



CS 11 C track: lecture 8

- Last week: hash tables, C preprocessor
- This week:
 - Other integral types: `short`, `long`, `unsigned`
 - bitwise operators
 - `switch`
 - "fun" assignment: virtual machine



Integral types (1)

- Usually use `int` to represent integers
- But many other integral (integer-like) types exist:
 - `short`
 - `long`
 - `char`
 - `unsigned int`
 - `unsigned short`
 - `unsigned long`
 - `unsigned char`



Integral types (2)

- Two basic things that can vary:
 - **unsigned** vs. signed (default)
 - length: **char**, **short**, **int**, **long**
- Note that **char** is an integral type
 - can always treat char as an 8-bit integer
- Two basic questions:
 - Why use **unsigned** types?
 - When should we use shorter/longer integral types?



Integral types (2)

- Why use **unsigned** types?
 - may be used for something that can't be negative
 - e.g. a length
 - gives you 2x the range due to last bit
 - may want to use it as an array of bits
 - so sign is irrelevant
 - C has lots of bitwise operators



Integral types (3)

- When should we use shorter/longer integral types?
 - to save space when we know range is limited
 - when we know the exact number of bits we need
- `char` always 8 bits
- `short` usually 16 bits
- `int` usually 32 bits (but sometimes 64)
- `long` usually 32 bits (but sometimes 64)
- guaranteed: $\text{length}(\text{char}) < \text{length}(\text{short}) \leq \text{length}(\text{int}) \leq \text{length}(\text{long})$



Integral types (4)

- `unsigned` by itself means unsigned `int`
- Similarly it's legal to say
 - `short int`
 - `unsigned short int`
 - `long int`
 - `unsigned long int`
- but usually we shorten by leaving off the `int`



Bitwise operators (1)

- You don't need to know this for this lab!
- But a well-rounded C programmer should know this anyway...
- There are several "bitwise operators" that do logical operations on integral types bit-by-bit
 - OR (`|`) (note difference from logical or: `||`)
 - AND (`&`) (note difference from logical and: `&&`)
 - XOR (`^`)
 - NOT (`~`) (note difference from logical not: `!`)



Bitwise operators (2)

- bitwise OR (`|`) and AND (`&`) work bit-by-bit
- `01110001 | 10101010 = ?`
 - `11111011`
- `01110001 & 10101010 = ?`
 - `00100000`
- NOTE: They don't do short-circuit evaluation like logical OR (`||`) and AND (`&&`) do
 - because that wouldn't make sense



Bitwise operators (3)

- bitwise XOR (^) also works bit-by-bit
- $01110001 \wedge 10101010 = ?$
 - 11011011
- Bit is set if one of the operand's bits is 1 and the other is 0 (not both 1s or both 0s)



Bitwise operators (4)

- bitwise NOT (\sim) also works bit-by-bit
- $\sim 10101010 = ?$
 - 01010101 (duh)
- Substitute 0 for 1 and 1 for 0



Bitwise operators (5)

- Two other bitwise operators:
 - bitwise left shift (\ll)
 - bitwise right shift (\gg)
- $00001111 \ll 2 = ?$
 - 00111100
- $00111100 \gg 2 = ?$
 - 00001111
- Can use to multiply/divide by powers of 2



switch (1)

- Minor language feature: `switch`
- Used to choose from multiple integer-valued possibilities
- Cleaner than a series of `if/else if/else` statements



switch (2)

- Common coding pattern:

```
void do_stuff(int i) {  
    if (i == 0) {  
        printf("zero\n");  
    } else if (i == 1) {  
        printf("one\n");  
    } else {  
        printf("something else\n");  
    }  
}
```



switch (3)

```
void do_stuff(int i) {  
    switch (i) {  
        case 0:  
            printf("zero\n");  
            break;  
        case 1:  
            printf("one\n");  
            break;  
        default:  
            printf("something else\n");  
            break;  
    }  
}
```



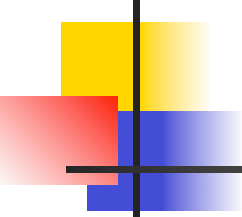
switch (4)

- `switch` statements more convenient than `if/else if/else` for many integer-valued cases
 - but not as general -- can only be used on integral types (`int`, `char`, etc.)
- Lab 8 code contains one `switch` statement that you don't have to write
 - but you should understand it anyway



switch (5)

```
switch (i) {  
    case 0: /* Start here if i == 0 */  
        printf("zero\n");  
        break; /* Exit switch here. */  
    ... /* other cases: 1, 2, 42 etc. */  
    default: /* if no case matches i */  
        printf("no match\n");  
        break;  
}
```



switch (6) -- fallthrough

```
switch (i) {  
    case 0: /* Start here if i == 0 */  
        printf("zero\n");  
        /* oops, forgot the break */  
    case 1: /* "fall through" from case 0 */  
        printf("one\n");  
        break;  
}
```

- Now, if `i` is `0` then prints "zero" and also "one"!
- Sometimes this is desired, but usually just a bug



Lab 8: Virtual machine (1)

- Where have you heard the term "virtual machine" before?
 - Java virtual machine
- A "virtual microprocessor"
- You define simple instructions for a mythical computer's assembly language
- Program interprets them



Virtual machine (2)

- Our virtual machine is very simple
- Only data type will be `int`
- All instructions will act on `ints`
- Instructions include
 - arithmetic
 - control flow
 - memory access
 - printing



Virtual machine (3)

- First need to define data structures for our virtual microprocessor:
 - **instruction memory** to hold instructions of program
 - **registers** to hold temporary results of computations
 - **stack** to hold results that are being operated on directly



Virtual machine (4)

- Instruction memory contains 2^{16} locations
 - = 65536
- Each location is a single byte (**unsigned char**)
- How many bits do we need to represent all possible locations in instruction memory?
 - 16
- Can use an **unsigned short** for this
 - Called the "instruction pointer" or IP
- Don't confuse with C's pointers! Not the same thing!
 - It's just an index into the instruction memory



Virtual machine (5)

- 16 **registers** (temporary storage locations)
- How many bits do we need to represent all possible locations in registers?
 - 4
- Can use an **unsigned char** for this
- Registers are just an array of 16 **ints**



Virtual machine (6)

- **Stack** which is 256 deep
- How many bits do we need to represent all possible locations in stack?
 - 8
- Can use an **unsigned char** for this
 - called the "**stack pointer**" or **SP**
 - also not a pointer in the C sense, just an index
- Stack is just an array of 256 **ints**

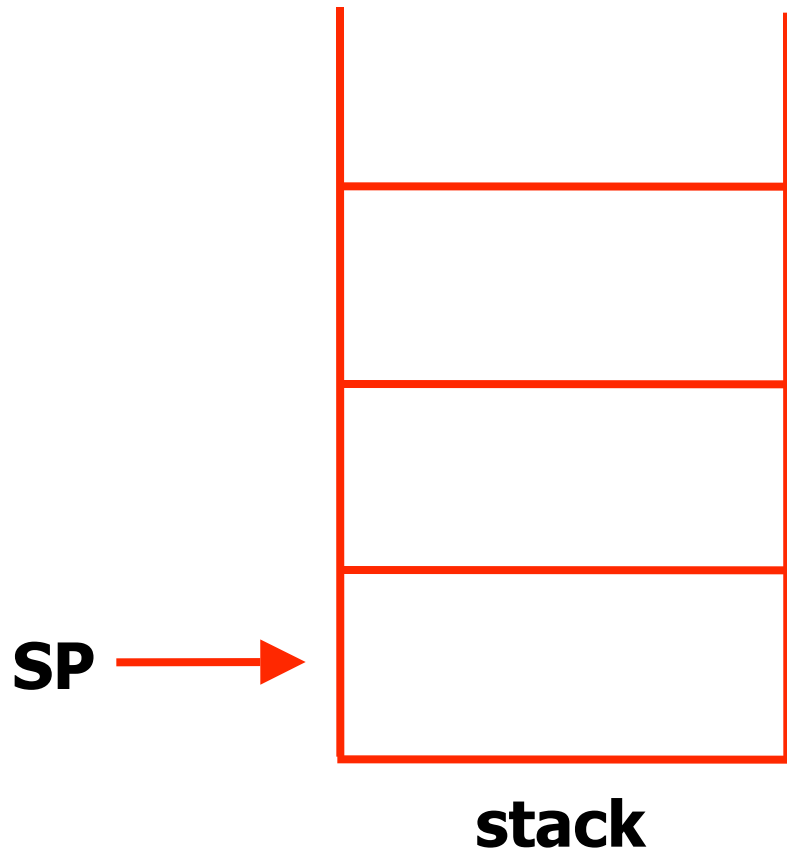


Push and pop (1)

- Stack has two operations: **push** and **pop**
- **push** puts a new value onto the stack
- **pop** removes a value from the stack
- Have to adjust stack pointer (SP) after push and pop
- Stack pointer "points to" first UNUSED element of stack
 - starts at zero for empty stack
- Top filled element in stack is "top of stack" (**TOS**)



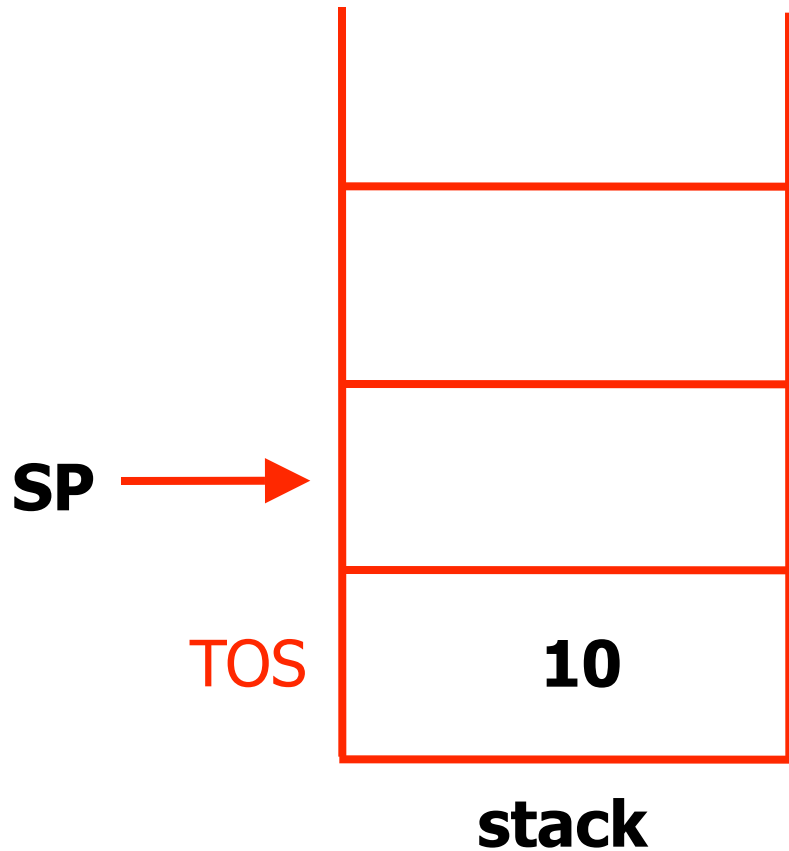
Push and pop (2)



Stack starts off empty;
SP points to first unused location



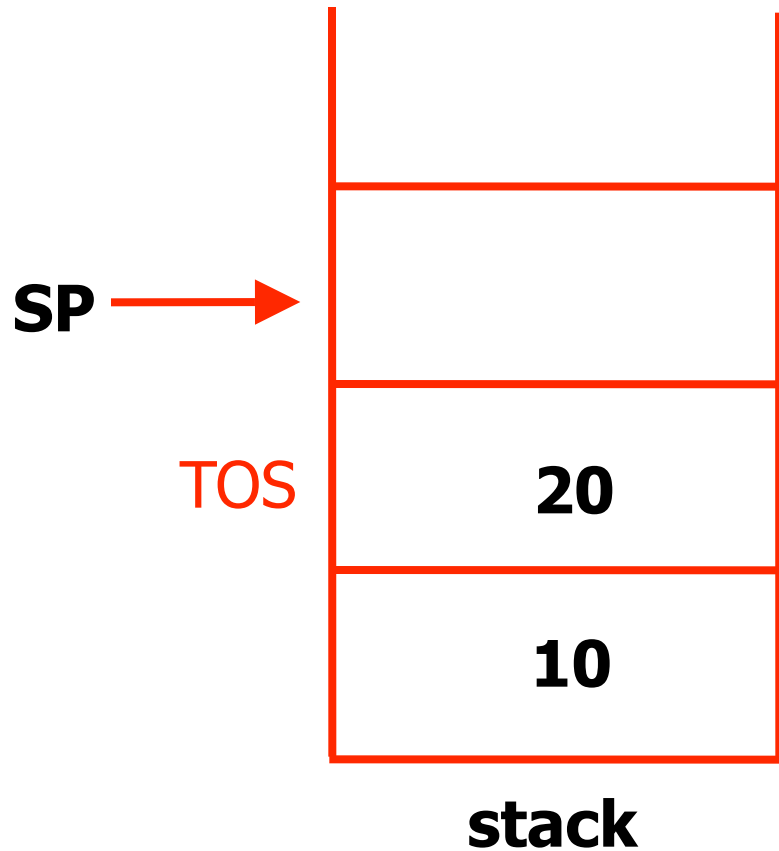
Push and pop (3)



**push 10 onto
stack**



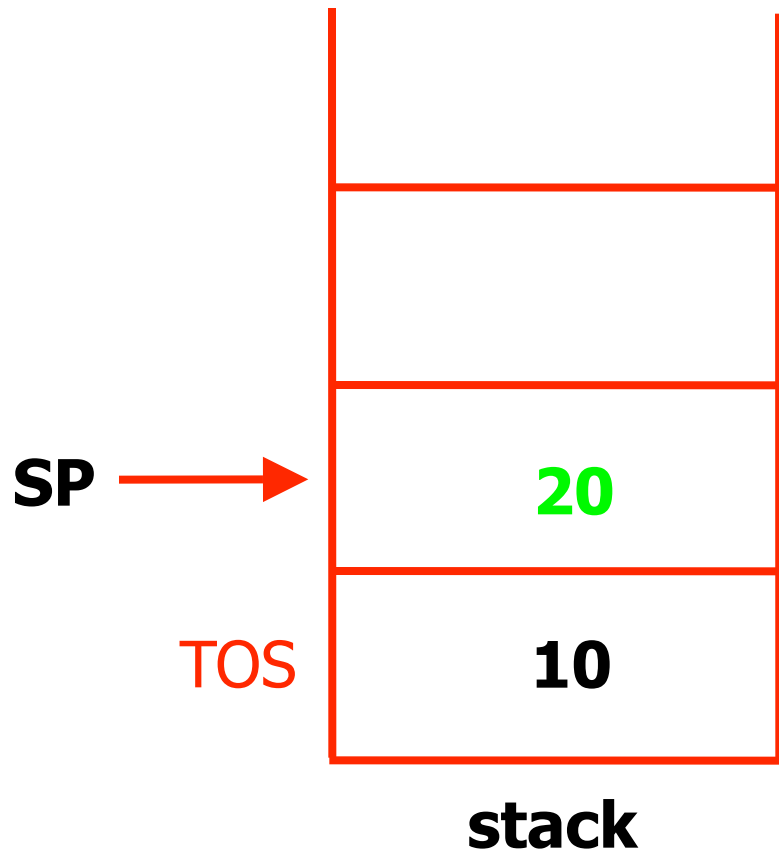
Push and pop (4)



**push 20 onto
stack**



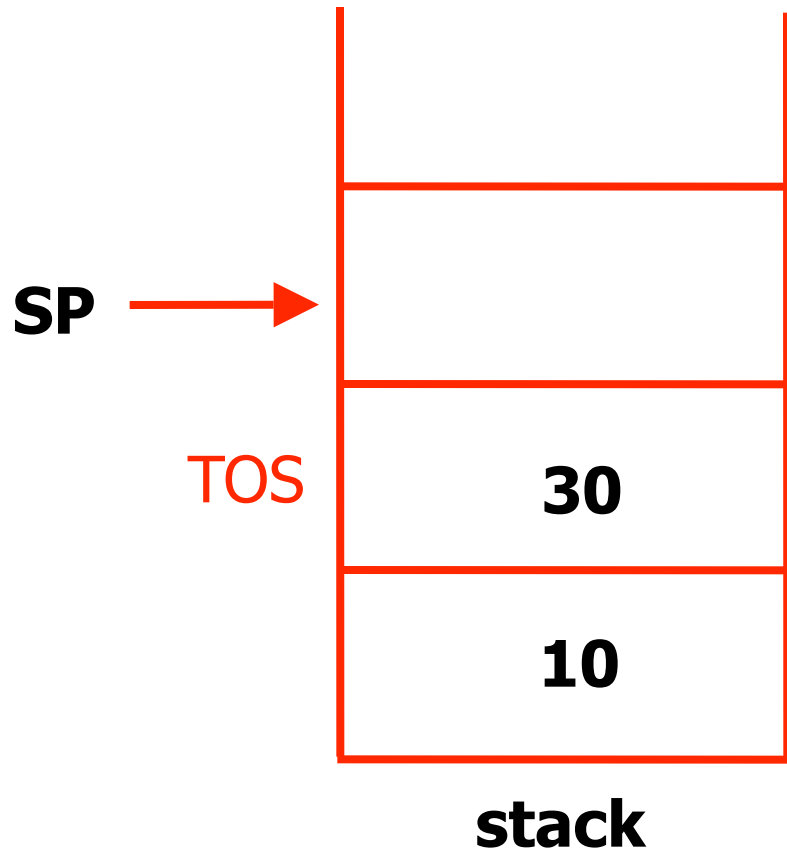
Push and pop (5)



pop stack;
20 still there,
but will be
overwritten next
push



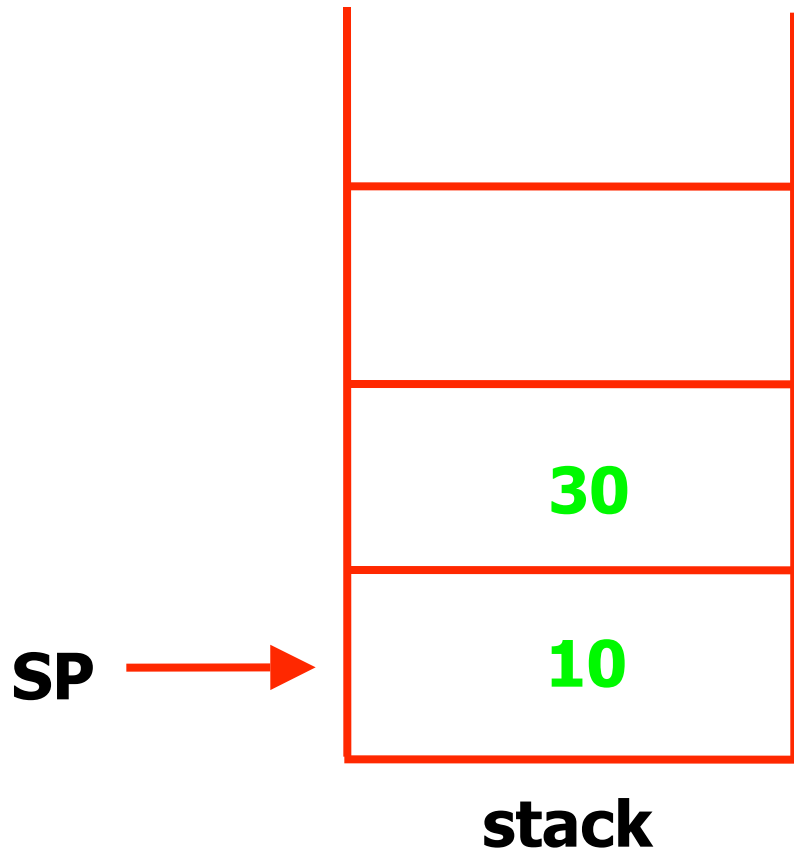
Push and pop (6)



**push 30 onto
stack;**
**old value (20)
gets overwritten**



Push and pop (7)



pop twice;
stack is now
"empty" again



VM instruction set (1)

- VM instructions are often called "bytecode"
 - because they fit into a byte (8 bits)
 - represented as an **unsigned char**
- Our VM has 14 different instructions
 - some take operands (some number of bytes)
 - some don't

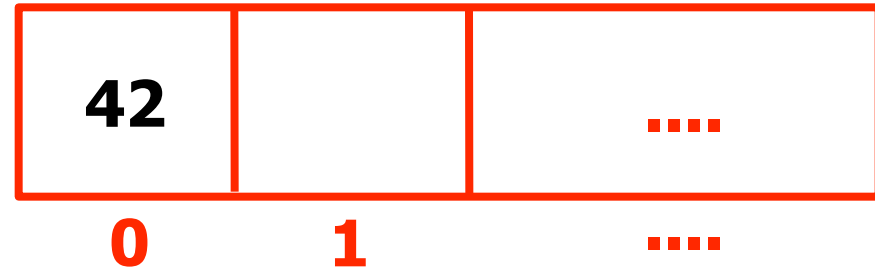
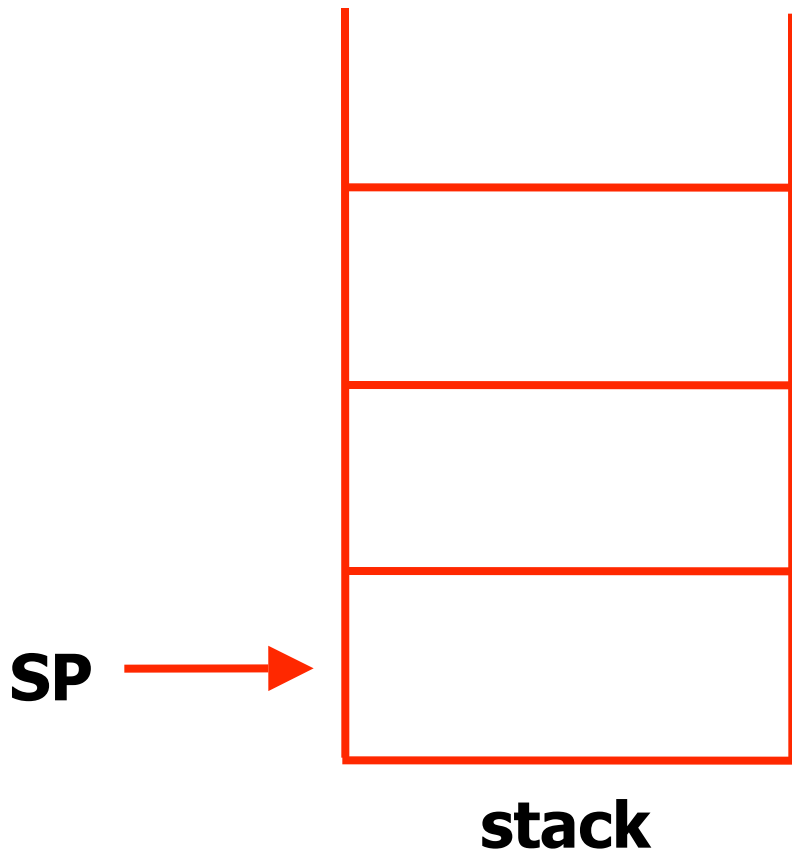


VM instruction set (2)

- Instructions:

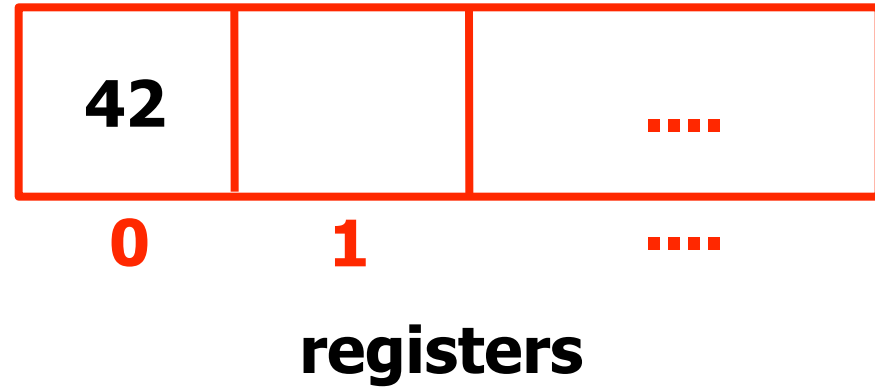
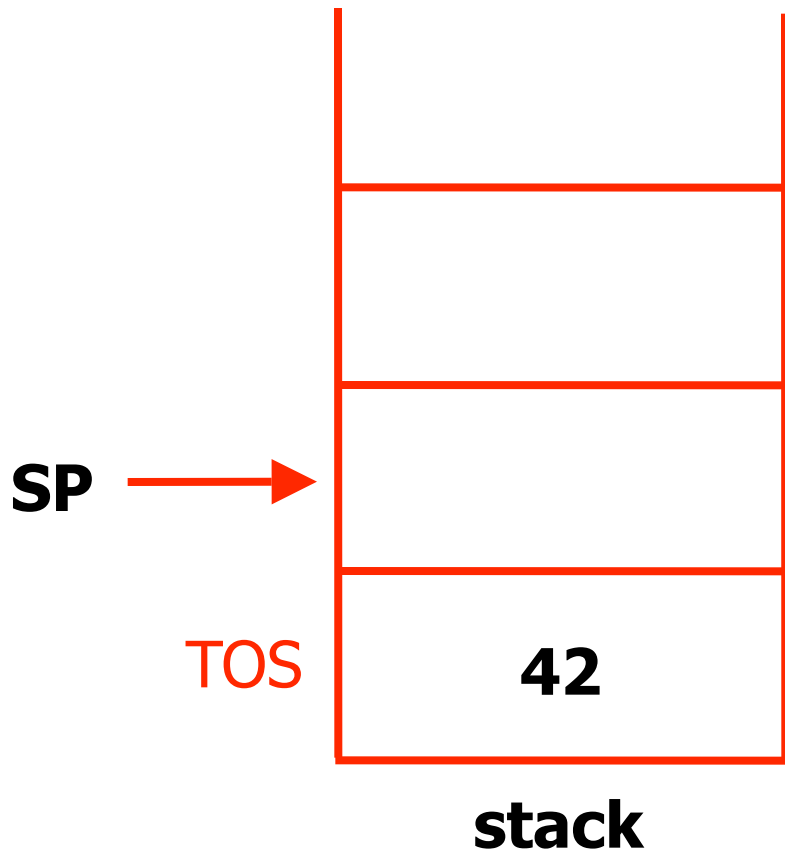
- **NOP** (0x00) – does nothing ("No **O**Peration")
- **PUSH** (0x01) – **PUSH** <n> pushes the integer <n> onto the stack
- **POP** (0x02) – removes the top element on the stack
- **LOAD** (0x03) – **LOAD** <r> pushes contents of register <r> to the top of the stack
- **STORE** (0x04) – **STORE** <r> pops top of stack and puts contents into register <r>

Load (1)



load 0

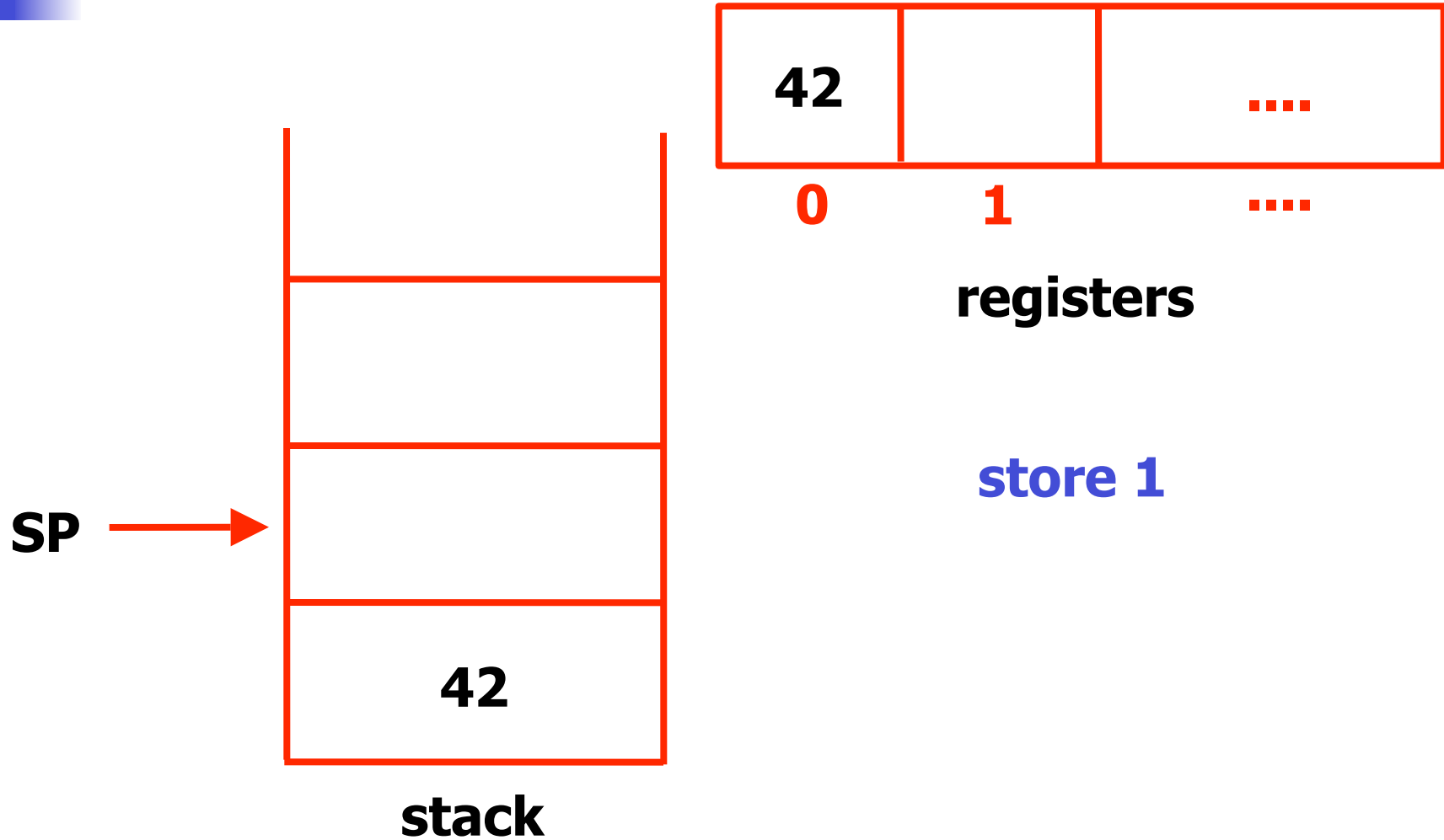
Load (2)



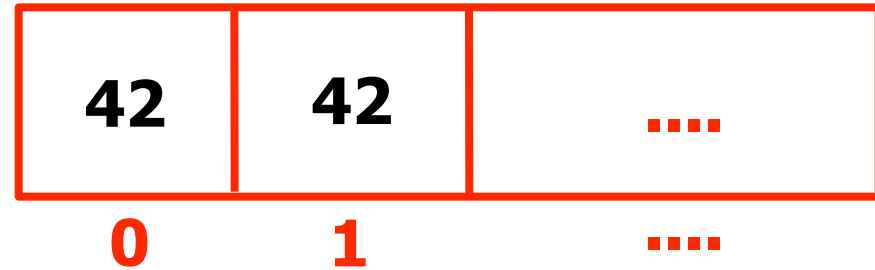
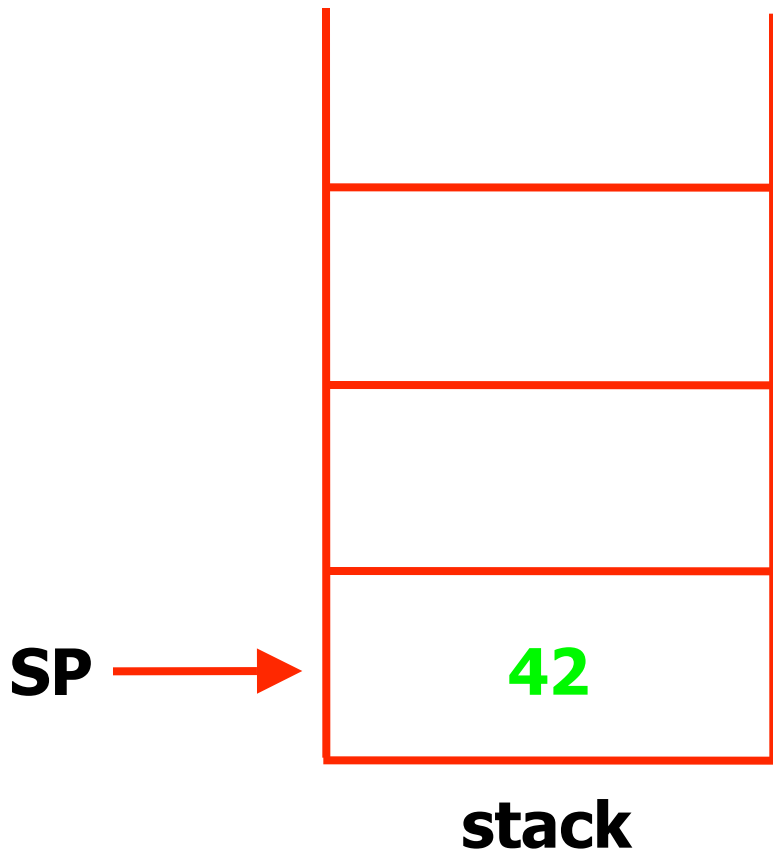
load 0
42 pushed onto stack



Store (1)



Store (2)



store 1

**topmost element
of stack copied
into register 1;
stack popped**



VM instruction set (3)

- Control flow instructions:
 - **JMP** (0x05) – **JMP <i>** sets the instruction pointer (IP) to <i> ("jump")
 - **JZ** (0x06) – **JZ <i>** sets IP to <i> only if the top value on the stack (TOS) is zero; also pops stack ("jump if zero")
 - **JNZ** (0x07) – **JNZ <i>** sets IP to <i> only if the TOS is not zero; also pops stack ("jump if nonzero")



VM instruction set (4)

- Arithmetic instructions:
 - **ADD** (0x08) – pops the top two entries in the stack, adds them, pushes result back
 - **SUB** (0x09) – pops the top two entries in the stack, subtracts them, pushes result back
 - Watch order! Should be **S2 – S1** on TOS
 - **MUL** (0x0a) and **DIV** (0x0b) defined similarly



Sub (1)

SP



TOS

31

42

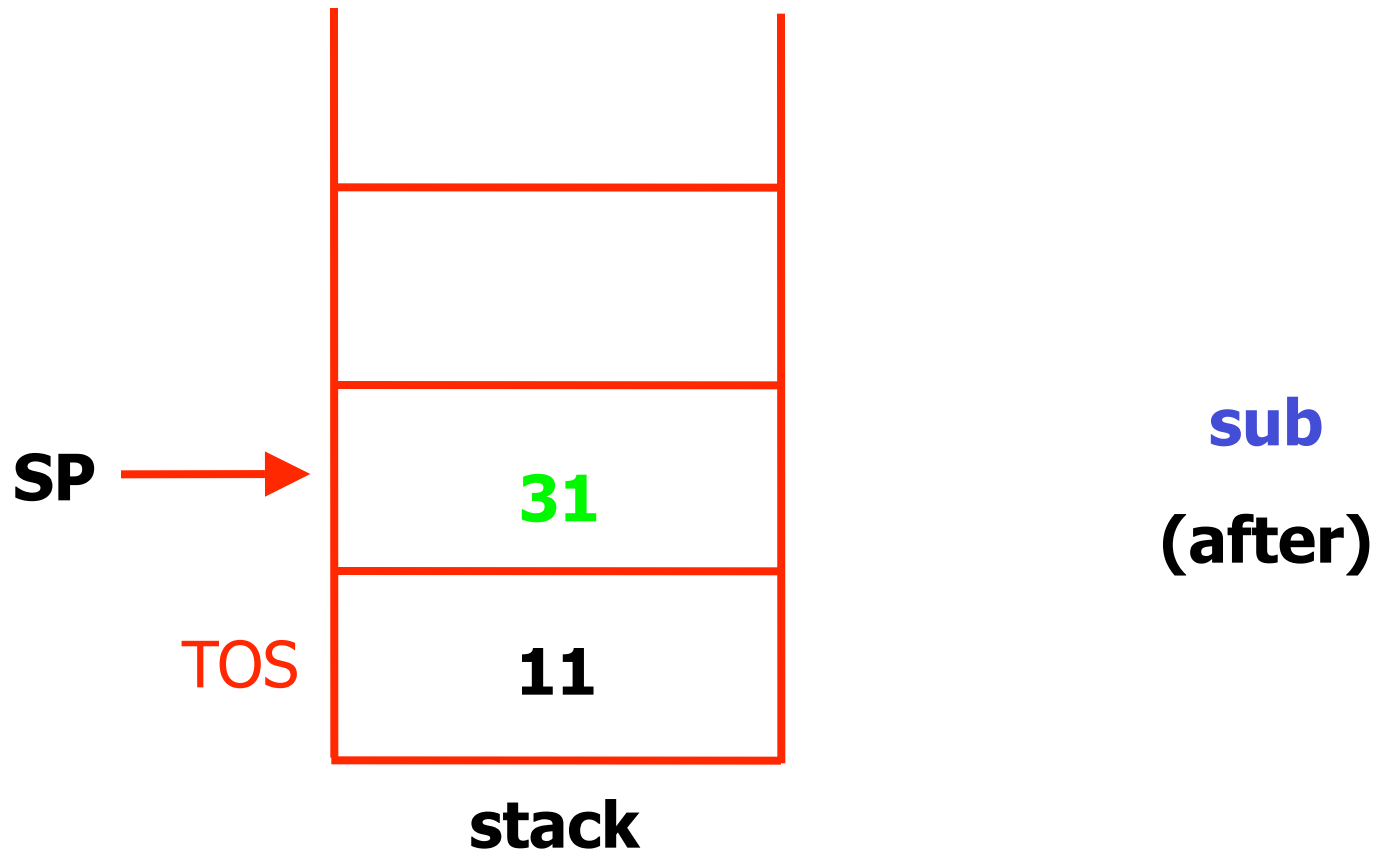
stack

sub

(before)



Sub (2)





VM instruction set (5)

- Other instructions:
 - **PRINT** (0x0c) – prints the TOS to stdout and pop TOS
 - **STOP** (0x0d) – terminates the virtual program



Example program (1)

- Program to generate factorial of 10 (10!)
- Which means...?
 - $10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$
 - $= 3628800$
- But we'll write a program in our virtual machine's language



Example program (2)

- Register 0 will contain the count
- Register 1 will contain the running total
- Register 0 will start off at 10
 - each step, will decrease by 1
- Register 1 will start off at 1
 - each step, will be multiplied by register 0 contents
- Continue until register 0 has 0
 - result is in register 1



Example program (3)

```
/* Initialize the registers. */
```

```
1 push 10
```

```
2 store 0
```

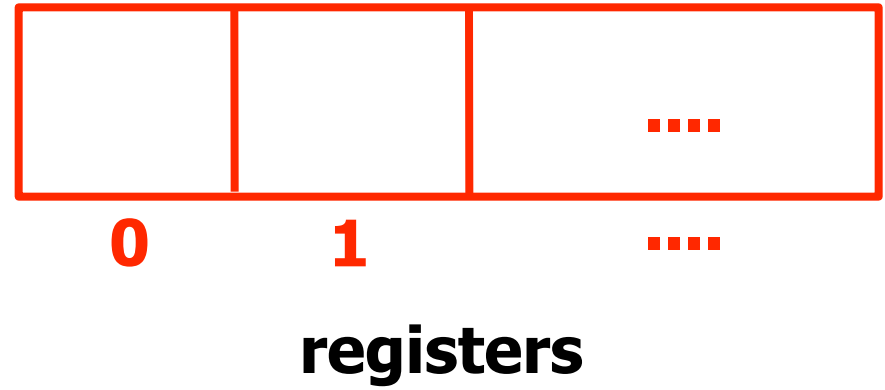
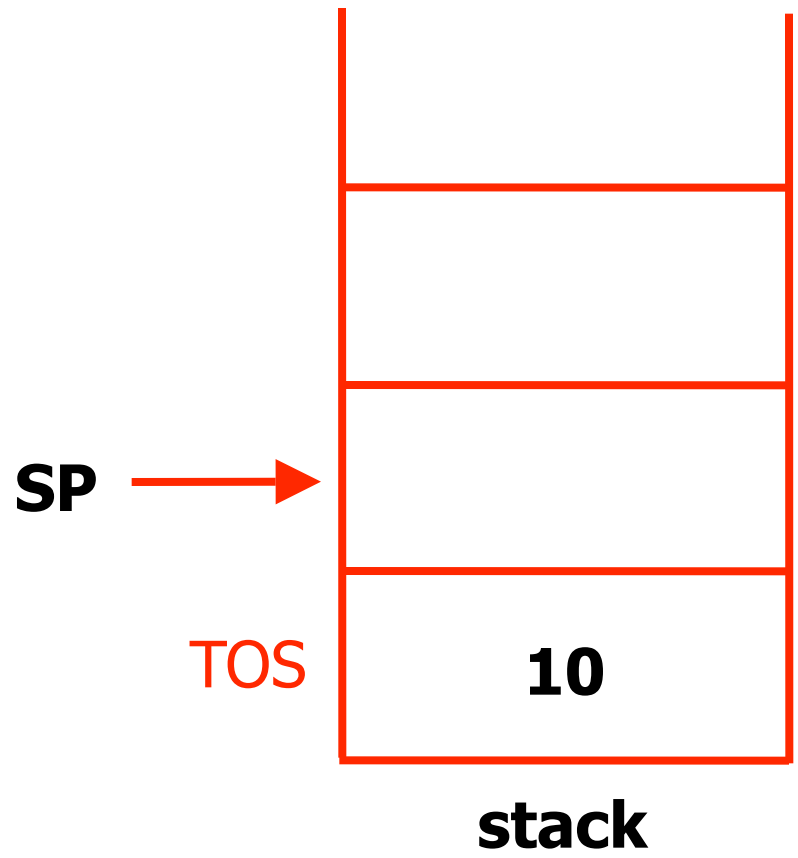
```
3 push 1    /* Initialize result. */
```

```
4 store 1
```

```
/* continued on next slide... */
```



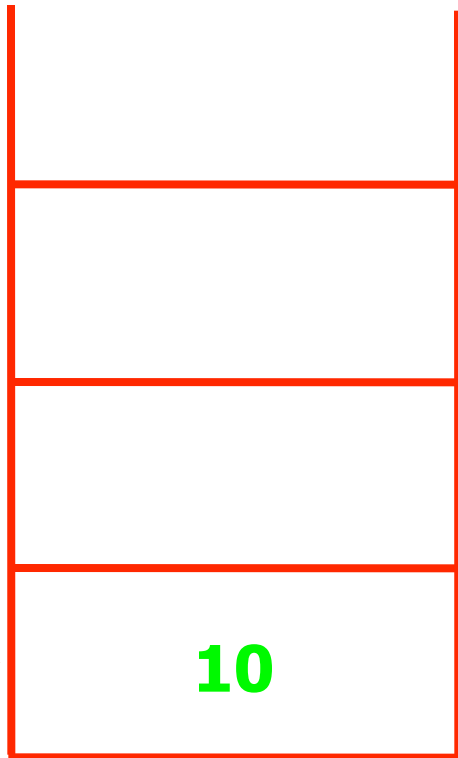
push 10



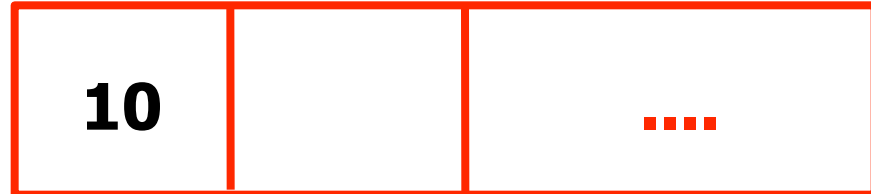


store 0

SP



stack



10

1

...

0

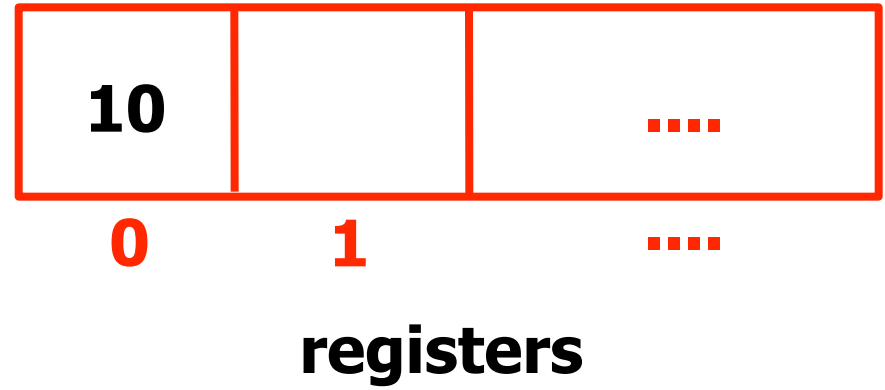
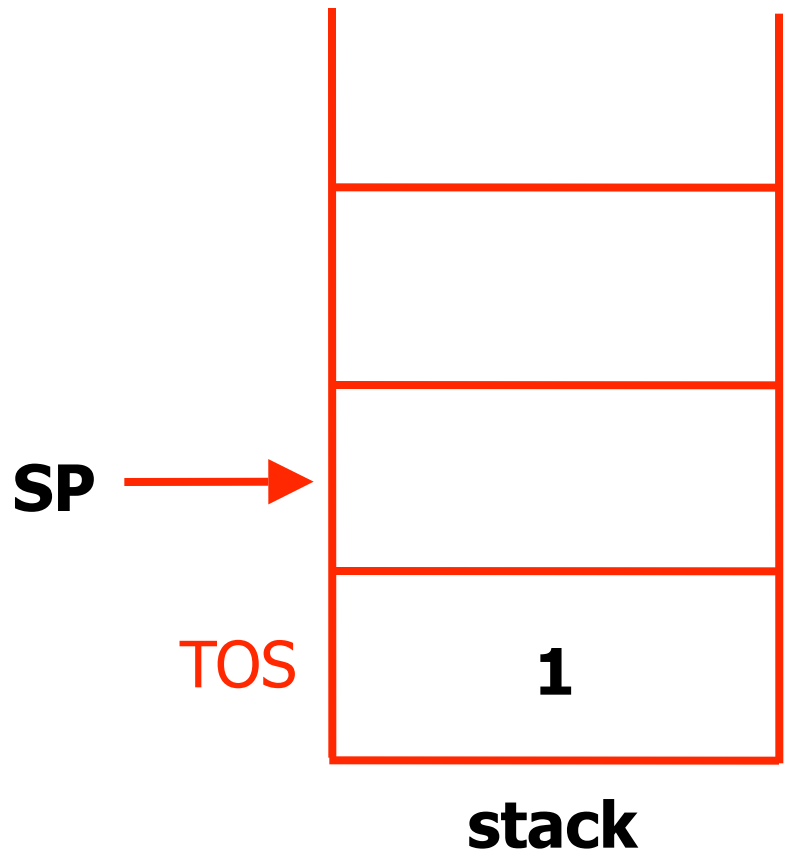
1

...

registers

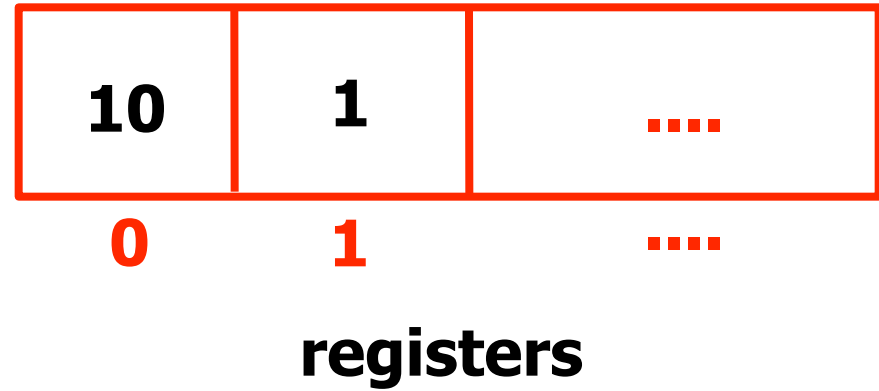
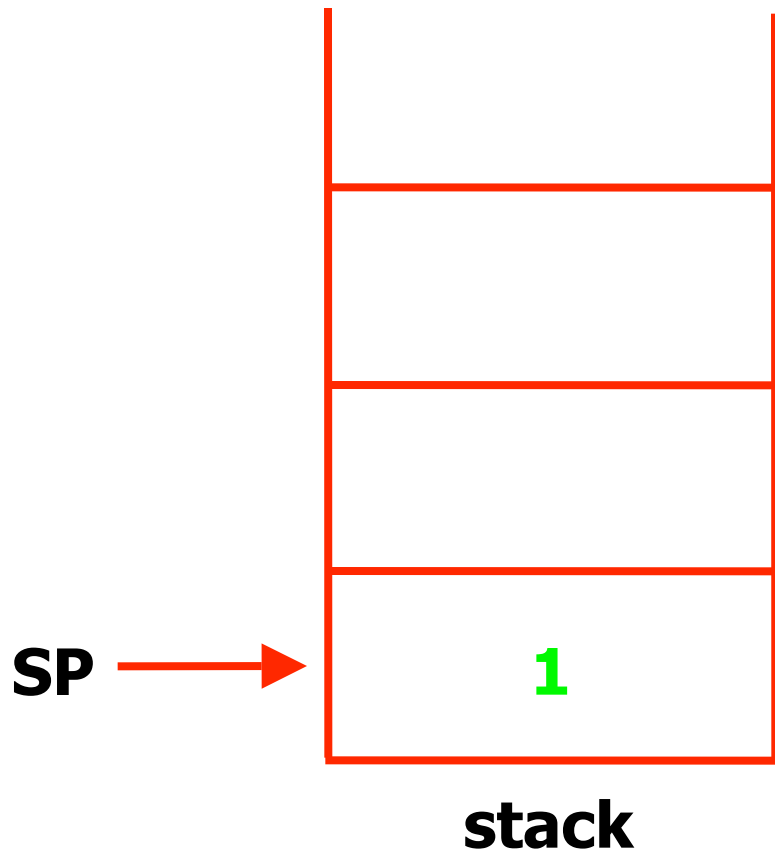


push 1





store 1





Example program (4)

```
/* Put counter value on stack.
```

```
If it's 0, we're done; register 1  
contains the final value. */
```

```
5 load 0 /* Load current count. */
```

```
6 jz 16 /* if 0, jump to instruction 16 */
```

```
/* 16 is the location of the instruction which  
is the target of the jz instruction i.e.  
where to jump to. */
```



Example program (5)

```
/* result = result * count */
```

```
7 load 1
```

```
8 load 0
```

```
9 mul
```

```
10 store 1
```

```
/* count = count - 1 */
```

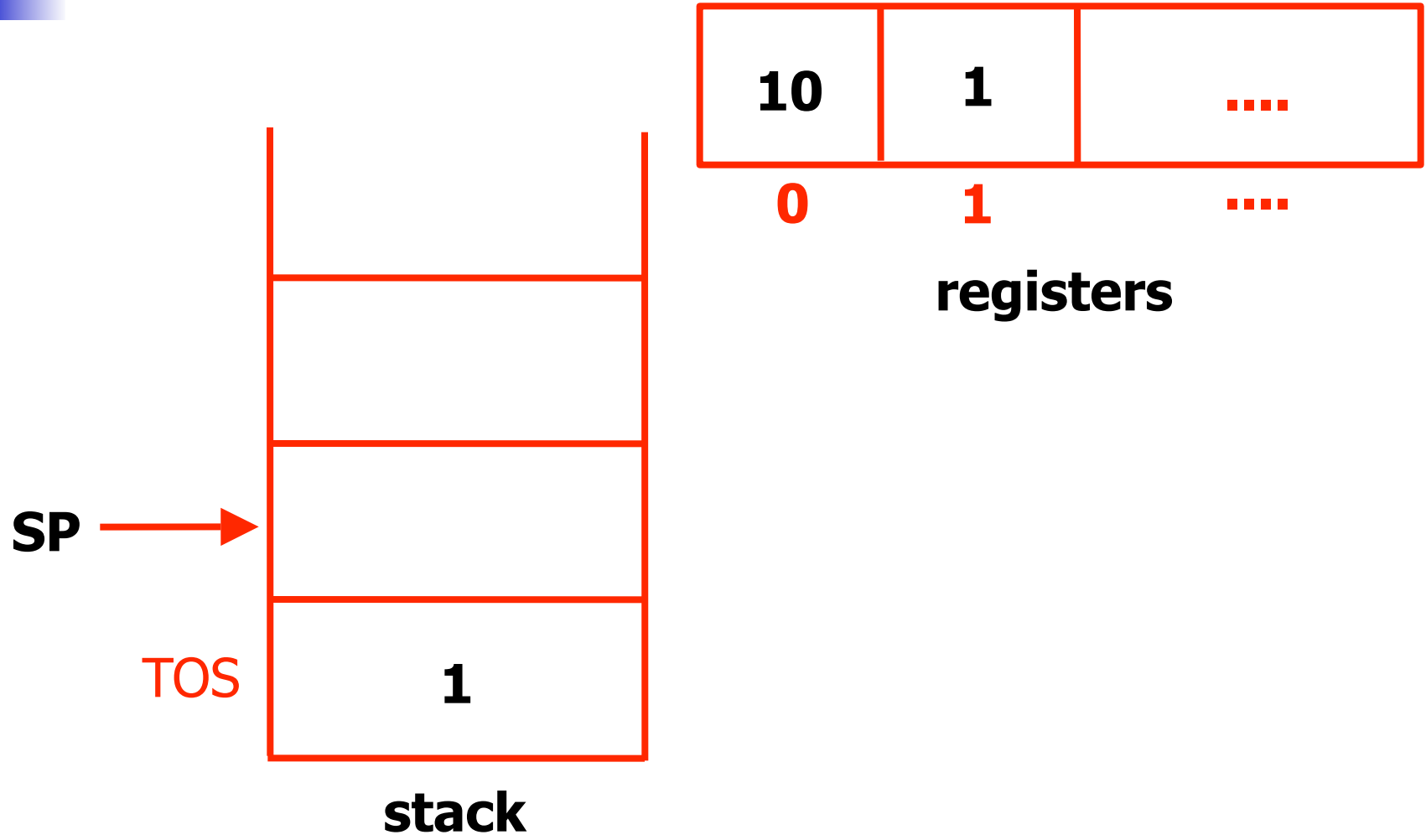
```
11 load 0
```

```
12 push 1
```

```
13 sub
```

```
14 store 0
```

load 1



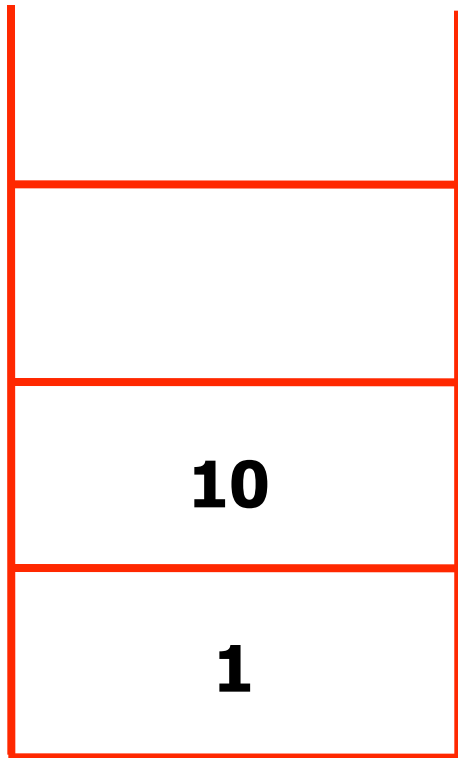


load 0

SP



TOS



10

1

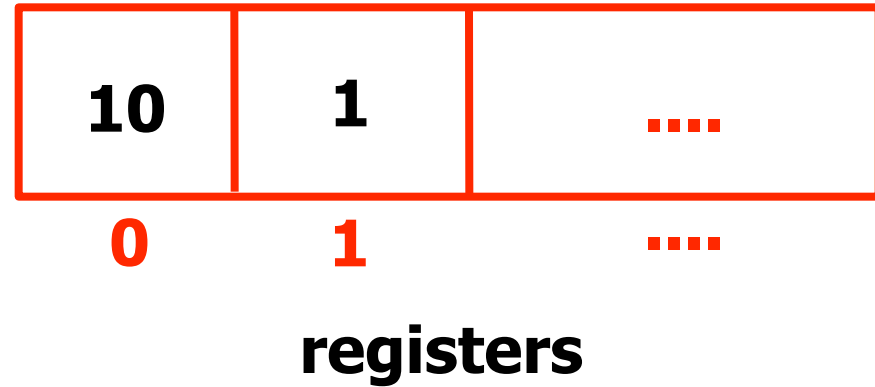
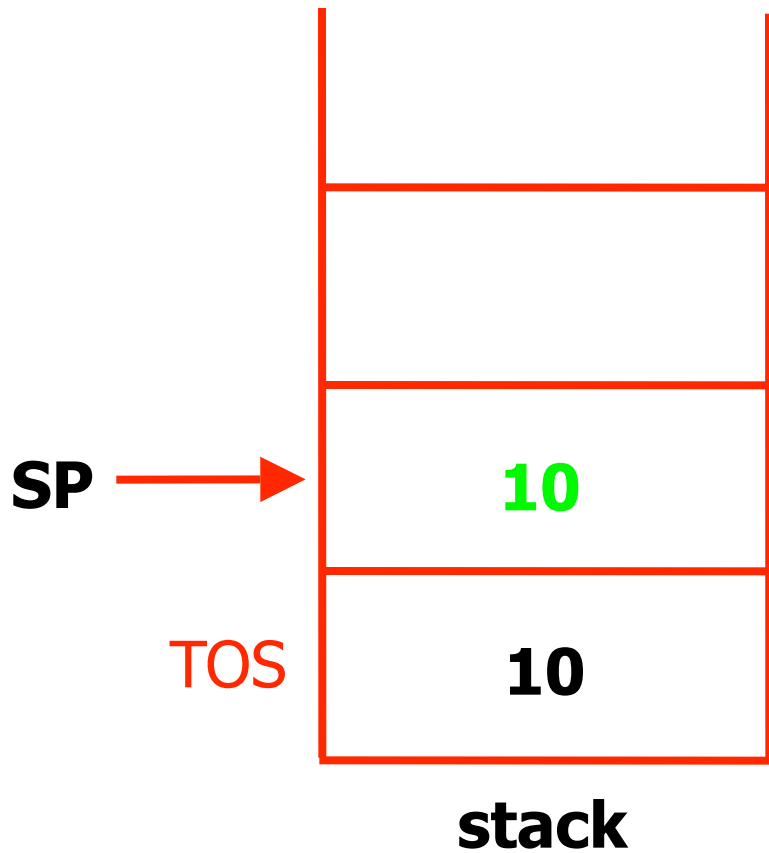
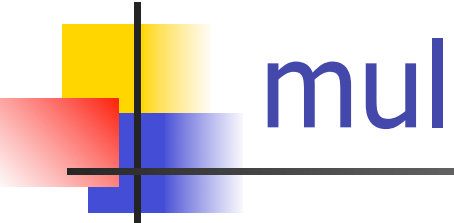
....

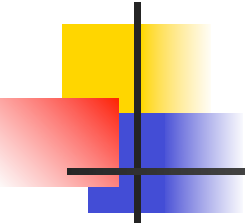
0

1

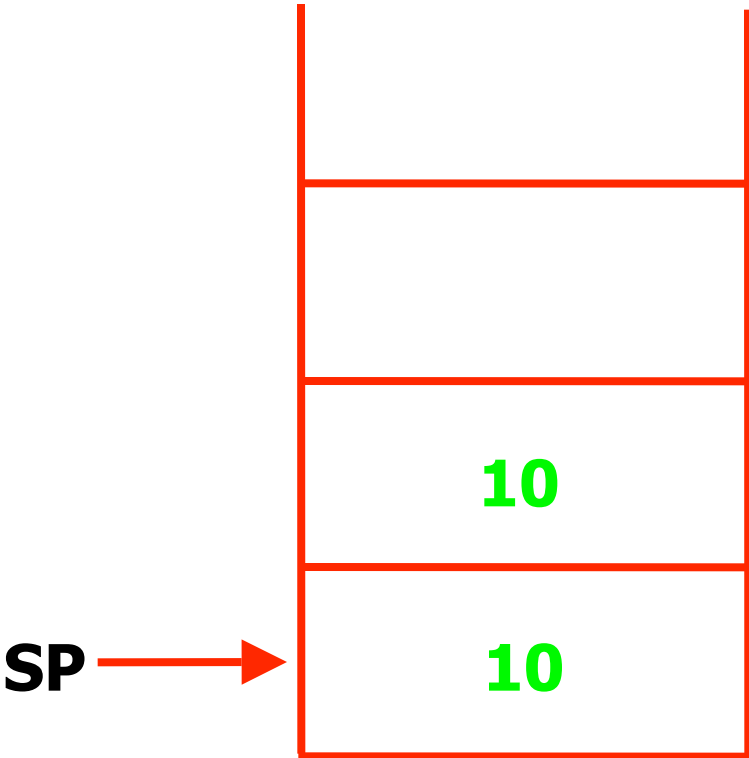
....

registers

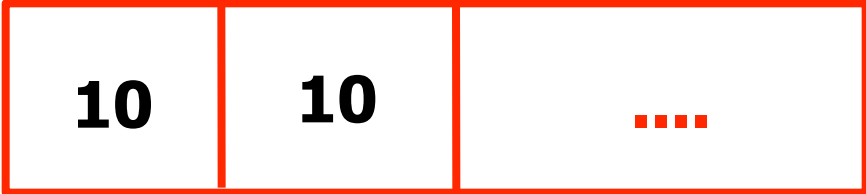




store 1



stack



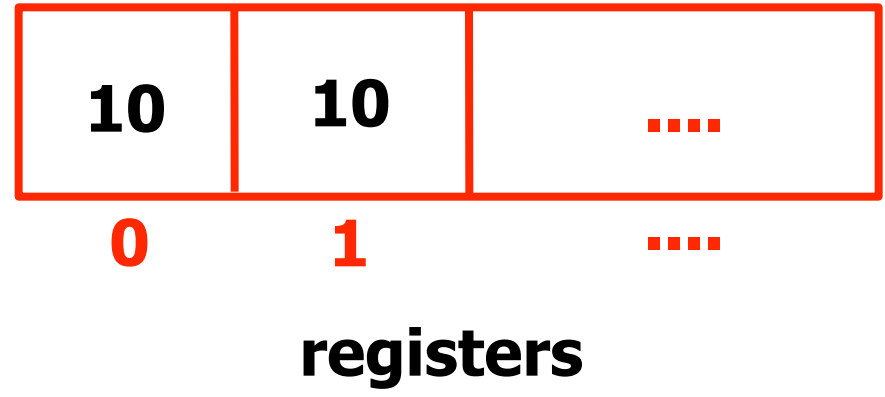
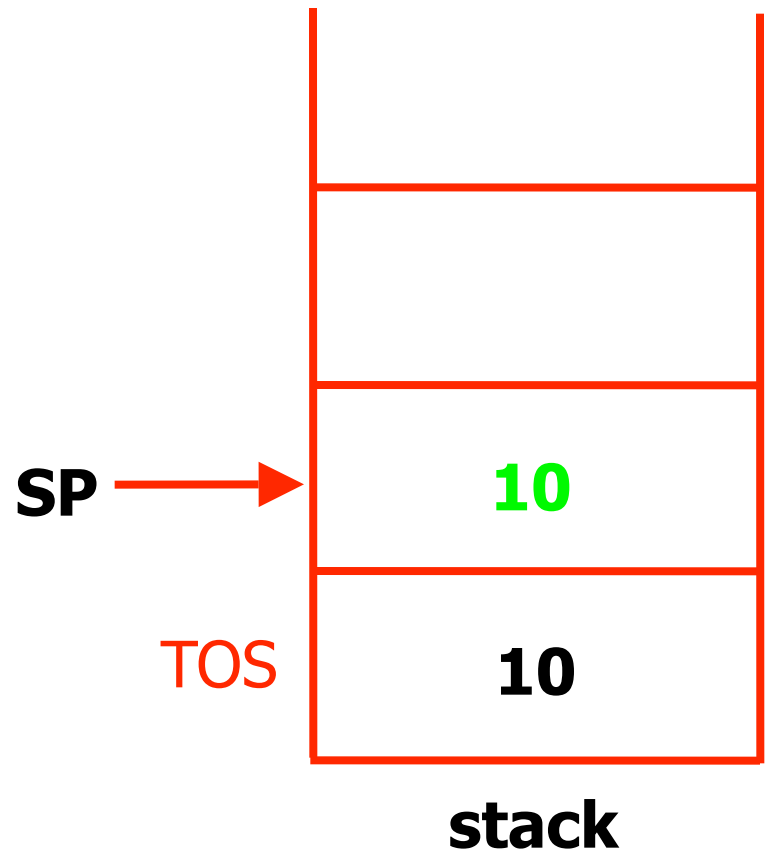
0

1

....

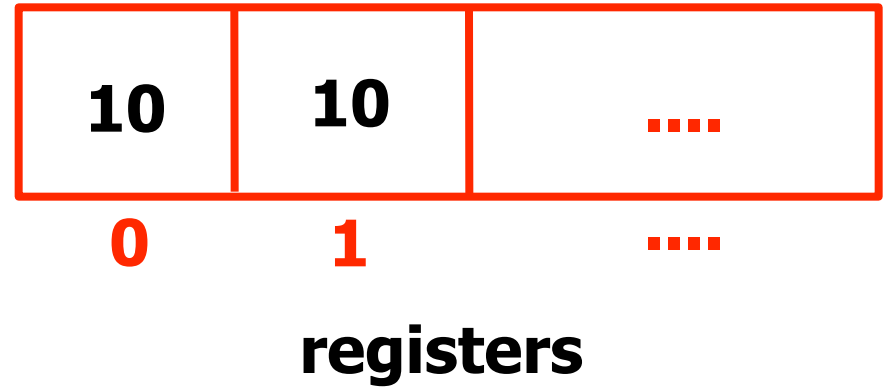
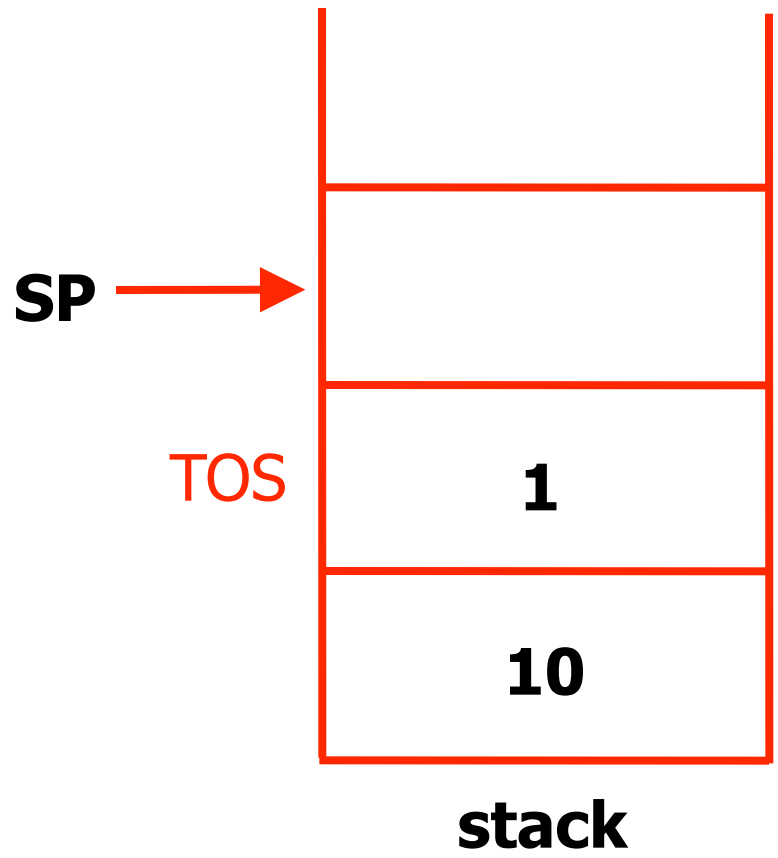
registers

load 0





push 1



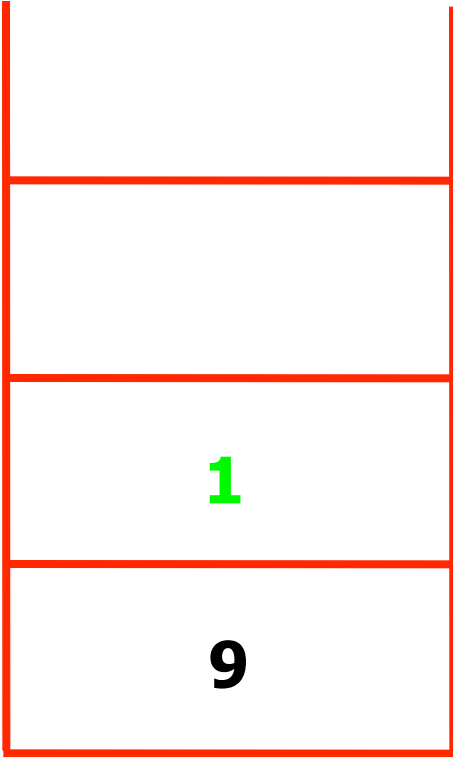


sub

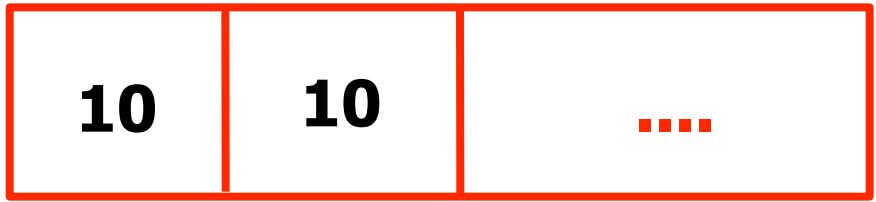
SP



TOS



stack



10

10

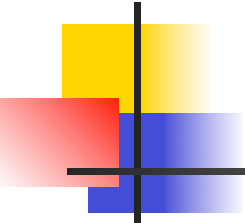
...

0

1

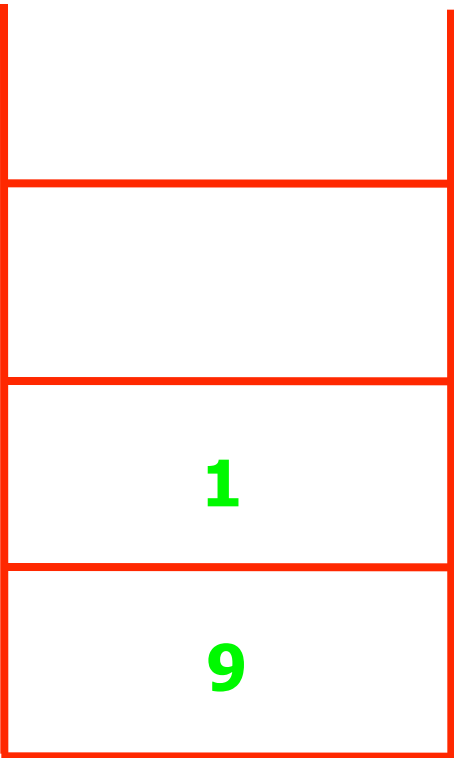
...

registers

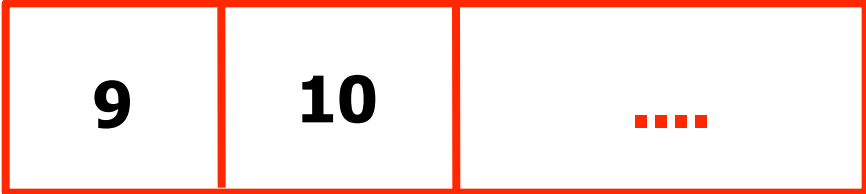


store 0

SP →



stack



0

1

...

registers

etc. ...



Repeating...

- Registers start off as 10, 1
- Then become 9, 10
- 8, $10*9$
- 7, $10*9*8$
- ...
- 0, $10!$
- ... and we're done.



Example program (6)

```
/* Go back and loop until done. */
```

```
15 jmp 5
```

```
/* When we get here, we're done. */
```

```
16 load 1
```

```
17 print
```

```
18 stop
```

```
/* End of program. */
```



Lab 8

- Program is given to you
- You need to write the byte-code interpreter
- Most of code is supplied; have to fill in the guts of the instruction-processing code
- Looks complicated but actually is pretty easy
- Watch out for error checking e.g.
 - popping an empty stack
 - pushing to a full stack
 - accessing non-existent register or instruction



Lab 8 -- error checking

- One subtlety with stack pushes
- If stack pointer is at 255, and you push onto stack, what is the new stack pointer value?
 - 0
 - (256 is too large for an **unsigned char**)
- But this is clearly incorrect
- How to detect "stack overflow"?
- Solution: don't allow stack to get to 255
- If stack pointer is 254, a push is invalid



Finally...

- Hope you enjoyed the course!
- If so, consider taking
 - other CS 11 tracks
 - (C++, java, advanced C/C++/java, python, ocaml, haskell)
 - CS 11 project track
 - CS 24
 - CS 2, 3 for larger-scale software projects