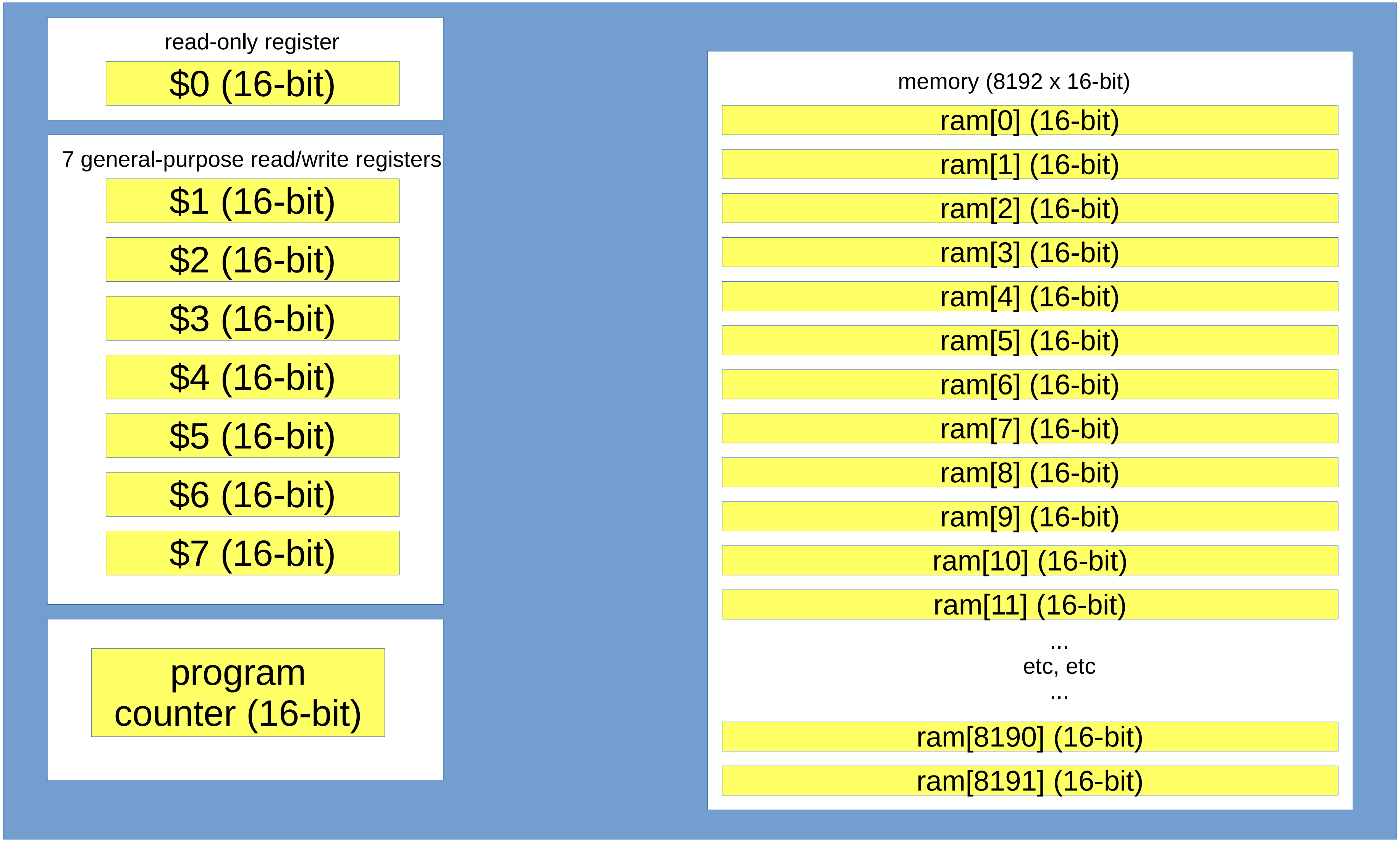
E20

What's inside the E20 processor?



Here's a sample of E20 assembly language.
Answer the questions based on your intuition.
How is this similar to E15 code?
How is this different from E15 code?

```
# Some simple math stuff

addi $1, $0, 5          # $1 := 5
addi $2, $1, -2         # $2 := $1 + (-2)
add $3, $1, $2          # $3 := $1 + $2

addi $4,$0, 55          # $4 := 55
sub $5, $4,$1           # $5 := $4 - $1
sub $4, $5, $4          # $4 := $5 - $4

or $6, $2, $5
and $7,$2, $5

halt                    # end the program
```

What does this program do?
What is the final value of each register?

Here's a sample of E20 assembly language.

How is this similar to E15 code?
How is this different from E15 code?

```
# A simple loop, counting down from 10

    movi $1,10          # Initialize counter to 10
beginning:
    jeq $1, $0,done      # if $1 == $0, go to done
    addi $1, $1,-1       # Decrement $1
    j beginning          # go to top of loop
done: halt               # we've finished
```

A *label* identifies a location in a program. Each label has a *name* and a *value*. The name is arbitrary. The value is the address of the memory where the label is defined.

Here, label **beginning** has value 1, and label **done** has value 4.

A label can be used as a destination of a jump, which is more convenient than giving the destination numerically.

n do?
of each register?

"The value of a label is the memory address where it is defined."

The E20 memory is an 8192-element array.

Each cell is 16-bits.

Each cell's value is initially zero, except for the program code, which is loaded starting at address 0 (not shown here).

Offset	Val
...	...
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
...	..

The E20 memory is an 8192-element array.

Each cell is 16-bits.

Each cell's value is initially zero, except for the program code, which is loaded starting at address 0 (not shown here).

Memory cells are changed with the **`sw`** instruction.

```
movi $1, 34
sw $1, 4($0)
```

Offset	Val
...	...
0	0
1	0
2	0
3	0
4	34
5	0
6	0
7	0
8	0
9	0
...	..

```
movi $1, 34
sw $1, 4($0)
```

The address to write to is
specified as the *sum*
of an immediate value
and a register.

In this case, we first
calculate the sum of 4 and $0.
That is the address to write to.

Offset	Val
...	...
0	0
1	0
2	0
3	0
4	34
5	0
6	0
7	0
8	0
9	0
...	..

Memory cells are read with the **`lw`** instruction. Given an address, they will copy the value of the memory cell into a register.

`lw $2, 4($0)`

After this, $2 will have value 34.

The address to read is specified as the *sum* of an immediate value and a register.

Offset	Val
...	...
0	0
1	0
2	0
3	0
4	34
5	0
6	0
7	0
8	0
9	0
...	..

Actually, I lied: the initial value of all memory cells is *not* zero. In reality, the program is loaded into memory, starting at address zero.

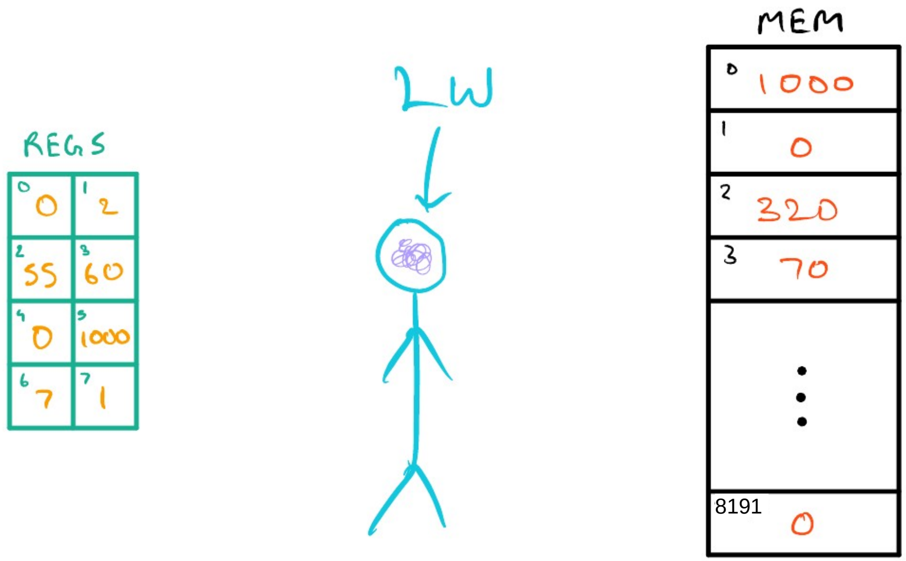
Consider this program:

Assembly code	Machine code (bin)	Machine code (dec)
`lw $2, 4($0)`	`1000000100000100`	`33028`
`halt`	`0100000000000001`	`16385`

Offset	Val
...	...
0	`33028`
1	`16385`
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
...	..

ILLUSTRATING LW

lw $3, 3($0)

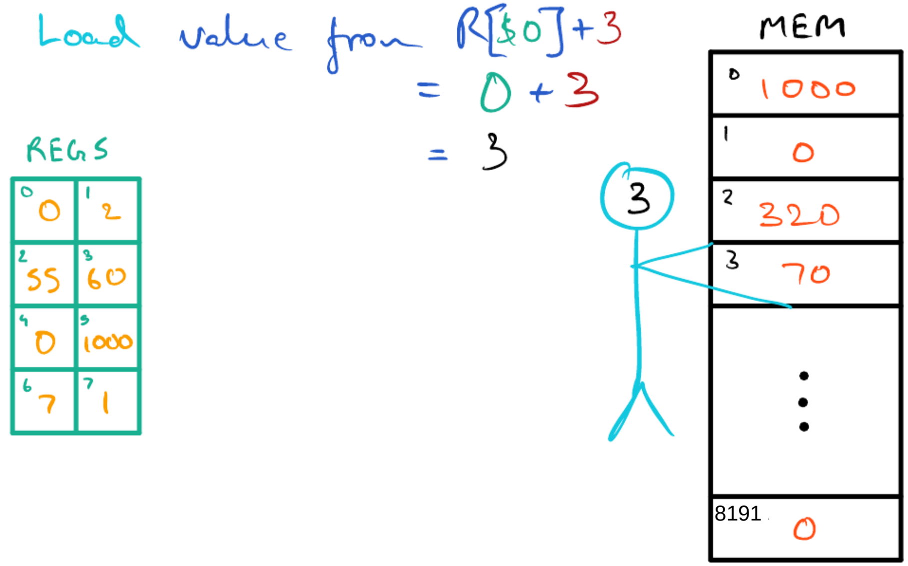


lw $3, 3($0)

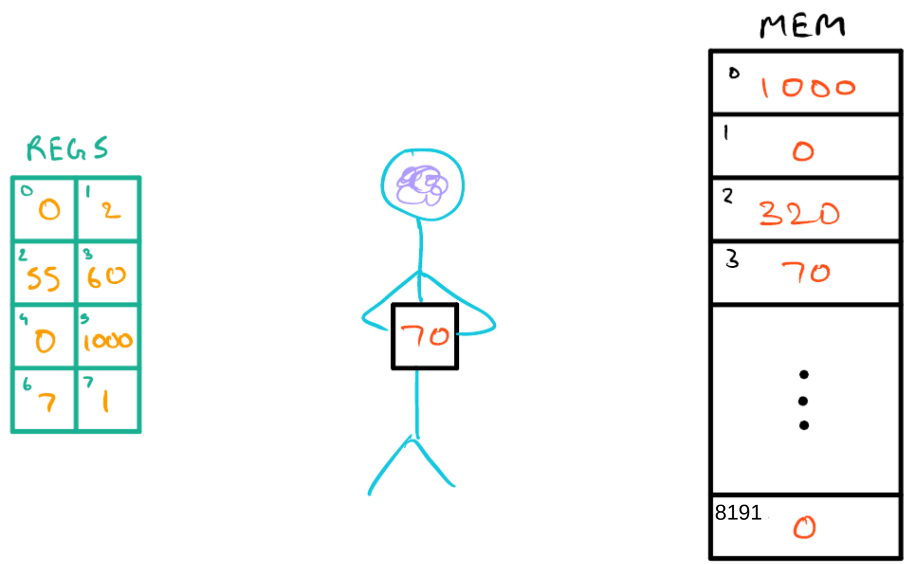
Load value from R[$0]+3

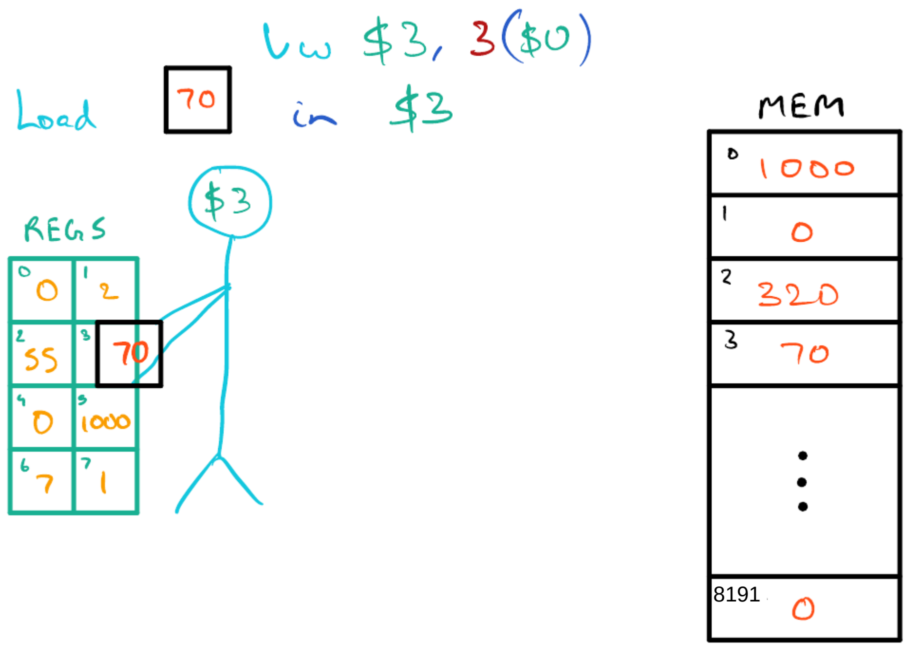
= 0 + 3

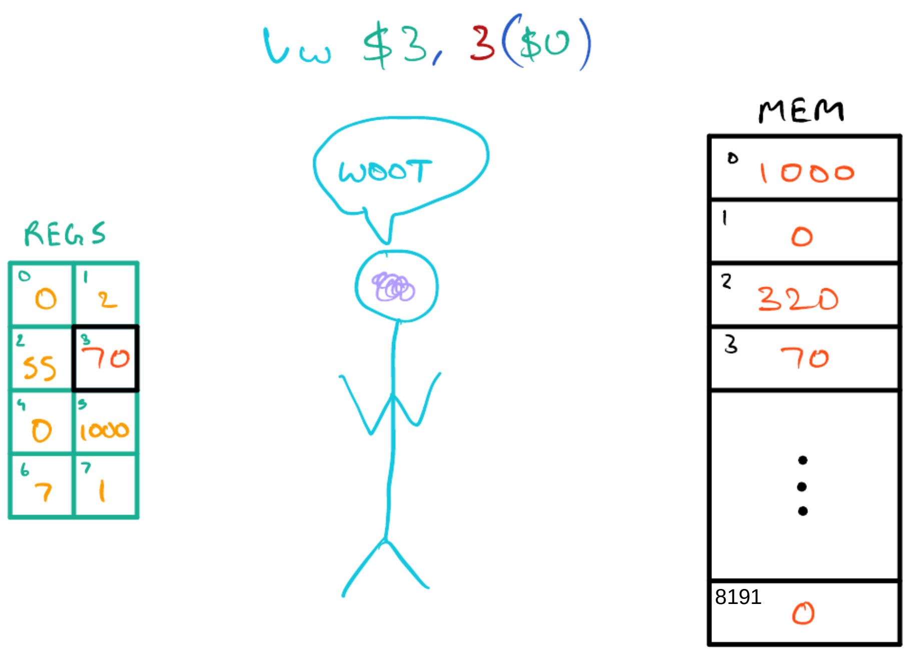
= 3



lw $3, 3($0)

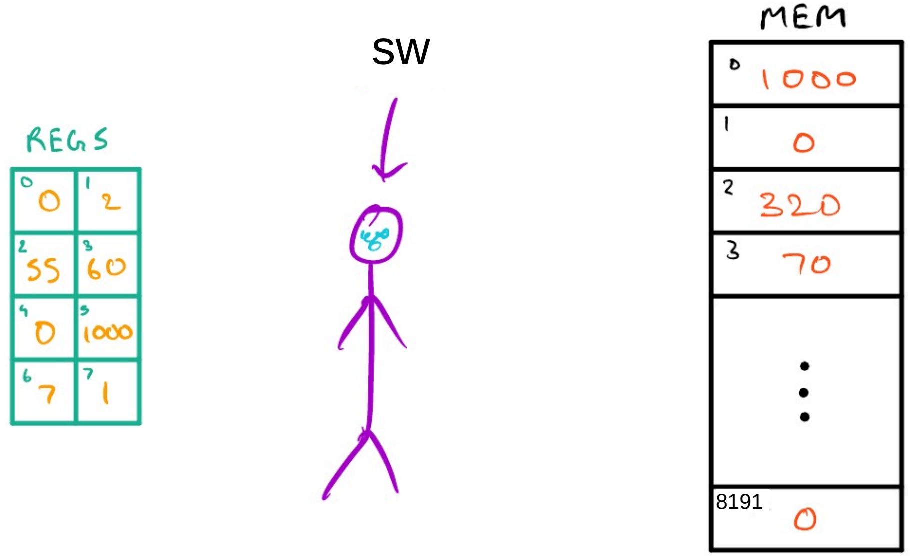



lw $3, 3($0)
Load 70 in $3
$3
REGS
0 0
1 2
2 55
3 70
4 0
5 1000
6 7
7 1
MEM
0 1000
1 0
2 320
3 70
8191 0


lw $3, 3($0)
WOOT
REGS
0 0
1 2
2 55
3 70
4 0
5 1000
6 7
7 1
MEM
0 1000
1 0
2 320
3 70
8191 0

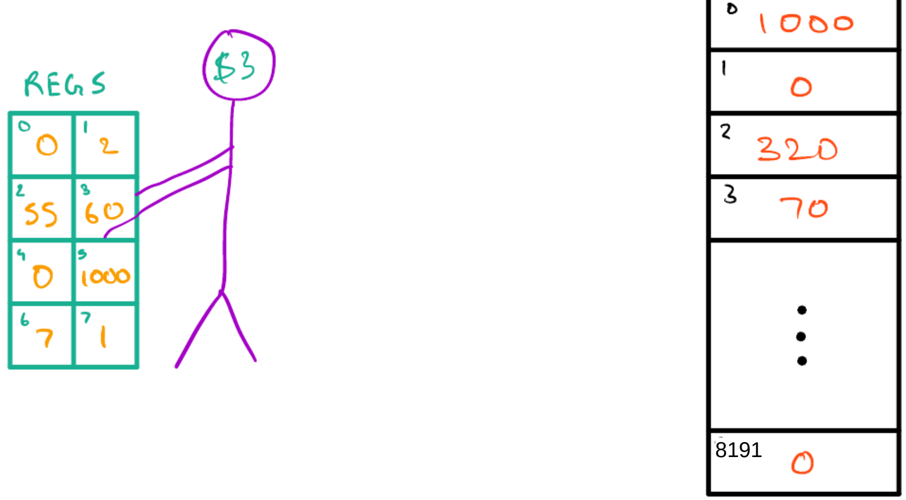
ILLUSTRATING SW

sw $3, 3($0)

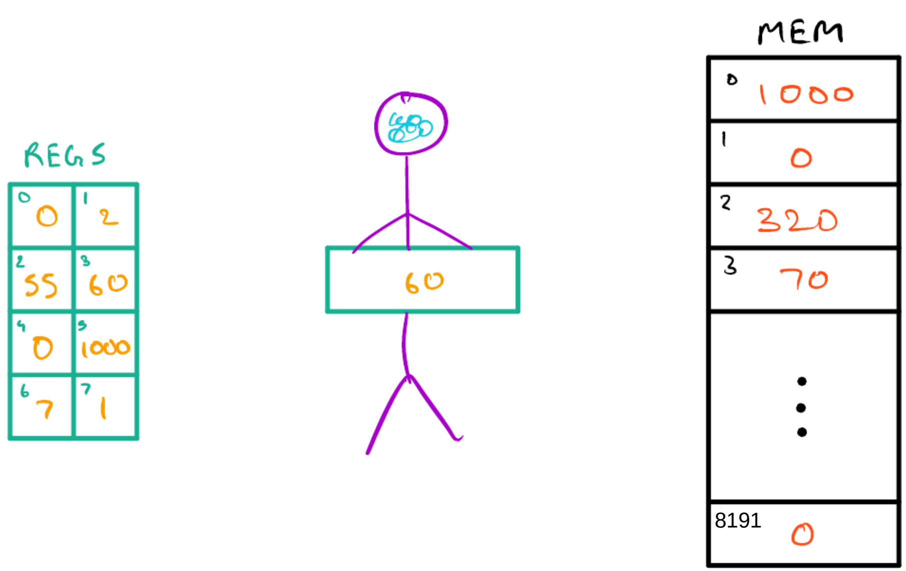


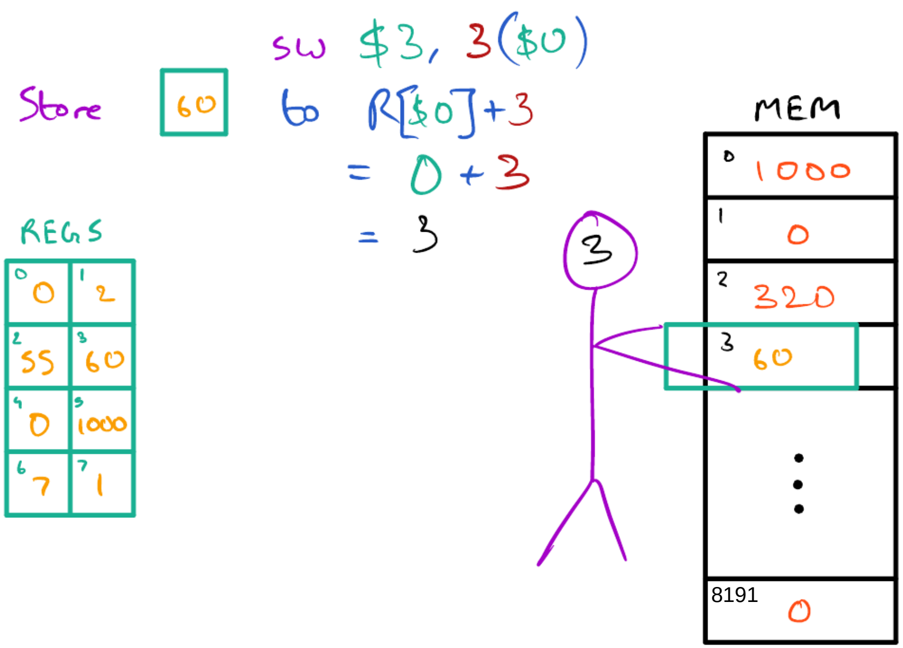
sw $3, 3($0)

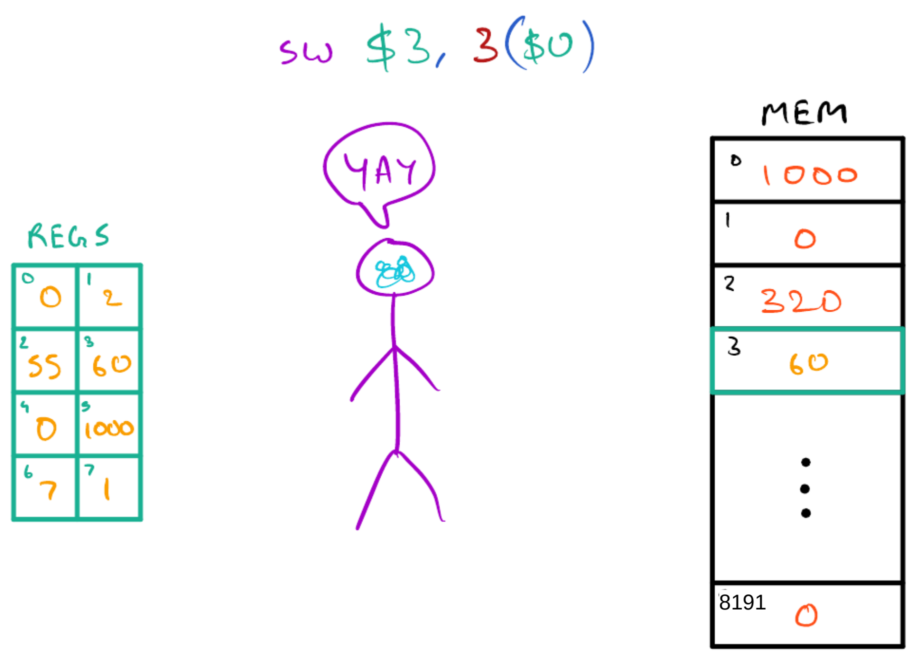
Store value from $3



sw $3, 3($0)




sw $3, 3($0)
Store 60 to R[$0]+3
= 0 + 3
= 3
REGS
0 0
1 2
2 55
3 60
4 0
5 1000
6 7
7 1
MEM
0 1000
1 0
2 320
3 60
8191 0
3


sw $3, 3($0)
YAY
REGS
0 0
1 2
2 55
3 60
4 0
5 1000
6 7
7 1
MEM
0 1000
1 0
2 320
3 60
8191 0

Here's a sample of E20 assembly language.

Notice in particular the use of **lw** and **sw** to access memory.

```
    lw $1,var1($0)     # read from address var1 + 0
    lw $2,var2($0)     # read from address var2 + 0
    and $3, $1, $2     # AND the values together
    or $4, $1, $2      # then OR them together
    sw $3,var3($0)     # write the AND result into memory
    movi $5, var3      # put the address (not the value!) in $5
    addi $6, $5, var4

    halt                # program ends

var1:
    .fill 30

var2:
    .fill 5

var3:
    .fill 0

var4:
```

A label can be used anywhere an immediate is accepted, including with opcodes **sw**, **lw**, and **movi.**

Here, we read from the memory location identified by label **var1** using **lw**.

What does this program do?
What is the final value of each register?

7 general-purpose read/write registers

$1 = 3

$2 = 0

$3 = 0

$4 = 0

$5 = 0

$6 = 0

$7 = 0

```
addi $1, $1, -5
```

?


```
addi $1, $1, -5
```

?

Crash?
Random number?
-2?
0?
Other?

7 general-purpose read/write registers

$1 = 3

$2 = 0

$3 = 0

$4 = 0

$5 = 0

$6 = 0

$7 = 0

```
addi $1, $1, -5
```

?

Decimal	Binary (2's complement)
3	`0000000000000011`
-5	`1111111111111011`

7 general-purpose read/write registers

$1 = 3

$2 = 0

$3 = 0

$4 = 0

$5 = 0

$6 = 0

$7 = 0

```
addi $1, $1, -5
```

?

Decimal	Binary (2's complement)
3	`0000000000000011`
-5	`1111111111111011`
-2	`1111111111111110`

7 general-purpose read/write registers

$1 = 3

$2 = 0

$3 = 0

$4 = 0

$5 = 0

$6 = 0

$7 = 0

```
addi $1, $1, -5
```

This is the correct result, based on the binary arithmetic that we learned earlier.
But how should your simulator display it?

?

	Binary (2's complement)
	0000000000000011
-5	1111111111111011
-2	1111111111111110

`addi $1, $1, -5`

?

Decimal (unsigned)	Decimal (signed)	Binary (2's complement)
3	3	`0000000000000011`
65531	-5	`1111111111111011`
65534	-2	`1111111111111110`

7 general-purpose read/write registers

$1 = 3

$2 = 0

$3 = 0

$4 = 0

$5 = 0

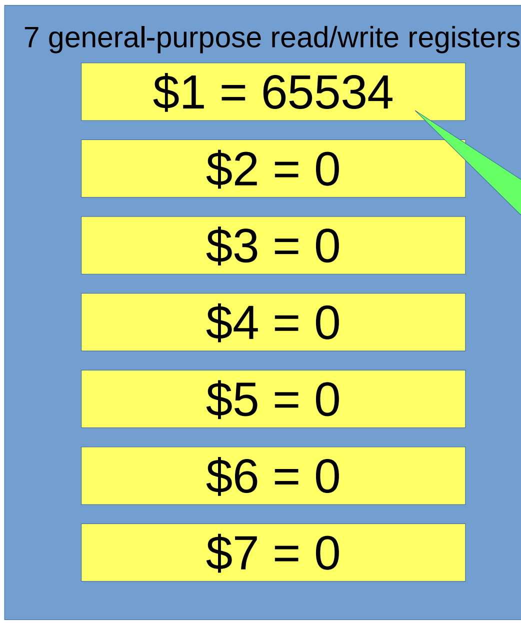
$6 = 0

`addi $1, $1, -5`

The E20 registers store 16-bit values.
Your simulator should always display those values are unsigned decimal integers.
So this is what we should see!

Decimal (unsigned)	Decim[illegible])	Binary (2's complement)
3	[illegible]	`0000000000000011`
65531	-5	`1111111111111011`
65534	-2	`1111111111111110`

`addi $1, $1, -5`



This is the "wraparound" effect. Binary arithmetic means that small numbers "wrap" to high numbers and vice versa.

Dec	
3	`0000000000000011`
-5	`1111111111111011`
-2	`1111111111111110`

Wraparound happen in any language that uses fixed-width integers.
C and C++ do this. Run this program and see for yourself.

```cpp
#include <iostream>
#include <cstdint>
using namespace std;
int main() {

    // uint16_t is a 16-bit unsigned int type
    uint16_t x = 3;
    x = x - 5;
    cout << "3-5=" << x << endl; // prints 65534

    uint16_t y = 60000;
    y = y + 7000;
    cout << "60000+7000=" << y << endl; // prints 1464


    return 0;
}
```

E20 Recursion

What is a function?

- A function needs to be know where it was called from, and go back there when it's done
  - use **`jal`** and **`jr`**
  - return address stored in \$7
- A function needs to accept parameters and return a result
  - we can decide to use registers
  - parameters in \$1, \$2, \$3, ...
  - return value in \$1

```
    movi $1, 1
    jal quadruple
    add $2, $0, $1   # save result

    movi $1, 5
    jal quadruple

    add $4, $2, $1    # $4 = 4*1 + 4*5
    halt

quadruple:
    add $1, $1, $1
    add $1, $1, $1
    jr $7
```

What is a function?

- But what happens when a function calls a function?
  - Saved return address and parameters will be lost

What is a function?

- But what happens when a function calls a function?
  - Saved return address and parameters will be lost

What is a function?

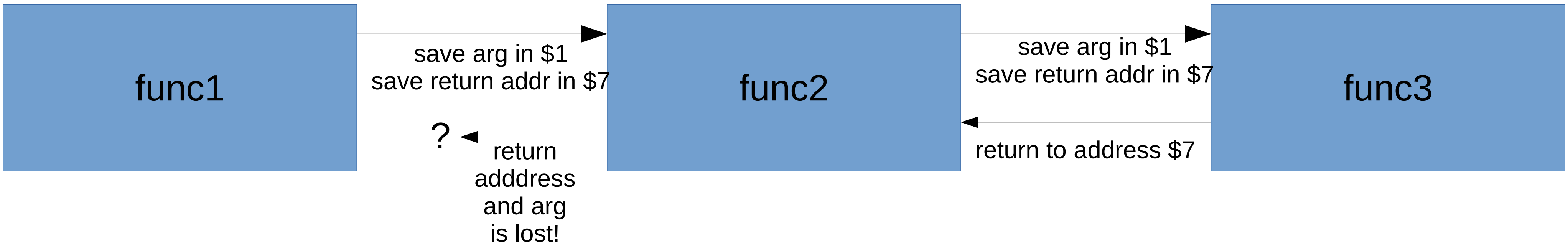
- But what happens when a function calls a function?
  - Saved return address and parameters will be lost

What is a function?

- But what happens when a function calls a function?
  - Saved return address and parameters will be lost

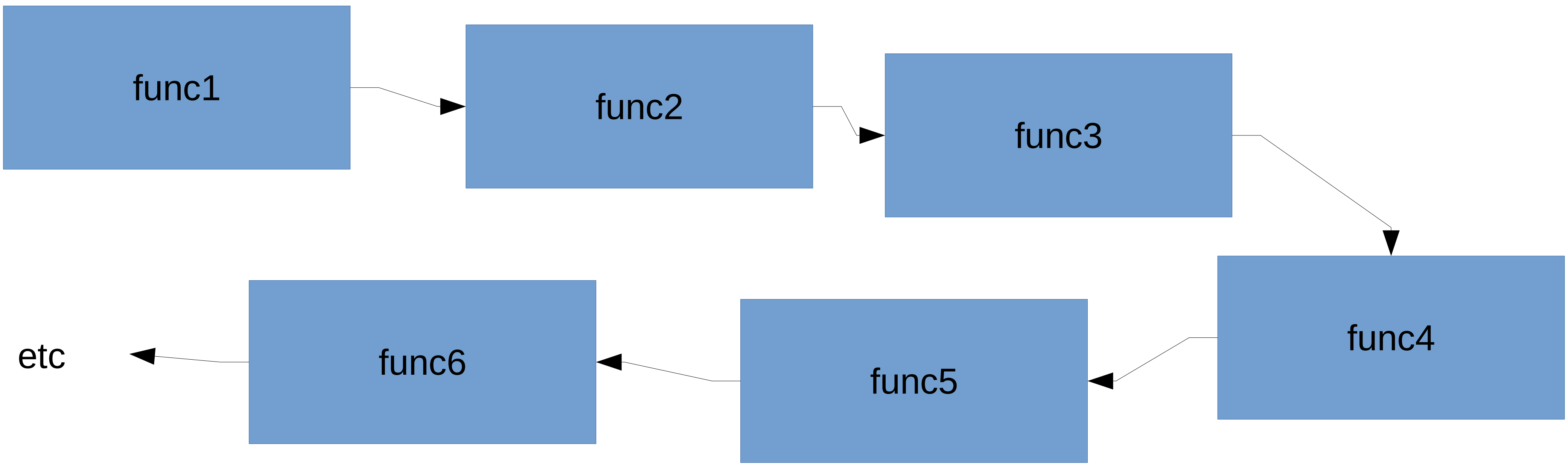
What is a function?

- But what happens when a function calls a function?
  - Saved return address and parameters will be lost



What is a function?

- But what happens when a function calls a function?
  - Saved return address and parameters will be lost
  - The chain of function can be arbitrarily long
    - Especially if recursion!



fib(3)

stack:

Address	Value
50	
51	
52	
53	
54	
55	
56	
57	

top of stack

fib(3)

save argument on stack

stack:

Address	Value
50	3
51	
52	
53	
54	
55	
56	
57	

top of stack

fib(3)

save argument on stack

save return address on stack

stack:

Address	Value
50	3
51	32
52	
53	
54	
55	
56	
57	

top of stack

fib(3)

save argument on sta

save return address o

Note: return address is at topofstack-1; and argument is at topofstack-2

ck:

Address	Value
50	3
51	32
52	
53	
54	
55	
56	
57	

top of stack

fib(3)

save argument on stack

save return address on stack

call fib(2)

stack:

Address	Value
50	3
51	32
52	
53	
54	
55	
56	
57	

top of stack

fib(3)

    save argument on stack

    save return address on stack

    call fib(2)

        save argument on stack

stack:

Address	Value
50	3
51	32
52	2
53	
54	
55	
56	
57	

top of stack

fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack

stack:

Address	Value	
50	3	
51	32	
52	2	
53	39	
54		top of stack
55		
56		
57		

fib(3)
save argument on stack
save return address on stack
call fib(2)
save argument on stack
save return address on stack
call fib(1)
save argument on stack
save return address on stack

stack:

Address	Value
50	3
51	32
52	2
53	39
54	1
55	3
56	
57	

top of stack (56)

fib(3)
save argument on stack
save return address on stack
call fib(2)
save argument on s
save return address
call fib(1)
save argument on
save return add ess on stack
base case!

It's a base case, so we don't recurse.

stack:

Address	Value	
50	3	
51	32	
52	2	
53	39	
54	1	
55	3	
56		top of stack
57		

fib(3)

    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address

`stack:`

Address	Value
50	3
51	32
52	2
53	39
54	
55	
56	
57	

top of stack

fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address
        save result from fib(1) on stack

stack:

Address	Value
50	3
51	32
52	2
53	39
54	1
55	
56	
57	

top of stack

fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address
        save result from fib(1) on stack
        call fib(0)
            [same as above]

stack:

Address	Value
50	3
51	32
52	2
53	39
54	1
55	1
56	
57	

top of stack (address 56)

fib(3)
  save argument on stack
  save return address on stack
  call fib(2)
    save argument on stack
    save return address on stack
    call fib(1)
      save argument on stack
      save return address on stack
      base case!
      remove return address from stack
      remove argument from stack
      jump to saved return address
    save result from fib(1) on stack
    call fib(0)
      [same as above]
    remove saved results from stack, sum

stack:

Address	Value
50	3
51	32
52	2
53	39
54	
55	
56	
57	

top of stack

fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address
        save result from fib(1) on stack
        call fib(0)
            [same as above]
        remove saved results from stack, sum
        remove return address from stack
        remove argument from stack
        jump to saved return address

stack:

Address	Value	
50	3	
51	32	
52		top of stack
53		
54		
55		
56		
57		

fib(3)
- save argument on stack
- save return address on stack
- call fib(2)
  - save argument on stack
  - save return address on stack
  - call fib(1)
    - save argument on stack
    - save return address on stack
    - base case!
    - remove return address from stack
    - remove argument from stack
    - jump to saved return address
  - save result from fib(1) on stack
  - call fib(0)
    - [same as above]
  - remove saved results from stack, sum
  - remove return address from stack
  - remove argument from stack
  - jump to saved return address
- save result from fib(2) on stack

stack:

Address	Value
50	3
51	32
52	2
53	
54	
55	
56	
57	

top of stack

fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address
        save result from fib(1) on stack
        call fib(0)
            [same as above]
        remove saved results from stack, sum
        remove return address from stack
        remove argument from stack
        jump to saved return address
    save result from fib(2) on stack
    call fib(1)
        [same as above]

stack:

Address	Value
50	3
51	32
52	2
53	1
54	
55	
56	
57	

top of stack

fib(3)
- save argument on stack
- save return address on stack
- call fib(2)
  - save argument on stack
  - save return address on stack
  - call fib(1)
    - save argument on stack
    - save return address on stack
    - base case!
    - remove return address from stack
    - remove argument from stack
    - jump to saved return address
  - save result from fib(1) on stack
  - call fib(0)
    - [same as above]
  - remove saved results from stack, sum
  - remove return address from stack
  - remove argument from stack
  - jump to saved return address
- save result from fib(2) on stack
- call fib(1)
  - [same as above]
- removed save results from stack, sum

stack:

Address	Value	
50	3	
51	32	
52	2	
53	1	
54		top of stack
55		
56		
57		

```
fib(3)
    save argument on stack
    save return address on stack
    call fib(2)
        save argument on stack
        save return address on stack
        call fib(1)
            save argument on stack
            save return address on stack
            base case!
            remove return address from stack
            remove argument from stack
            jump to saved return address
        save result from fib(1) on stack
        call fib(0)
            [same as above]
        remove saved results from stack, sum
        remove return address from stack
        remove argument from stack
        jump to saved return address
    save result from fib(2) on stack
    call fib(1)
        [same as above]
    removed save results from stack, sum
    remove return address from stack
    remove argument from stack
    jump to saved return address
```

	Address	Value	
stack:	50		top of stack
	51		
	52		
	53		
	54		
	55		
	56		
	57		