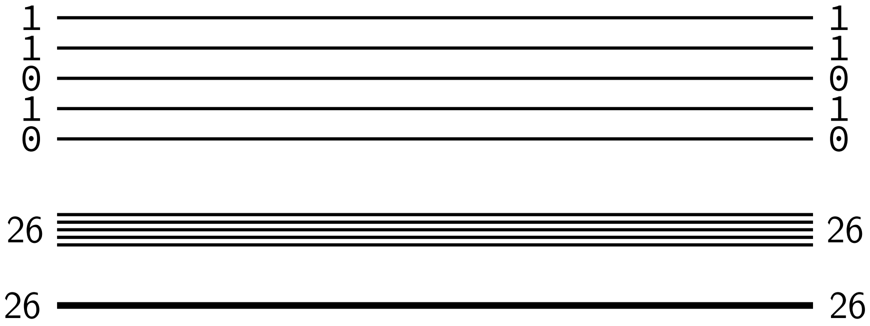
output wires contain result of function on input

changes as input changes (with delay)

need not be same width as output



HCLRS: wire (bundle)s

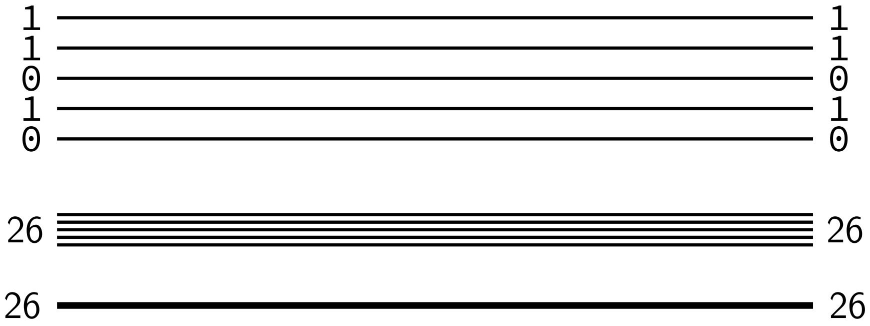


```
wire foo : 5; foo = 0b11010;
```
OR

```
wire foo : 5; foo = 26;
```
OR

```
wire foo : 5; foo = 0x1a;
```

HCLRS: wire (bundle)s



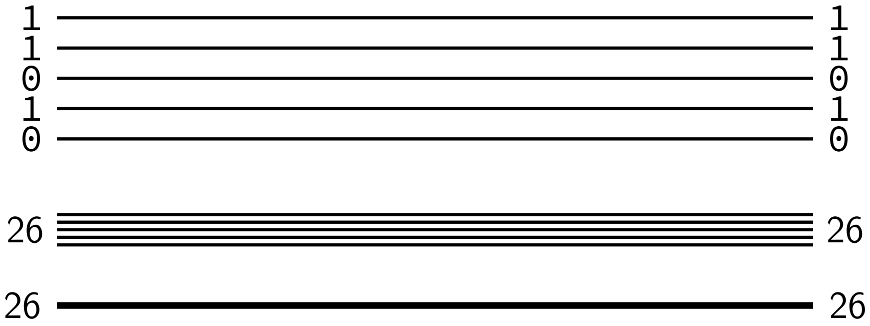
```
wire foo : 5; foo = 0b11010;
```
OR

```
wire foo : 5; foo = 26;
```
OR

```
wire foo : 5; foo = 0x1a;
```

name

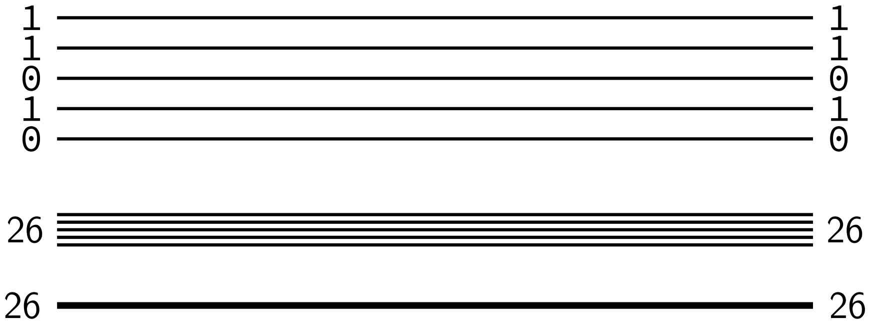
HCLRS: wire (bundle)s



```
wire foo : 5; foo = 0b11010;
```
OR
```
wire foo : 5; foo = 26;
```
OR
```
wire foo : 5; foo = 0x1a;
```

width (in bits)

HCLRS: wire (bundle)s



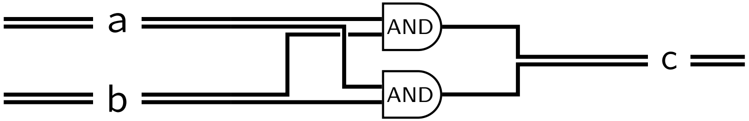
```
wire foo : 5; foo = 0b11010;    OR
wire foo : 5; foo = 26;         OR
wire foo : 5; foo = 0x1a;
```

assignment

indicates wire is *connected* to value

HCLRS: gates + calcuations (1)

```
wire a : 2; wire b : 2; wire c : 2;
c = b & a;
a = 0b10;
b = 0b11;
```



HCLRS: gates + calcuations (1)

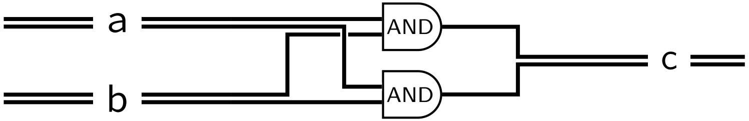
```
wire a : 2; wire b : 2; wire c : 2;
c = b & a;
a = 0b10;
b = 0b11;
```

same as

```
a = 0b10;
b = 0b11;
c = b & a;
```

order doesn't matter

connected or not

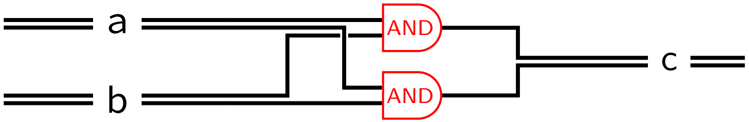


HCLRS: gates + calcuations (1)

```
wire a : 2; wire b : 2; wire c : 2;
c = b & a;
a = 0b10;
b = 0b11;
```

C-like expressions supported

`0b10 & 0b11 = 0b10`

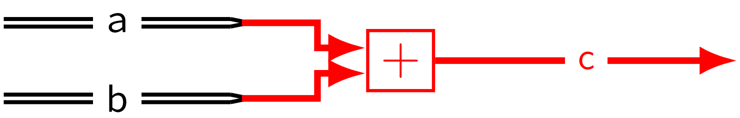


HCLRS: gates + calcuations (2)

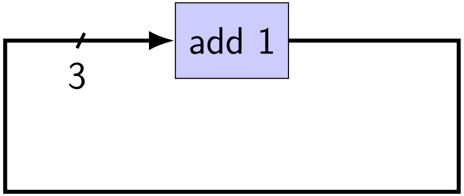
```
wire a : 2; wire b : 2; wire c : 2;
c = b + a; /* was ~~b & a~~ */
a = 0b10;
b = 0b11;
```

more than bitwise operators supported

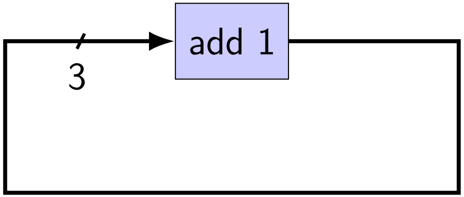
0b10 + 0b11 = 0b101 $\rightarrow$ 0b01 (extra bits lost)



example: (broken) counter circuit (1)

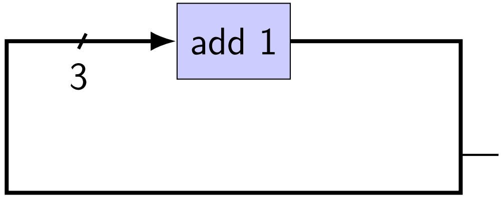


example: (broken) counter circuit (1)



```
wire x : 3;
x = x + 1;
```

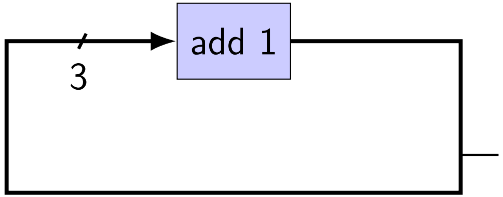
example: (broken) counter circuit (1)



time 0: `000`

```
wire x : 3;
x = x + 1;
```

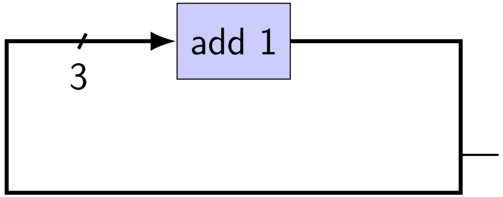
example: (broken) counter circuit (1)



time 0: `000` ← set how???

```
wire x : 3;
x = x + 1;
```

example: (broken) counter circuit (1)



time 0: 000

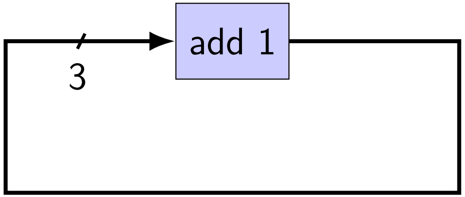
time 1: 001?

time 2: 010?

time 3: 011?

```
wire x : 3;
x = x + 1;
```

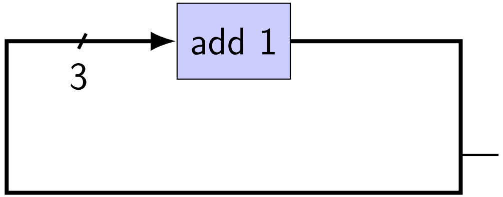
example: (broken) counter circuit (2)



```
~~wire x : 3;~~
~~x = x + 1;~~
```

HCLRS: compile error
"`Circular dependency detected:`
`x depends on x`"

example: (broken) counter circuit (3)



time 0: 000

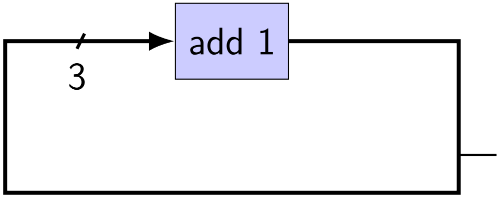
time 1: 001?

time 2: 010?

time 3: 011?

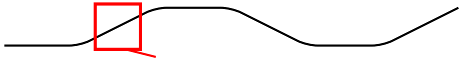
```
wire x : 3;
x = x + 1;
```

example: (broken) counter circuit (3)



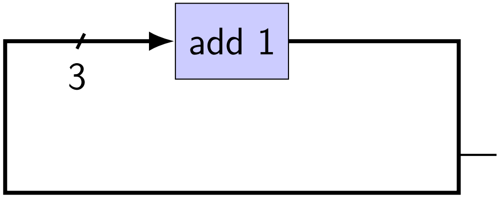
time 0: `000`
time 1: `001?`
time 2: `010?`
time 3: `011?`

```
wire x : 3;
x = x + 1;
```



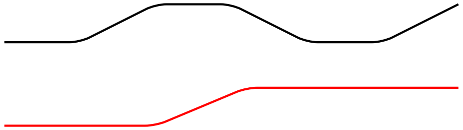
problem 1: how will “add 1” react to this value?
(not zero or one) ...

example: (broken) counter circuit (3)

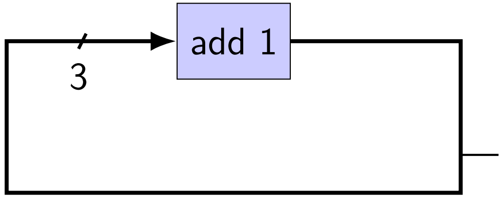


time 0: 000
time 1: 001?
time 2: 010?
time 3: 011?

```
wire x : 3;
x = x + 1;
```

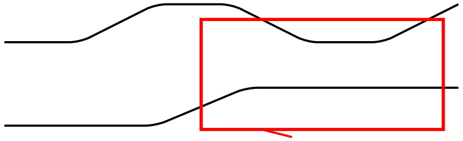


example: (broken) counter circuit (3)



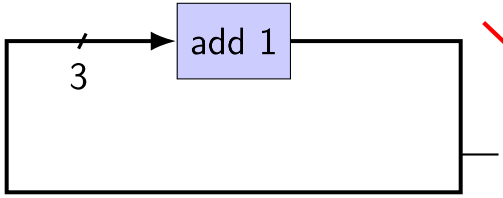
time 0: `000`
time 1: `001?`
time 2: `010?`
time 3: `011?`

```
wire x : 3;
x = x + 1;
```



problem 2: changes not in sync?

example: (broken) counter circuit (4)



~~time 0: 000~~
~~time 1: 001?~~
~~time 2: 010?~~
~~time 3: 011?~~

```
wire x : 3;
x = x + 1;
```

circuit is not stable
transient values during changes
hard to predict behavior

circuits: state

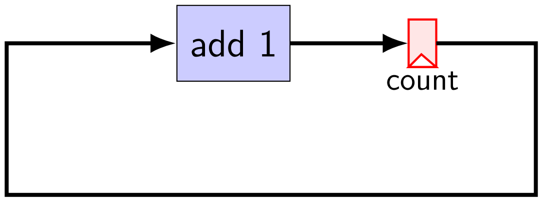
logic performs calculations all the time

never stores values!

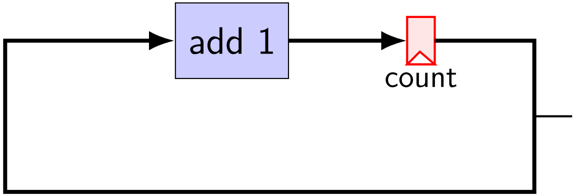
need extra elements to store values

- registers, memory

example: counter circuit (corrected)



example: counter circuit (corrected)



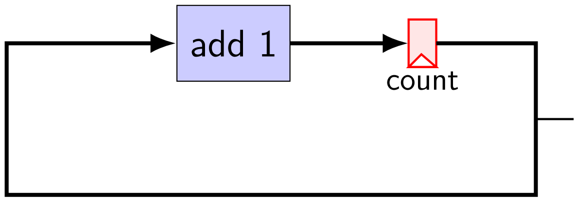
time 0: `000`

time 1: `001`

time 2: `010`

time 3: `011`

example: counter circuit (corrected)



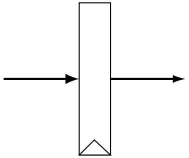
time 0: `000`
time 1: `001`
time 2: `010`
time 3: `011`

add register to store current count

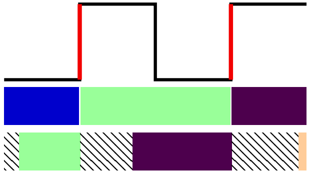
updates based on "clock signal" (not shown)

avoids intermediate updates

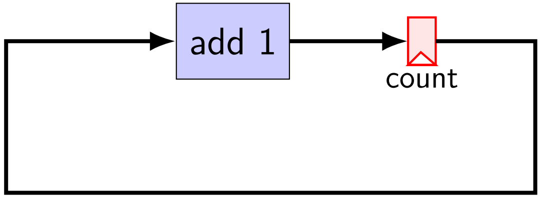
registers



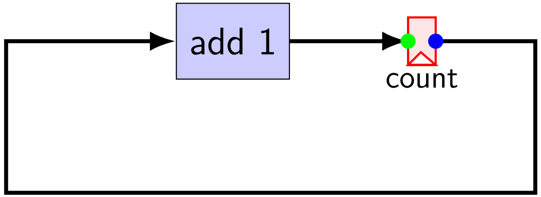
updates every clock cycle



example: counter circuit (real HCLRS)

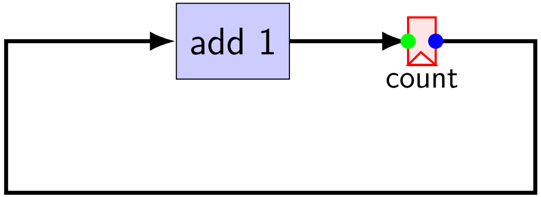


example: counter circuit (real HCLRS)



```
register xY {
     count : 3 = 0b000 ;
}
 x_count = Y_count + 0b001;
```

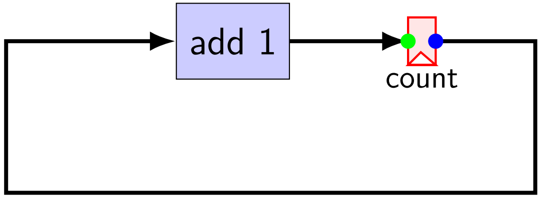
example: counter circuit (real HCLRS)



```
register  xY  {
      count  : 3 =  0b000 ;
}
 x_count  =  Y_count  + 0b001;
```

register “bank”
can have multiple (related) registers

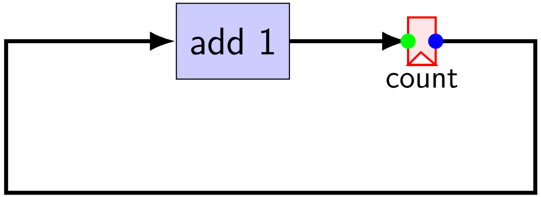
example: counter circuit (real HCLRS)



label for left/right side of registers
x: label for input side (always lowercase)
Y: label for output side (always uppercase)

```
register xY {
     count : 3 = 0b000 ;
}
 x_count = Y_count + 0b001;
```

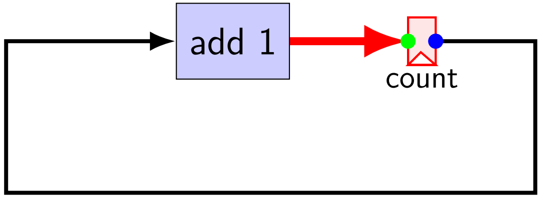
example: counter circuit (real HCLRS)



register "name"
input/output = *prefix*_name

```
register  xY  {
     count  : 3 =  0b000 ;
}
 x_count  =  Y_count  + 0b001;
```

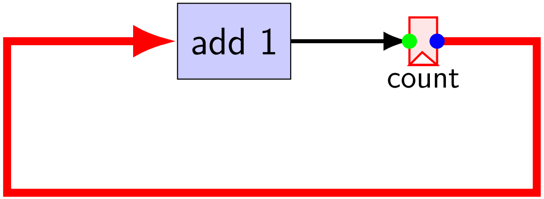
example: counter circuit (real HCLRS)



```
register xY {
     count : 3 = 0b000 ;
}
x_count = Y_count + 0b001;
```

input wire to register

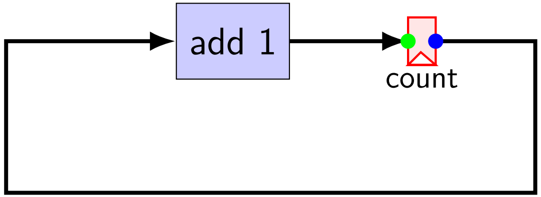
example: counter circuit (real HCLRS)



```
register xY {
    count : 3 = 0b000 ;
}
x_count = Y_count + 0b001;
```

output wire of register

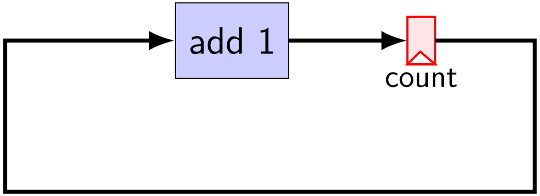
example: counter circuit (real HCLRS)



initial value of register
first value for output wire (Y_count)

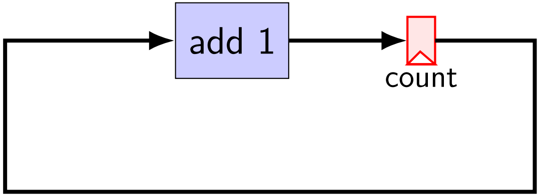
```
register xY {
     count : 3 = 0b000;
}
 x_count = Y_count + 0b001;
```

example: counter circuit



```
register  xY  {
        count  : 3 =  0b000 ;
}
 x_count  =  Y_count  + 0b001;
```

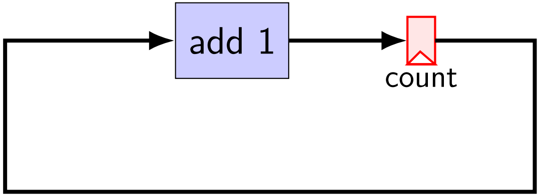
example: counter circuit



```
register  xY  {
        count  : 3 =  0b000 ;
}
 x_count  =  Y_count  + 0b001;
```

time	Y_count	x_count
start	000	001
start + 1 rising edge	001	010
start + 2 rising edges	010	011
start + 3 rising edges	011	100
…	…	…

example: counter circuit



```
register xY {
        count : 3 = 0b000 ;
}
 x_count = Y_count + 0b001;
```

time	Y_count	x_count
start	000	001
start + 1 rising edge	001	010
start + 2 rising edges	010	011
start + 3 rising edges	011	100
…	…	…

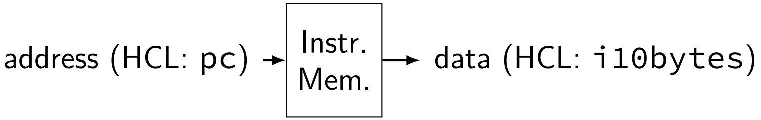
HCL circuit with registers

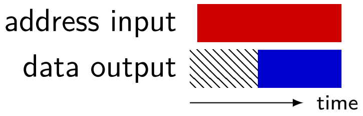
```
register xY {
    a : 4 = 1;  /* <-- initial Y_a */
    b : 4 = 1;  /* <-- initial Y_b */
}
x_b = x_a + Y_a;
x_a = Y_a + Y_b;
```

exercise: value of Y_a, Y_b after two rising edges of clock?

A. Y_a = 2, Y_b = 3
B. Y_a = 2, Y_b = 2
C. Y_a = 3, Y_b = 5
D. Y_a = 3, Y_b = 7
E. Y_a = 3, Y_b = 11
F. Y_a = 5, Y_b = 7
G. Y_a = 7, Y_b = 11
H. none of the above

instruction memory





Stat signal

how do we stop the simulated machine?

hard-wired mechanism — `Stat` wire

possible values:

- `STAT_AOK` — keep going
- `STAT_HLT` — stop, normal shtdown
- `STAT_INS` — invalid instruction
- …(and more errors)

(predefined 3-bit constants)

must be set

determines if simulator keeps going