

9/16/26

COMP 311:

Computer Organization!

Lecture 8: First Look at Hardware ☺



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp311-fa26>

Logistics Update



- Lab 1 is out!
 - Due in 2 weeks.

Plan for Class Today



- 3:35-4:10: Quiz
- 4:10-4:15: BREAK
- 4:15-4:25: The Frame Pointer, Close Out Unit 1
- 4:25-4:50: Intro to Transistors: Our Most Fundamental HW Unit

Guiding Question 😊

How do we manage the complexity of function calling in assembly?

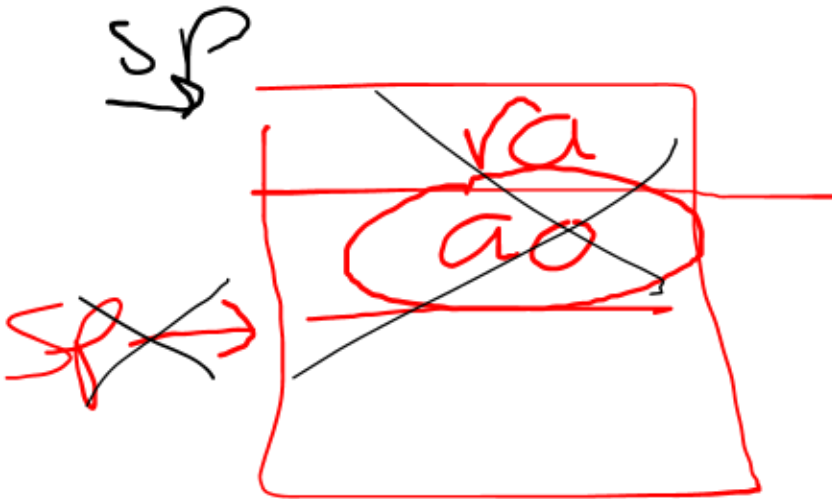
Calling Convention Details



Procedure Linkage is Nontrivial

- The details can be overwhelming. How do we manage this complexity?
 - Abstraction: High-level languages hide the details
- There are great many implementation choices:
 - which variables are saved
 - who saves them
 - where are arguments stored?
- Solution: **CONTRACTS!**
 - Caller and Callee must agree on the details

Code Example



Set up stack frame,
save ra,
save original arg

sqr

```
addi sp, sp, -8  
sw    ra, 4(sp)  
sw    a0, 0(sp)
```

Check base case

```
slti t0, a0, 2  
bne t0, x0, base
```

Make recursive call

```
addi a0, a0, -1  
jal  ra, sqr
```

Restore original arg
Compute
Skip to base
case/return

```
lw    a0, 0(sp)  
add   a0, a0, a0  
add   a0, a0, a0  
addi  a0, a0, -1  
j     done
```

Restore the caller's
saved return address,
give back 8 bytes of
stack space,
Jump back to caller

```
base:  
    # return n
```

```
done:  
    lw    ra, 4(sp)  
    addi  sp, sp, 8  
    jr    ra
```



Procedure Linkage: Caller Contract



The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame
(t0-t6, a0-a7)
- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

Call procedure, using a **jal** instruction (places return address in ra).

- Access procedure's return values in a0-a1

Code Lawyer

- Our example is a CALLER. Let's make sure it obeys its contractual obligations

```
sqr:  addi sp, sp, -8
      sw   ra, 4(sp)
      sw   a0, 0(sp)
```

```
      slti t0, a0, 2
      bne  t0, x0, base
```

```
      addi a0, a0, -1
      jal  ra, sqr
```

```
      lw   a0, 0(sp)
      add  a0, a0, a0
      add  a0, a0, a0
      addi a0, a0, -1
      j    done
```

```
base:
      # return n
```

```
done:
      lw   ra, 4(sp)
      addi sp, sp, 8
      jr   ra
```



Code Lawyer

sqr:

```
addi sp, sp, -8  
sw    ra, 4(sp)  
sw    a0, 0(sp)
```

```
slti t0, a0, 2  
bne  t0, x0, base
```

```
addi a0, a0, -1
```

```
jal  ra, sqr
```

```
lw    a0, 0(sp)
```

```
add  a0, a0, a0  
add  a0, a0, a0  
addi a0, a0, -1  
j    done
```

```
base:  
# return n
```

```
done:  
lw    ra, 4(sp)  
addi sp, sp, 8  
jr    ra
```



- Our example is a CALLER. Let's make s contractual obligations

The CALLER will:

- Save all caller-saved registers that it wants to survive subsequent calls in its stack frame

(t0-t6, a0-a7)

- Pass the first 8 arguments in registers a0-a7, and save subsequent arguments on stack,

Call procedure, using a **jal** instruction (places return address in ra).

- Access procedure's return values in a0-a1

Procedure Linkage: Callee Contract



If needed the CALLEE will:

- 1) Allocate a stack frame with space for saved registers, local variables, and spilled args
- 2) Save any “preserved” registers used:
(s0-s11, and ra if we call another fn)
- 3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set fp (s0) to a fixed location in stack frame (optional)
- 4) EXECUTE procedure ~~47~~
- 5) Place return values in a0-a1
- 6) Restore saved registers
- 7) Restore sp to its original value
- 8) Return to CALLER with jr ra or ret

sqr:

Code Lawyer: w/ Callee this Time

- Our example is also a CALLEE. Are these contractual obligations satisfied?



If needed the CALLEE will:

- 1) Allocate a stack frame with space for saved registers, local variables, and spilled args
- 2) Save any “preserved” registers used:
(s0-s11, and ra if we call another fn)
- 3) If CALLEE has local variables -or- needs access to args on the stack, save CALLER’s frame pointer and set fp (s0) to a fixed location in stack frame (optional)
 - 4) EXECUTE procedure
 - 5) Place return values in a0-a1
 - 6) Restore saved registers
 - 7) Restore sp to its original value
 - 8) Return to CALLER with jr ra or ret

```
addi sp, sp, -8
sw    ra, 4(sp)
sw    a0, 0(sp)
```

```
slti t0, a0, 2
bne t0, x0, base
```

```
addi a0, a0, -1
jal  ra, sqr
```

```
lw    a0, 0(sp)
add   a0, a0, a0
add   a0, a0, a0
```

```
addi a0, a0, -1
j     done
```

```
base:
    # return n
```

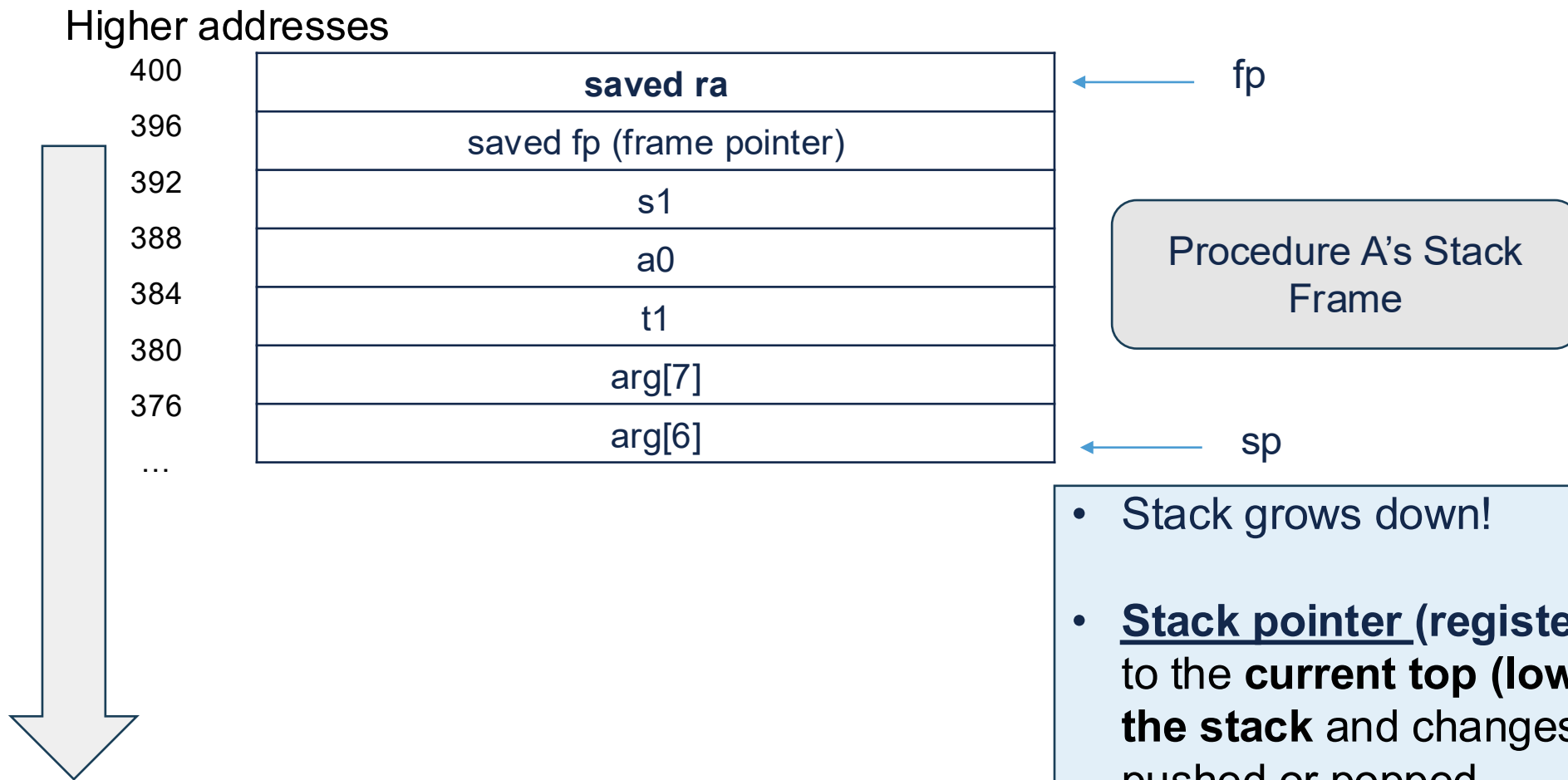
```
done:
    lw    ra, 4(sp)
    addi sp, sp, 8
    jr    ra
```

The Frame Pointer



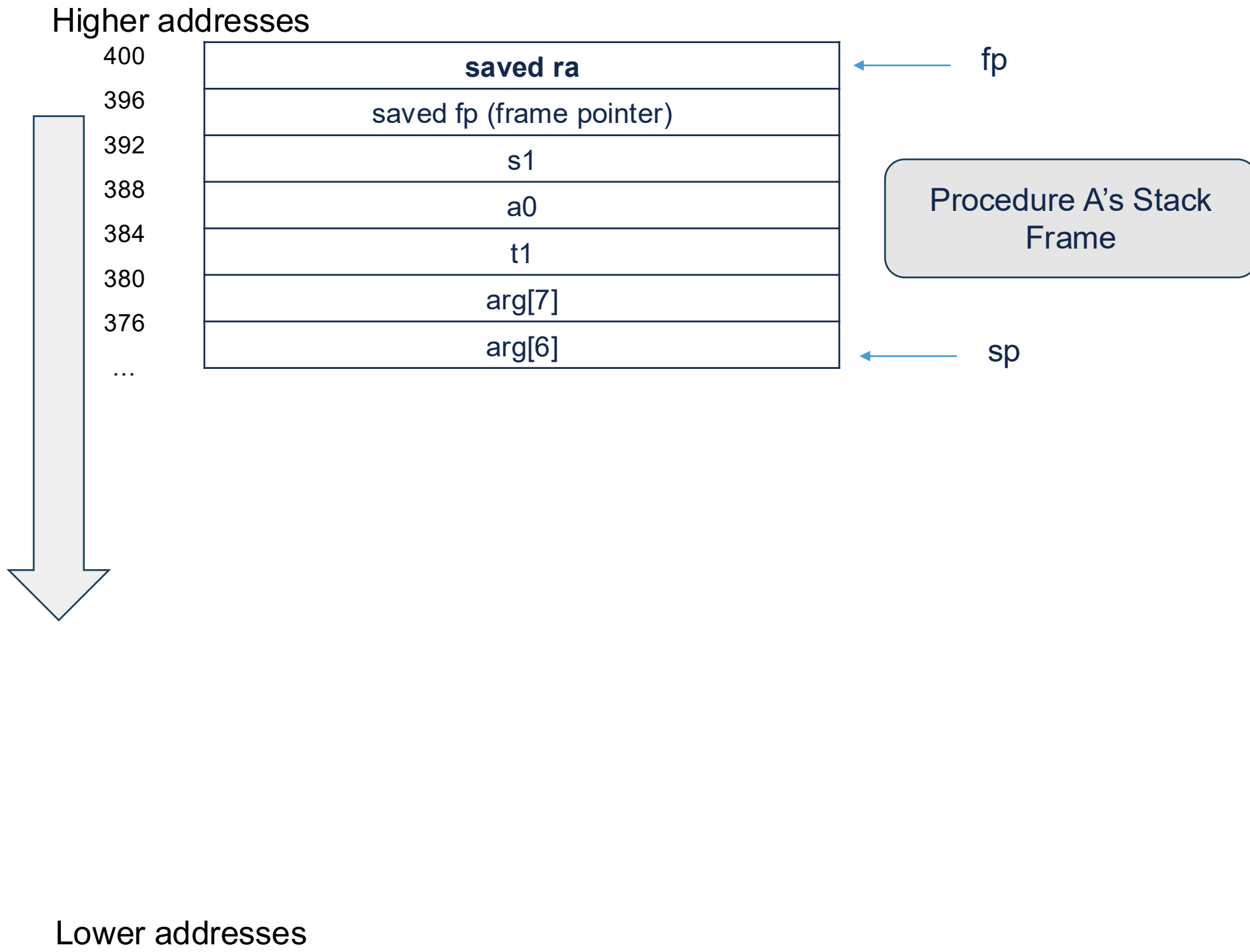
- Frame pointer (fp / s0) gives a stable reference point within the current stack frame.
- Useful because sp may move while a function is running.
- Local variables / saved values can be accessed at fixed offsets from fp.
- It is optional, many functions can use only sp!

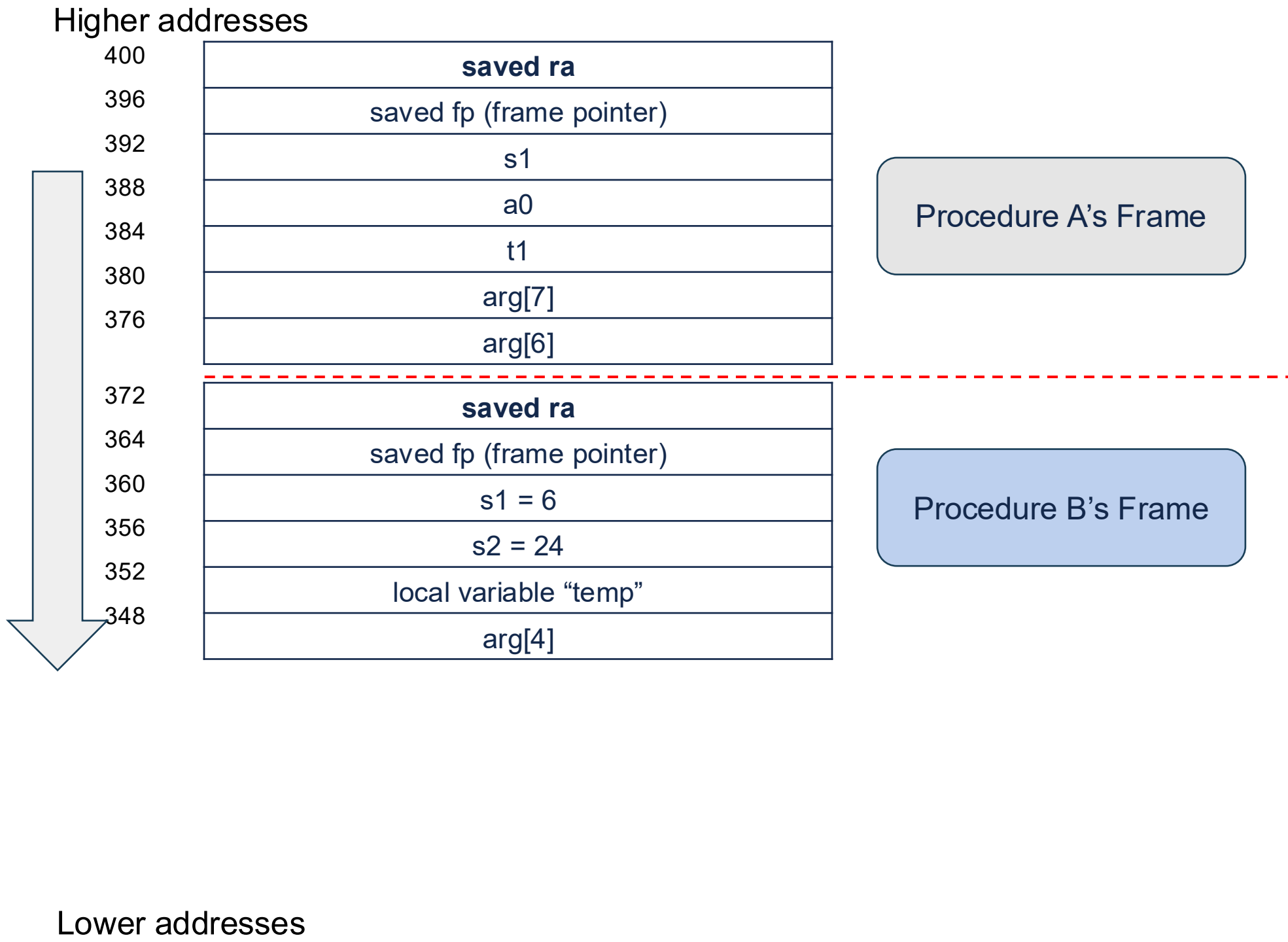
Think of sp as “where the stack is now”
and fp as “where this function’s frame started.”

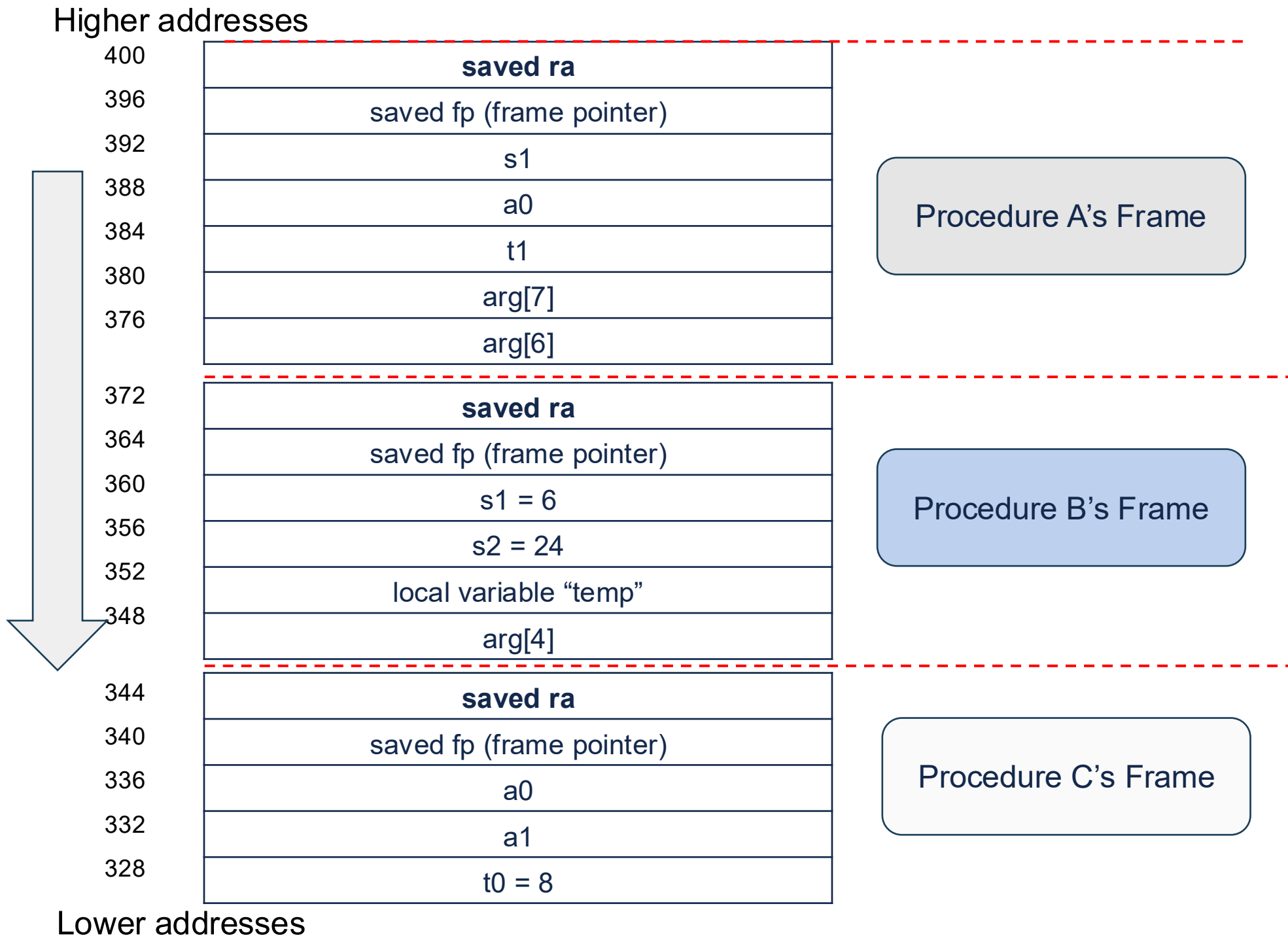


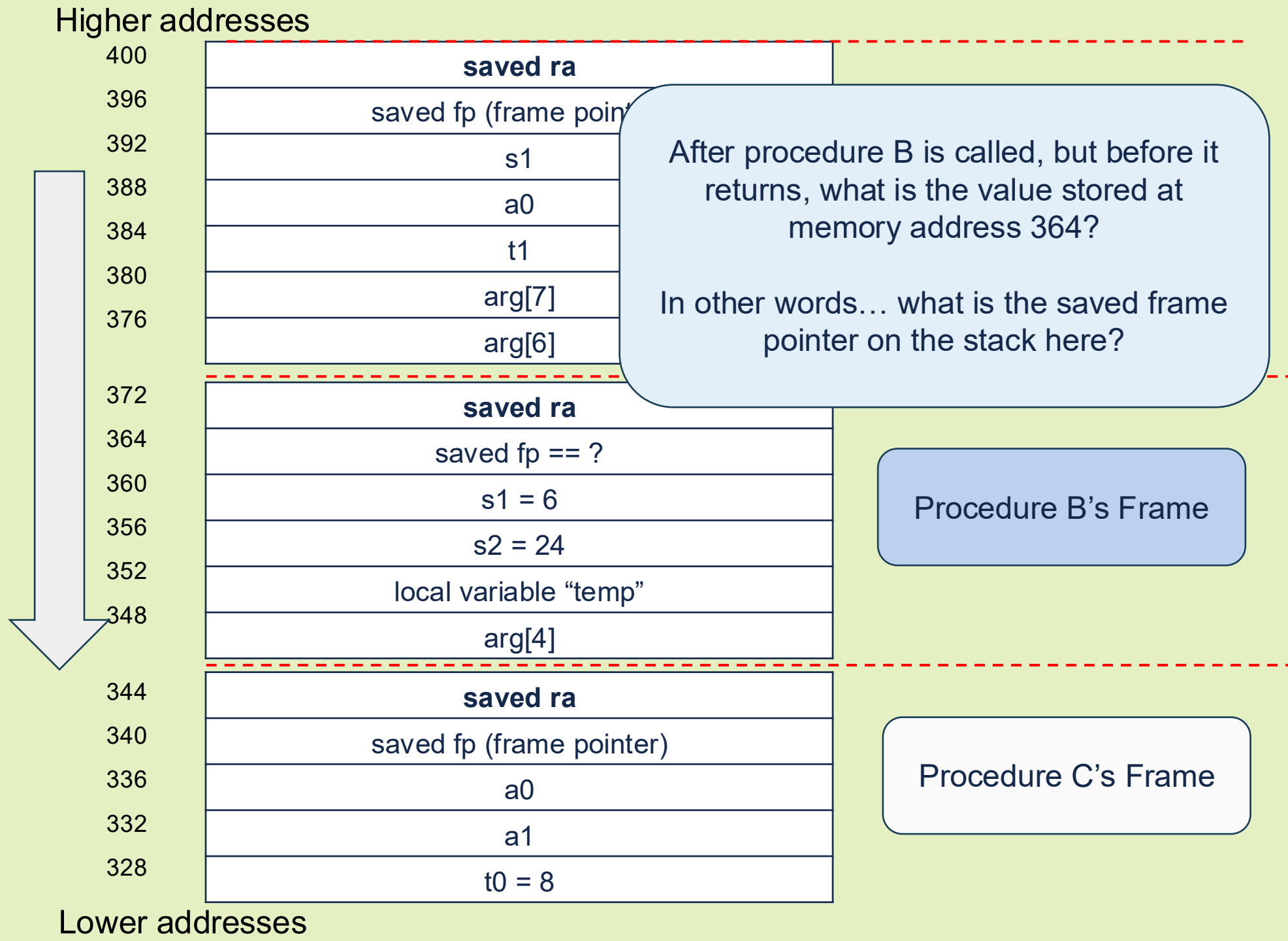
- Stack grows down!
- **Stack pointer (register sp/x2)**: points to the **current top (lowest address) of the stack** and changes as values are pushed or popped.
- **Frame pointer**: (fp/x8/s0) Provides a **stable** reference point within the **current stack frame** so data can be accessed with fixed offsets.

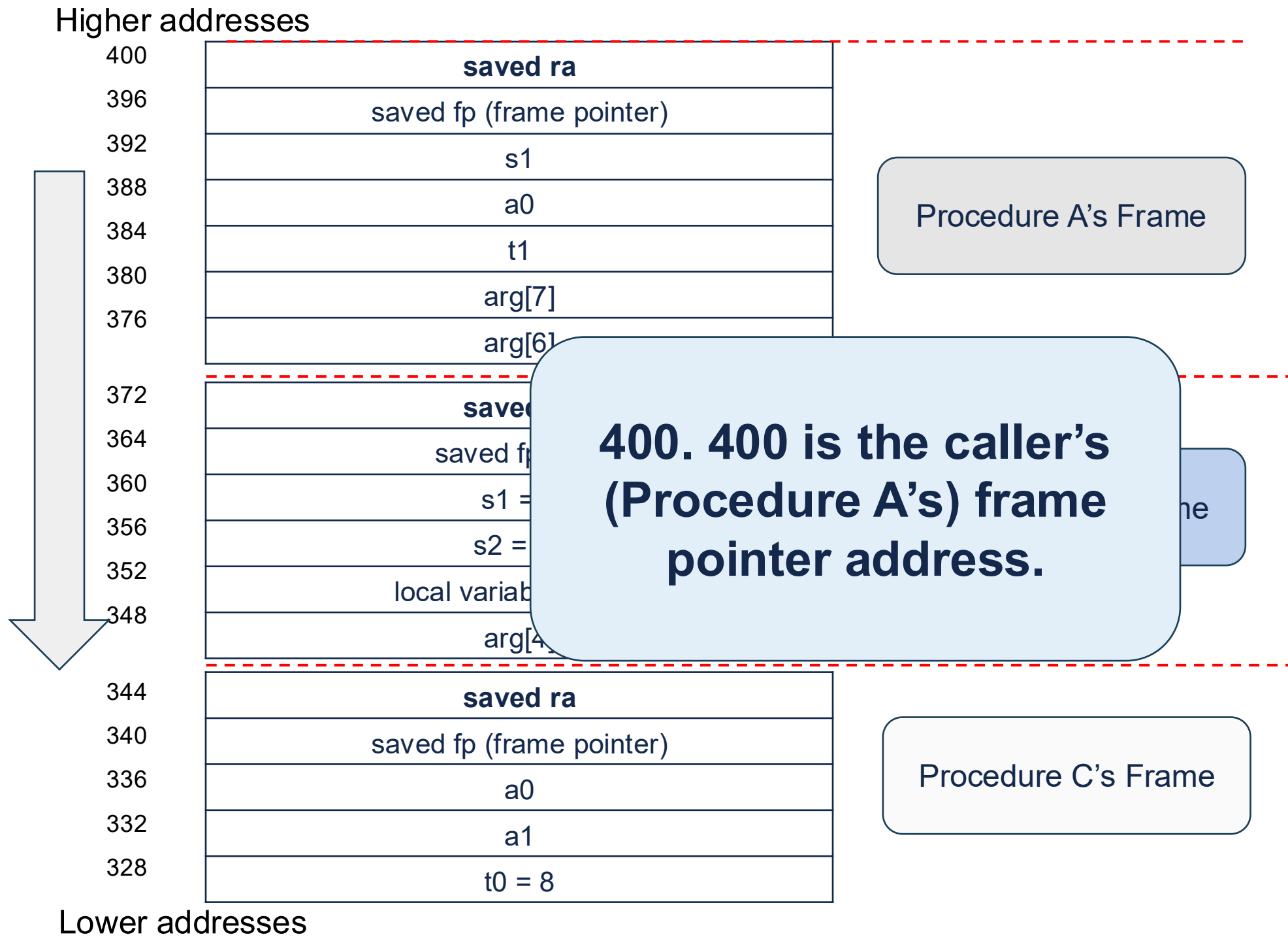
Lower addresses









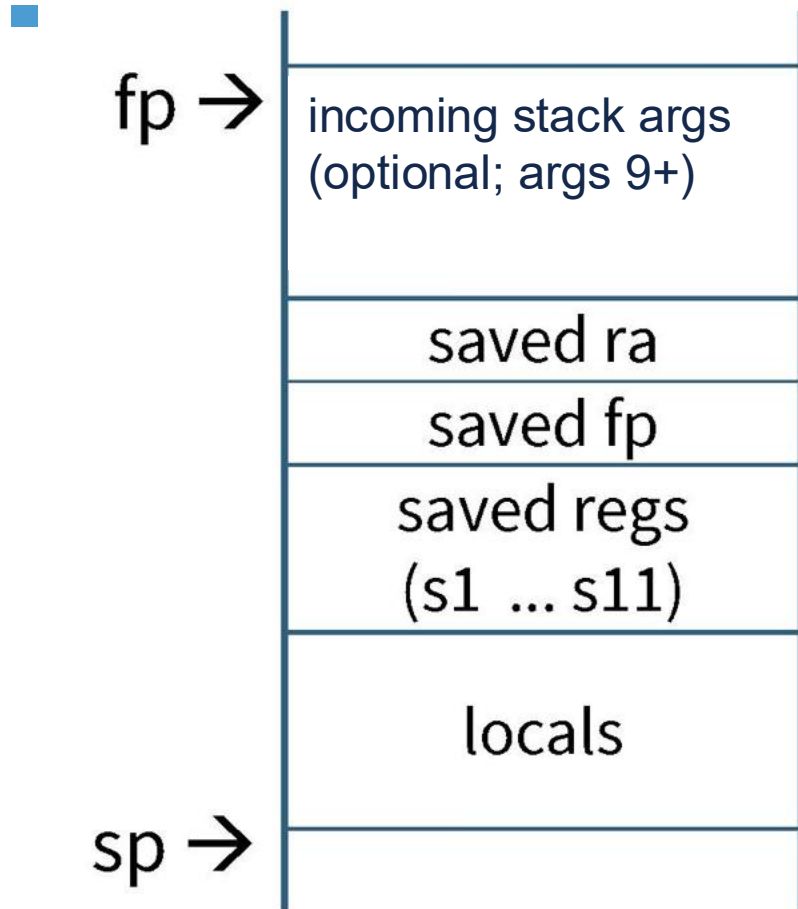


Code interacts with the stack in 3 main ways



- At the beginning of the function, it will “*push* a stack frame onto the call stack” by moving `sp` downward to make space for its own stack frame.
- During the execution of the function, it will use (positive) offsets from `sp` or `fp` to locate each of its local variables.
- At the end of the function, before it returns, it will “*pop* the stack frame off the call stack” by moving `sp` back up to wherever it used to be, “destroying” its stack frame.

RISC-V Calling Convention Cheat Sheet/Summary



- *first eight* args passed in registers `a0`, `a1`, ..., `a7`
- Space for args passed in *child's* stack frame
- return value (if any) in `a0`, `a1`
- stack frame at `sp`
 - contains `ra` (clobbered on JAL to sub-functions)
 - contains `fp`
 - contains *local vars* (possibly clobbered by sub-functions)
 - contains *space for incoming args*
- *Saved registers* (callee save regs) are *preserved*
- *Temporary registers* (caller save) regs are *not*

RISC-V Registers

- Return address: `x1` (`ra`)
- Stack pointer: `x2` (`sp`)
- Frame pointer: `x8` (`fp/s0`)
- First eight arguments: `x10-x17` (`a0-a7`)
- Return result: `x10-x11` (`a0-a1`)
- Callee-save free regs: `x18-x27` (`s2-s11`)
- Caller-save free regs: `x5-x7`, `x28-x31` (`t0-t6`)
- Global pointer: `x3` (`gp`)
- Thread pointer: `x4` (`tp`)

Source:

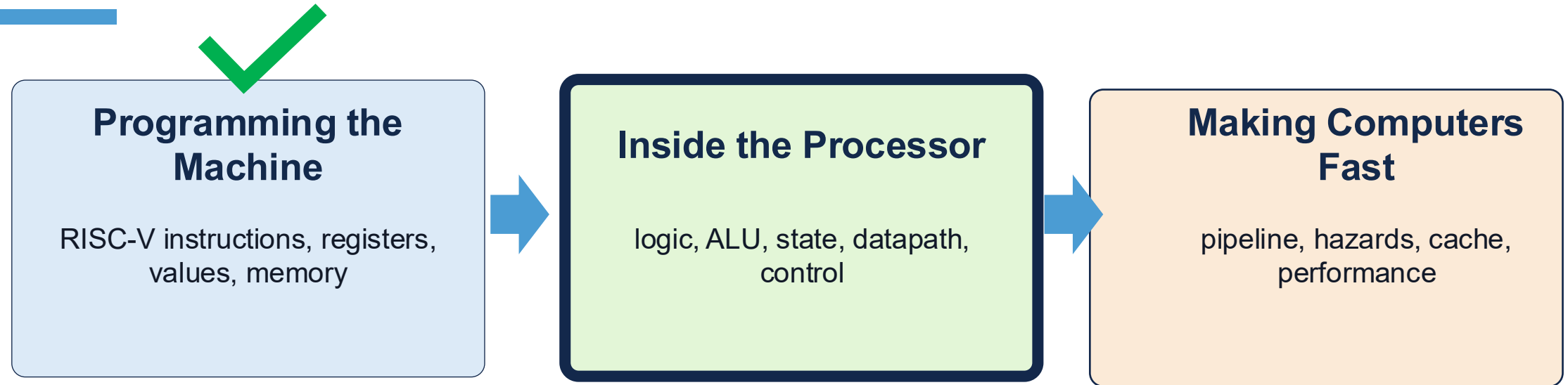
<https://www.cs.cornell.edu/courses/cs3410/2025fa/notes/asm-call.html>

Conclusions



- Need a convention (contract) between caller and callee
- Implement stack for storing each procedure's variables
- Procedure calls can now be arbitrarily nested
 - Recursion possible too
- FOLLOW the convention meticulously when programming!

Onto the next chapter!! 😊



- We now know how to program in assembly, according to the ISA.
 - How can hardware actually make that happen?
- We'll build up from logic + state to the ALU, data path, and control.
- **Goal:** connect each assembly instruction to the hardware actions needed to execute it.



UNIT 2: INSIDE THE PROCESSOR

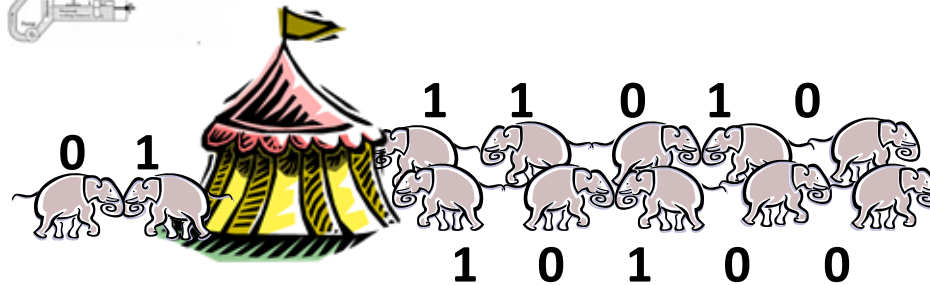
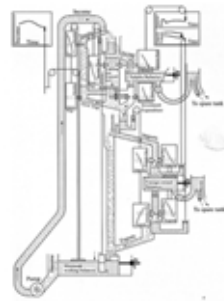
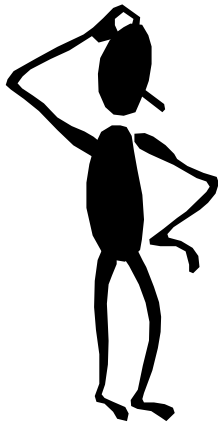
Basic Circuits

A Substrate for Computation

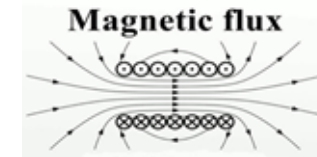


- We can build devices for processing and representing bits using almost any physical phenomenon

Wait! Some of those
might have potential...



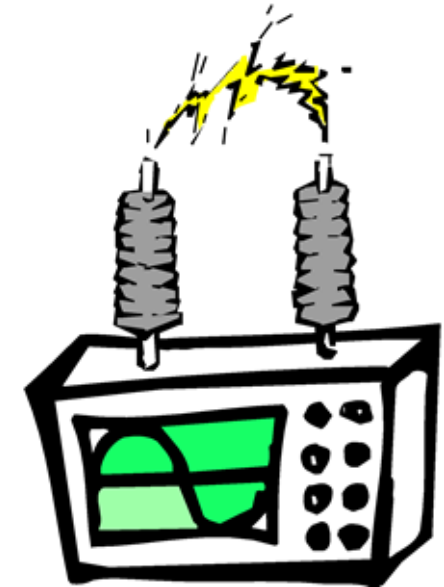
- magnetic flux
- trained elephants
- falling water
- turning gears
- DNA sequences
- polarization of a photon



Using Electromagnetic Phenomena



- Some EM things we could encode bits with:
 - *voltages, phase, currents, frequency*
- With today's technologies **voltages** are most often used.
- Voltage pros:
 - *easy generation, detection*
 - *voltage changes can be very fast*
 - *lots of engineering knowledge*
- Voltage cons:
 - *easily affected by environment*
 - *DC connectivity required?*
 - *R & C effects slow things down*



Voltage and Current



- Voltage
 - The force that makes electrons flow
 - Measured in Volts (V)
- Current
 - The rate of flow of electrons
 - Measured in Amperes (A)



Representing Information with Voltages



Representation of each point (x, y) in a B&W Picture:

0 volts: BLACK
1 volt: WHITE
0.37 volts: 37% Gray
etc.



Representation of a picture:

Scan points in some prescribed
raster order... generate voltage
waveform

**How much
information
at each point?**

Information Processing = Computation



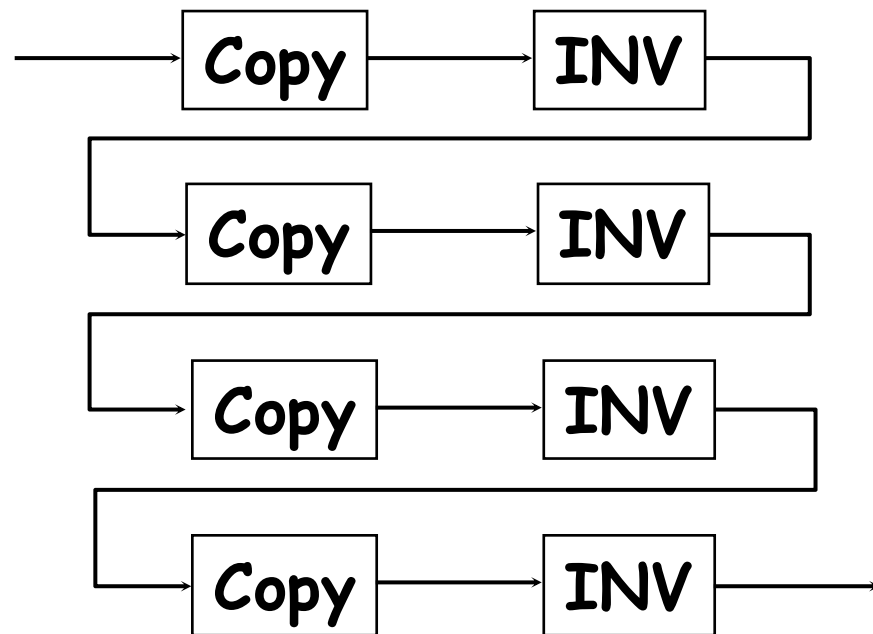
First, let's consider some processing blocks:



Let's Build a System!



input



(Reality)



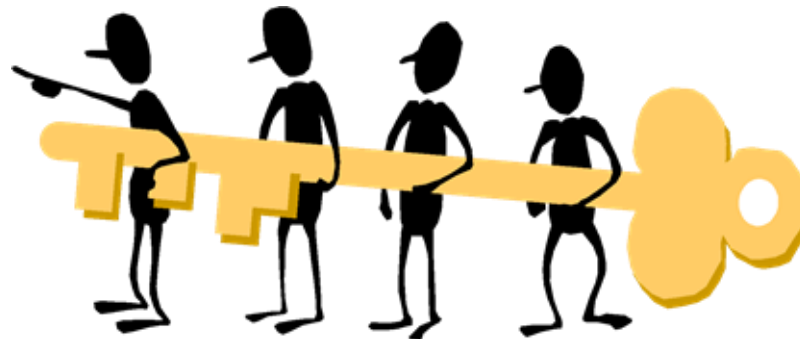
output

Contracts: Key to System Design



A SYSTEM is a structure that is “guaranteed” to exhibit a specified behavior, assuming all of its components obey their specified behaviors.

How is this achieved?



Through Contracts

Every system component will have clear obligations and responsibilities. If these are maintained we have every right to expect the system to behave as planned. If contracts are violated all bets are off.

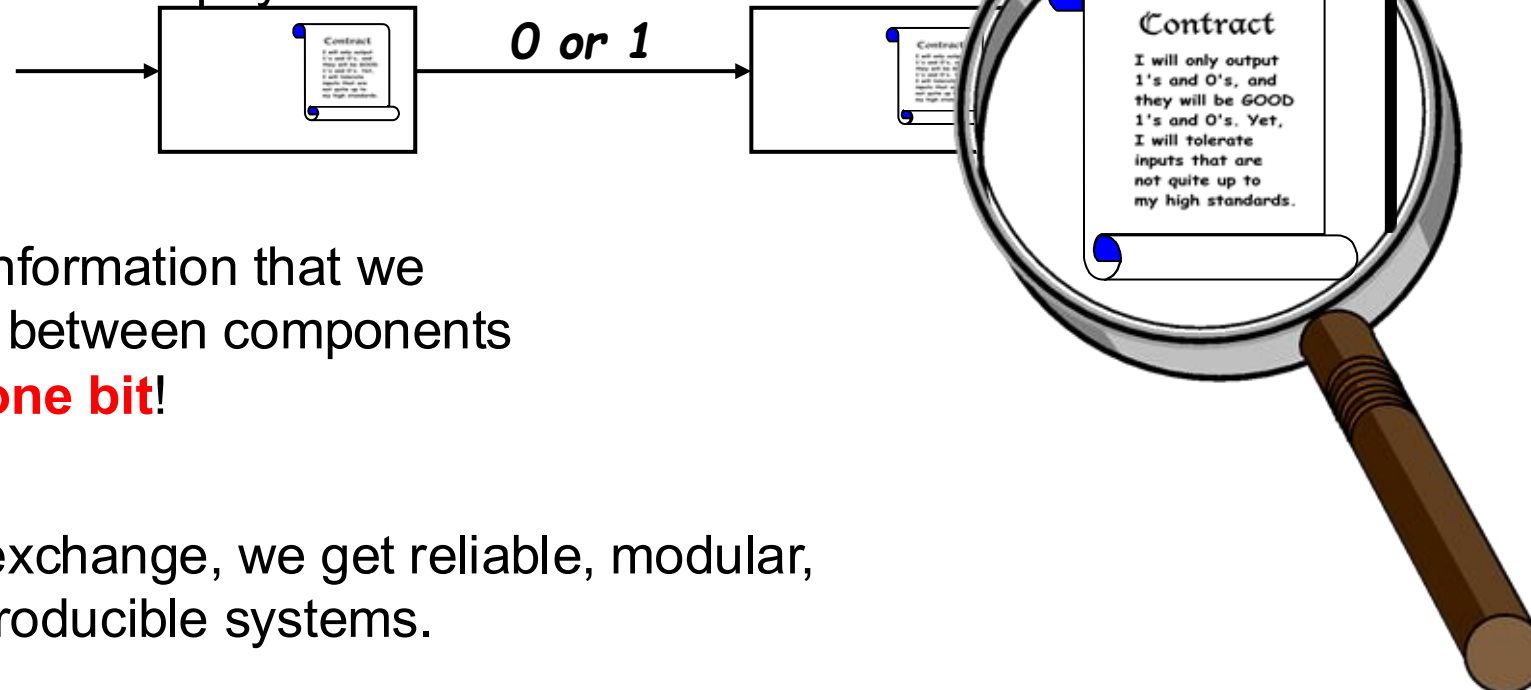
Digital Contracts



Why DIGITAL?

... because it keeps the contracts SIMPLE!

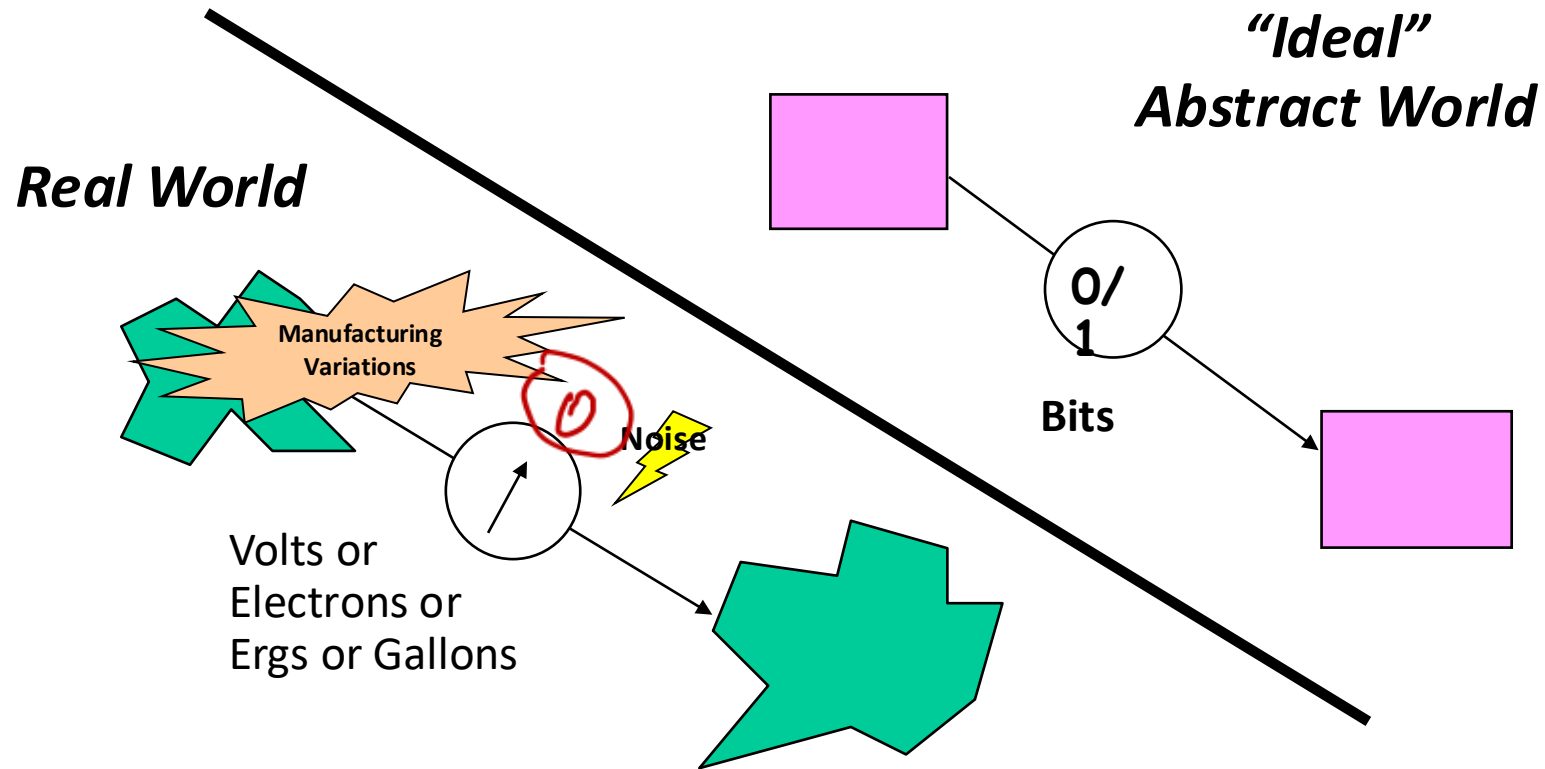
It's the price we pay for this robustness?



All the information that we transfer between components is only **one bit**!

But, in exchange, we get reliable, modular, and reproducible systems.

The Digital Abstraction



Keep in mind, **the real world is not digital, we engineer it to behave that way.**
We **coerce** real physical phenomena to implement digital designs!

Nose Margins



- Key idea:
Don't allow "0" to be mistaken for a "1" or vice versa
- Use the same "uniform bit-representation convention", for every component in our digital system
- To implement devices with high reliability, we outlaw "close calls" via a representation convention which forbids a range of voltages between "0" and "1".
- Ensure the valid input range is more tolerant (larger) than the valid output range

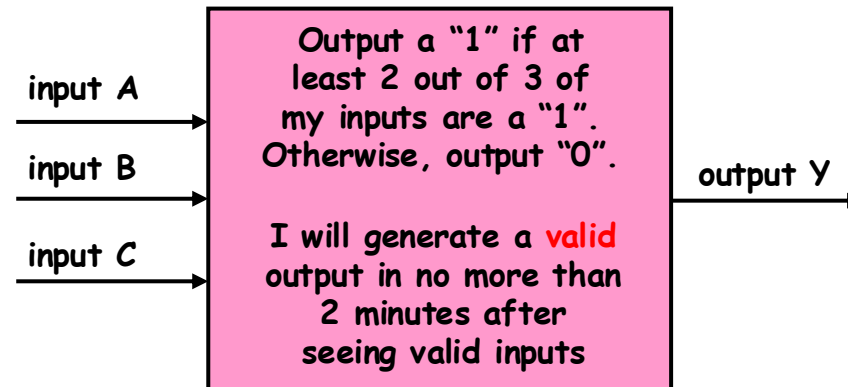


Our definition of valid does not preclude inputs and outputs from passing through invalid values. In fact, they must, but only during transitions. Our specifications allow for this (i.e. outputs are specified sometime (T_{pd}) after after inputs become valid).

A Digital Processing Element



- A *combinational device* is a digital element that has
 - one or more digital inputs *0 / 1*
 - one or more digital outputs
 - a functional specification that details the value of each output for *every possible combination of valid input values*
 - a timing specification consisting (at a minimum) an upper bound propagation delay, t_{pd} , on the required time for the device to compute the specified *valid* output values from an arbitrary set of stable, *valid* input values



A Combinational Digital System



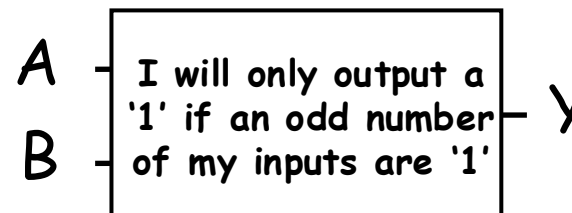
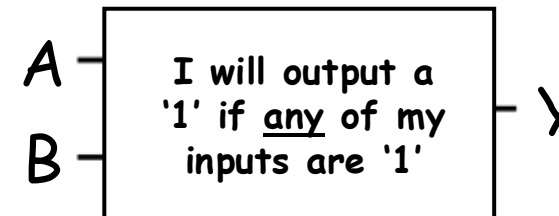
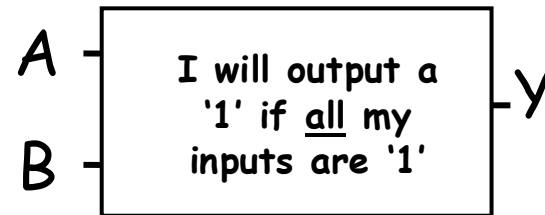
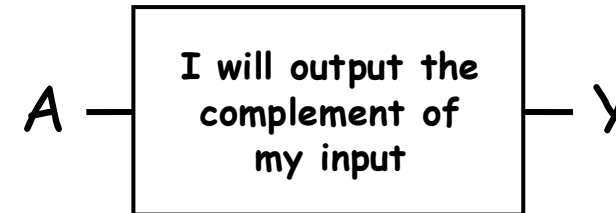
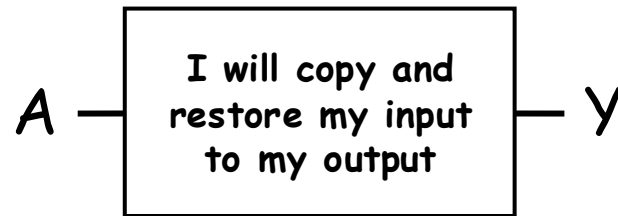
A system of interconnected elements is combinational if:

- Each circuit element is combinational
- Every input is connected to exactly one output or directly to some source of 0's or 1's
- The circuit contains no directed cycles
- Adheres to some noise margin threshold → AKA we can expect good 0s and good 1s! 😊

Digital Processing Elements (Gates)



Some digital processing elements occur so frequently that we give them special names and symbols



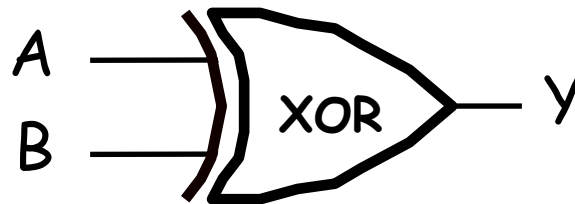
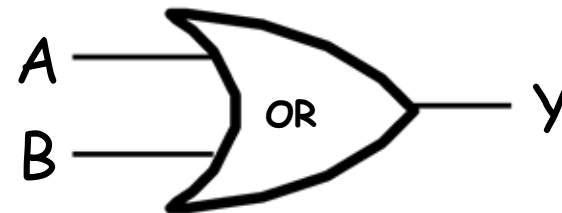
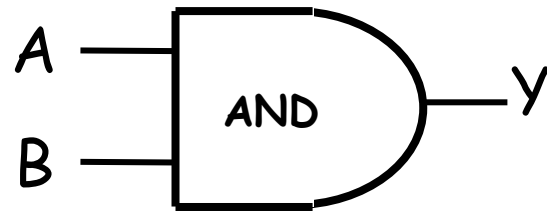
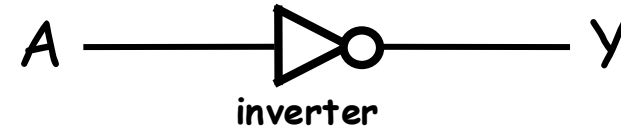
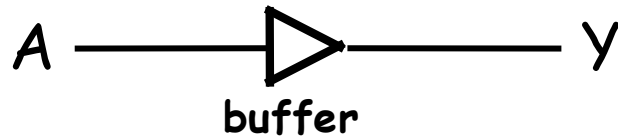
Q: What is the point of a buffer?
Doesn't a wire do the same thing?
A: A buffer restores marginal digital signals, because the output is as good or "better" than the input (i.e. it solves that bad image problem from slide 7).



Digital Processing Elements (Gates)



Some digital processing elements occur so frequently that we give them special names and symbols

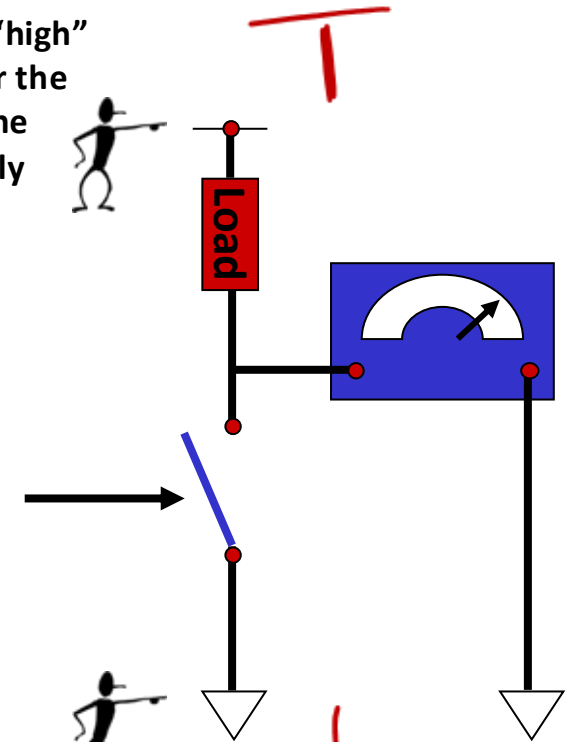


How do we make gates?



- A controllable switch is the common link of all computing technologies
- Switches control voltages by creating and opening paths between higher and lower potentials
- Closed circuit:
 - *fully connected, allows electricity to flow uninterrupted*
- Open circuit
 - *A circuit that contains a broken connection*
 - *Electricity stops flowing at the point where the connection was lost*

This symbol indicates a “high” potential, or the voltage of the power supply

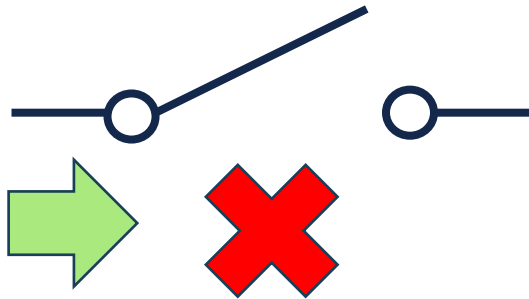


This symbol indicates a “low” or ground potential

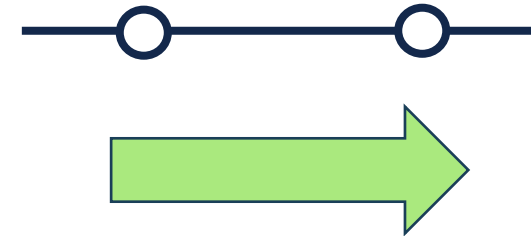
Switches



Open (off)

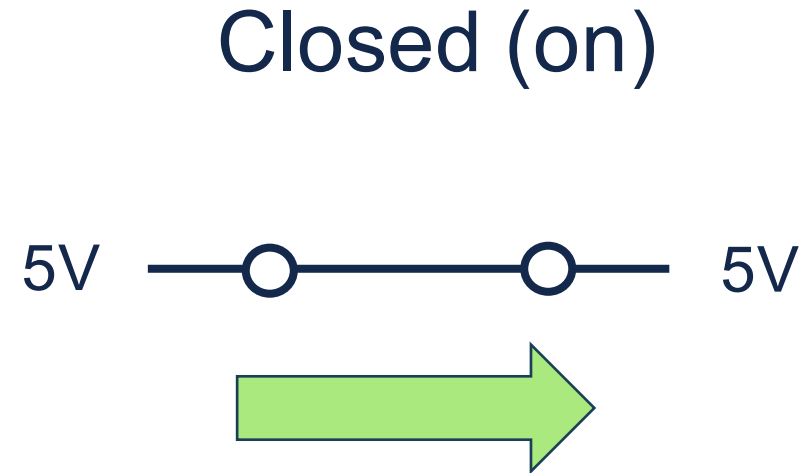
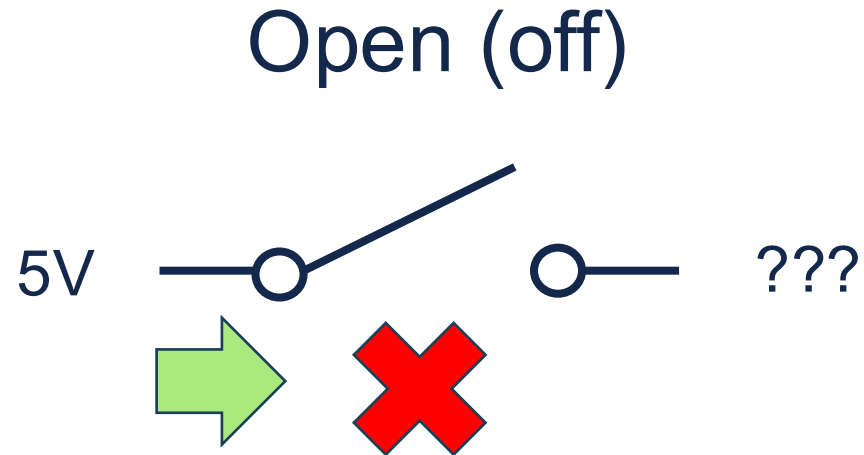


Closed (on)



Electricity can flow through a closed switch, it cannot flow through an open switch

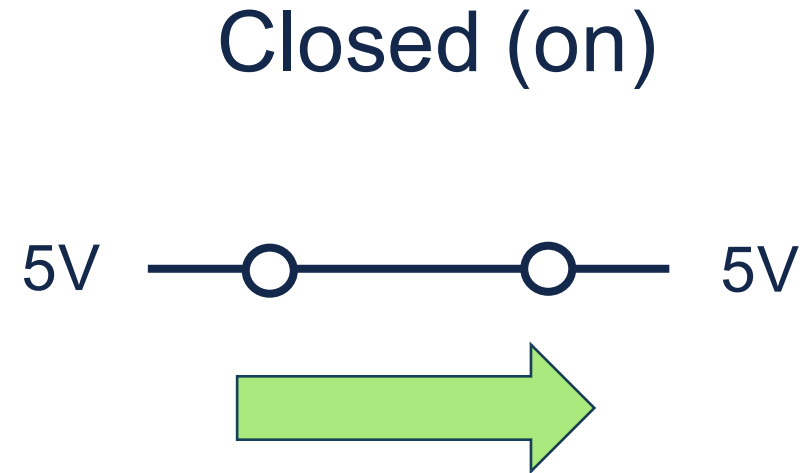
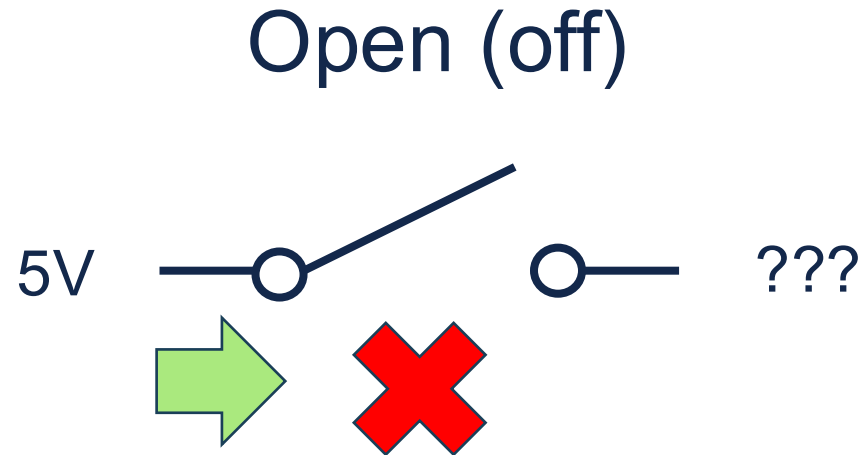
Switches



Let's say we apply 5V to the left side of the switch

- If the switch is open, we don't necessarily know the voltage on the right
 - Would need to see the full circuit to determine the voltage
- If the switch is closed, the voltage on the right is 5V

Switches



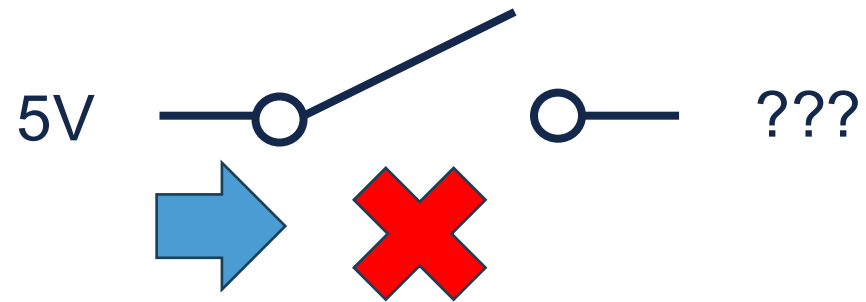
Let's say we apply 0V to the left side of the switch

- If the switch is open, we don't necessarily know the voltage on the right
 - Would need to see the full circuit to determine the voltage
- If the switch is closed, the voltage on the right is 0V

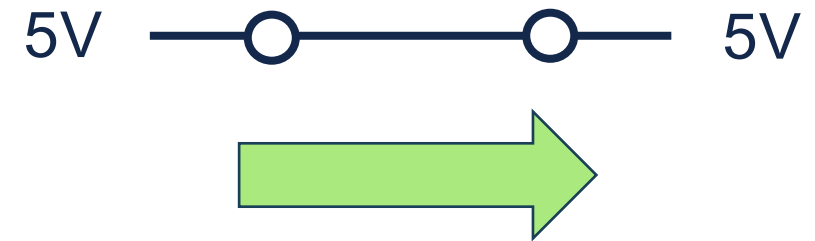
Switches



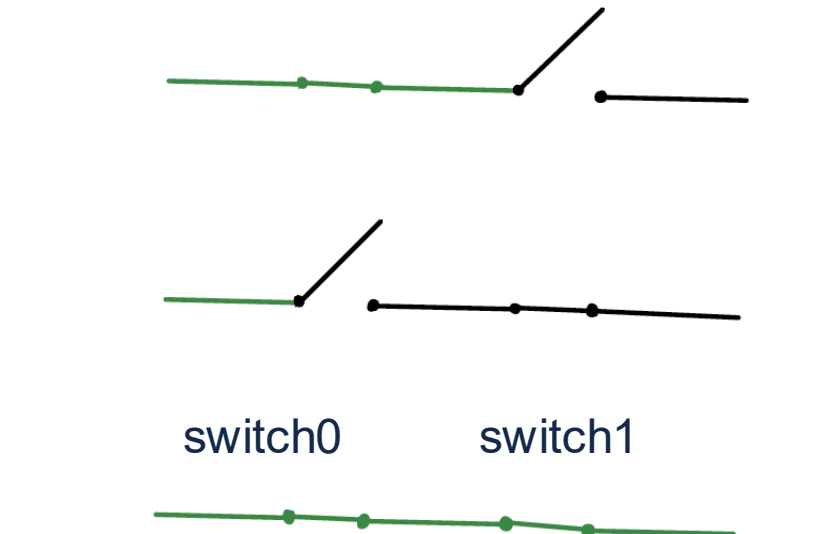
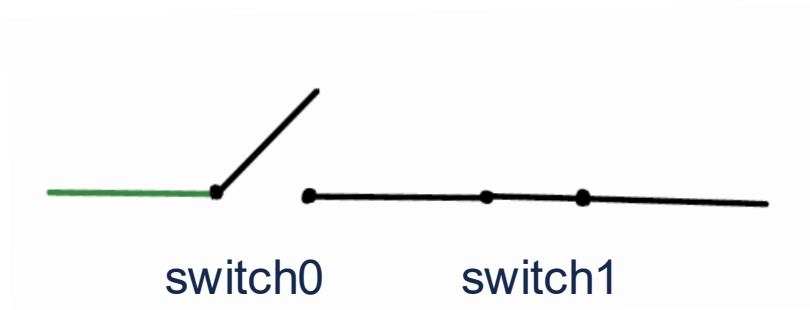
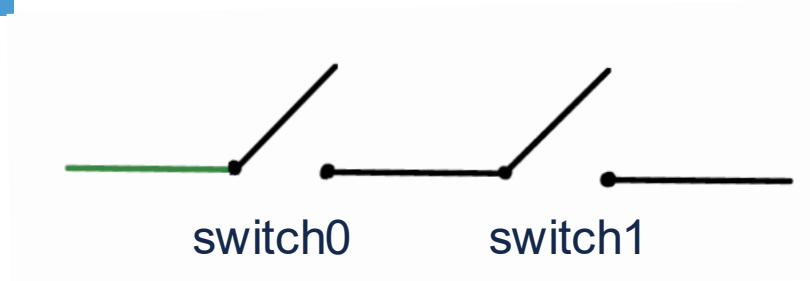
Open (off)



Closed (on)



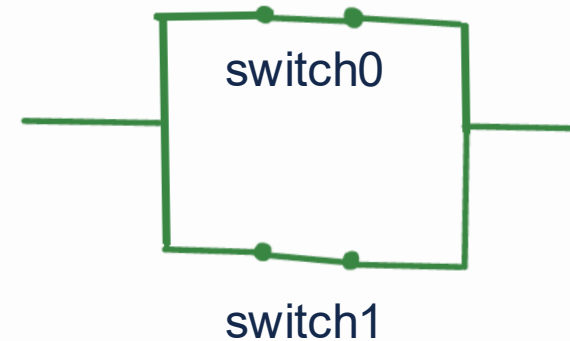
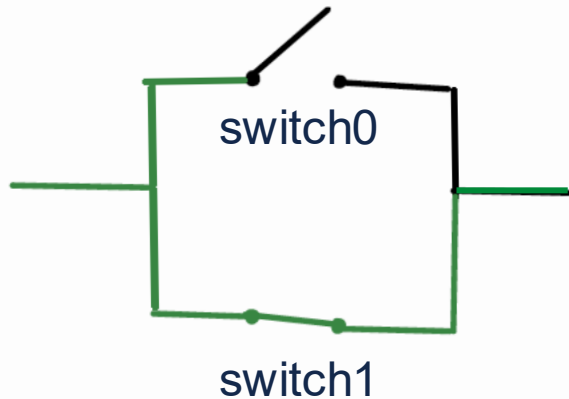
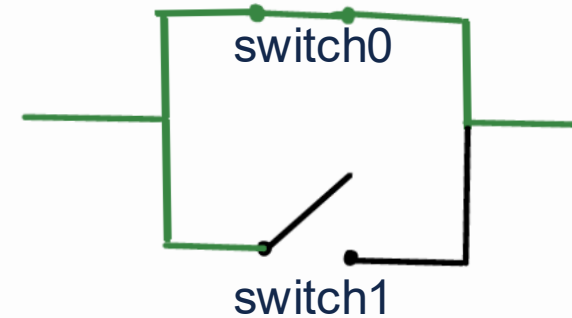
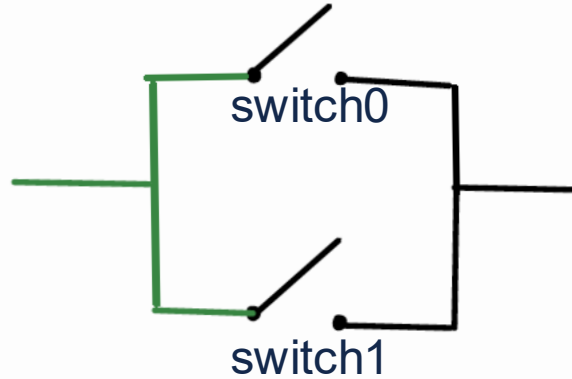
Switches in Series



Current can only flow between the two ends if both switches are closed

$$\text{current_flow} = \text{switch}_0 \text{ AND } \text{switch}_1$$

Switches in Parallel



Current can flow between the two ends if either of the switches is closed
 $\text{current_flow} = \text{switch}_0 \text{ OR } \text{switch}_1$

UNIT 1: PROGRAMMING THE MACHINE

Transistors 😊

Transistors are the physical building blocks of modern computers!



Number of Transistors in CPUs over Time



A transistor is the physical device we use to turn continuous electrical behavior into digital 0s and 1s.

You can think of it as an electrically controlled switch that turns current on or off.

Transistors are like switches



Switches are controlled by **physical contact**

Transistors are controlled by a **voltage**

Two States: Binary!



- Transistors can be in two distinct states: on or off
- This is why our machines communicate in binary (1s and 0s)!

On



Off



Logic Levels



- Instead of specifying the exact voltage level (e.g. 3V or 5V), we'll simply use binary terms when discussing transistors.
- We'll either have a **high voltage** or a **low voltage**.
- There are a few different terms for these voltage levels.
 - High voltage
 - logic high = logic 1 = 1
 - Low voltage
 - logic low = logic 0 = 0

Transistors

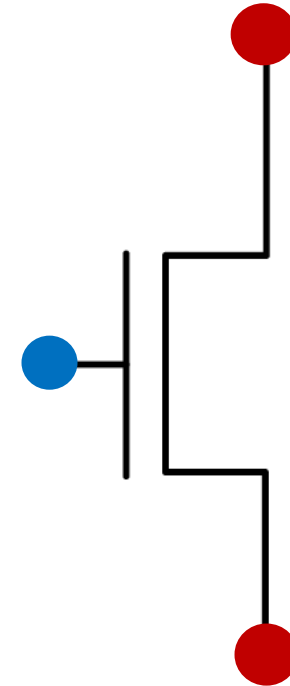


- We will be working with two types of transistors
 - nMOS transistors
 - pMOS transistors
- There are three terminals
 - **Gate**: controls whether the transistor is on or off
 - **Source** and **Drain**: endpoints

Transistor Symbol



Switch



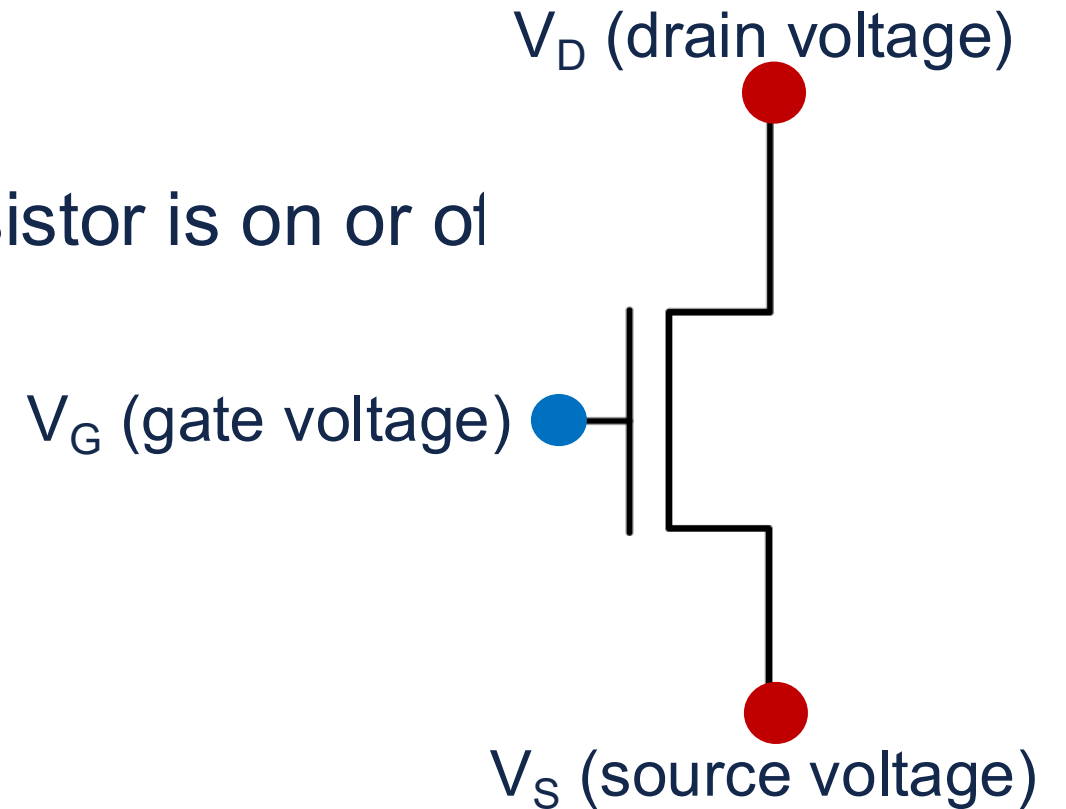
Transistor
(n-channel MOSFET)

The terminal labeled with the **blue** dot determines whether current can flow between the terminals labeled with the **red**

NMOS Transistor



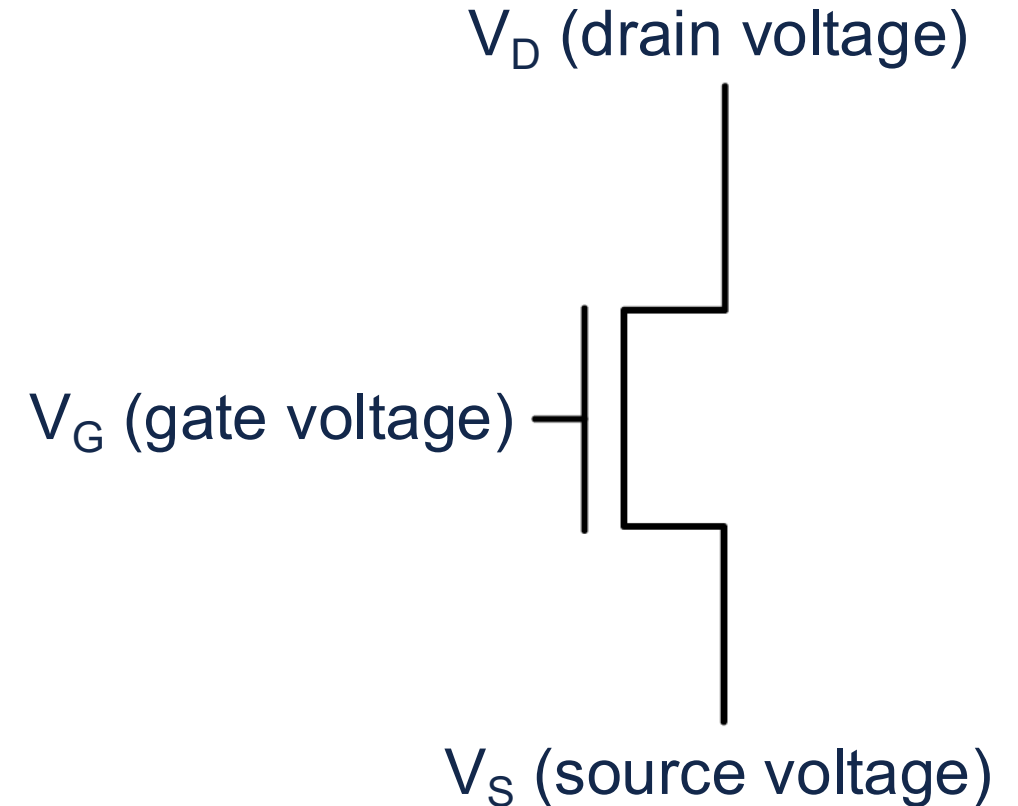
- There are three terminals
 - **Gate**: controls whether the transistor is on or off
 - **Source** and **Drain**: endpoints



NMOS Transistor



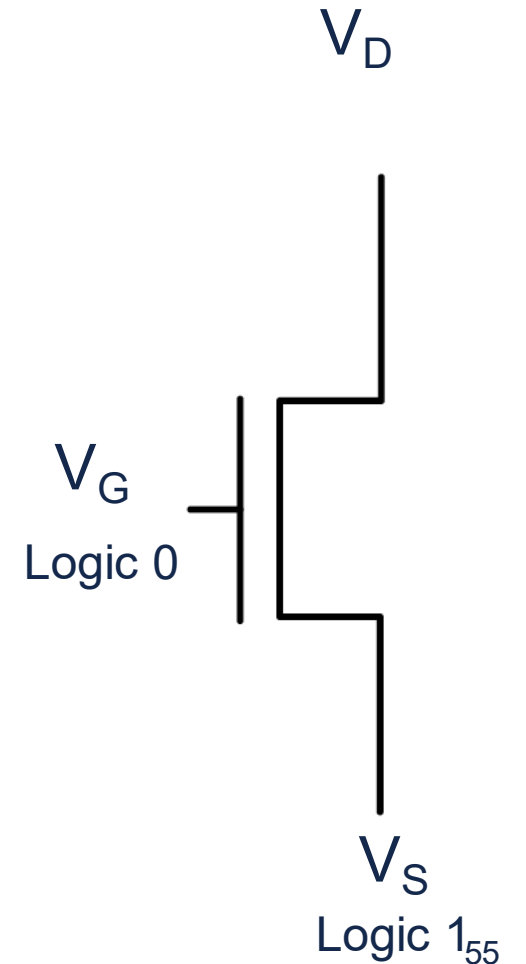
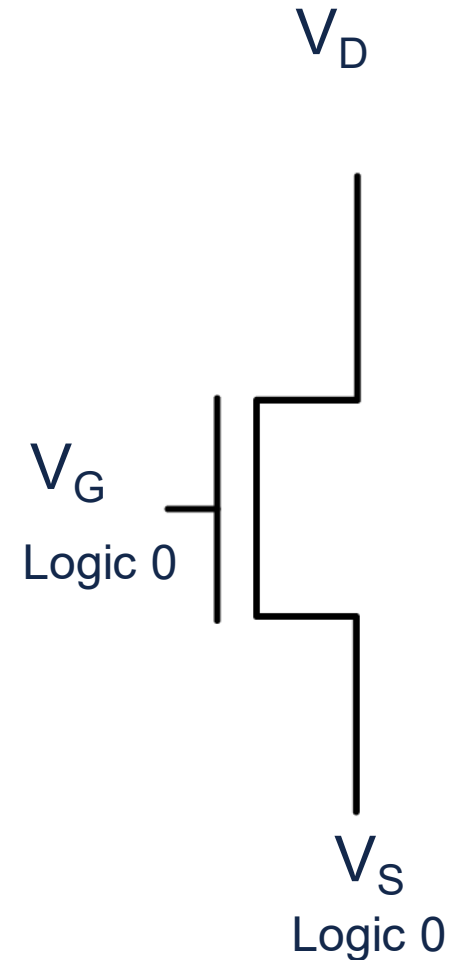
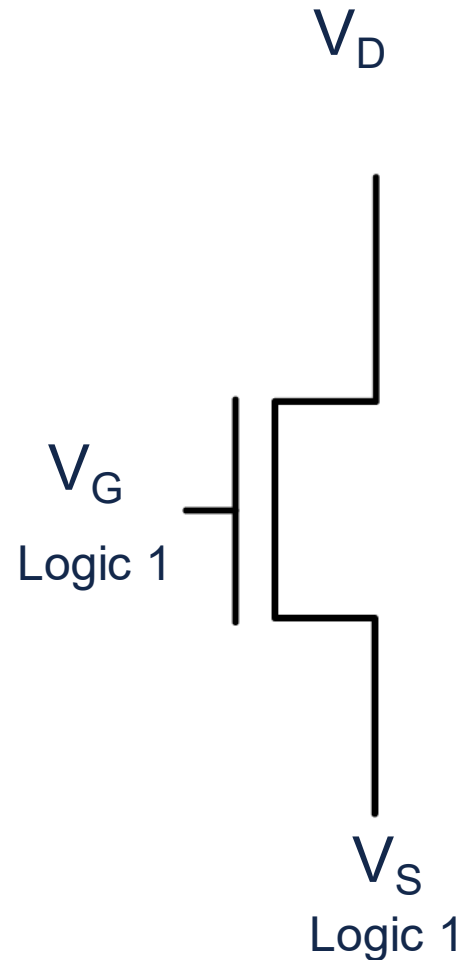
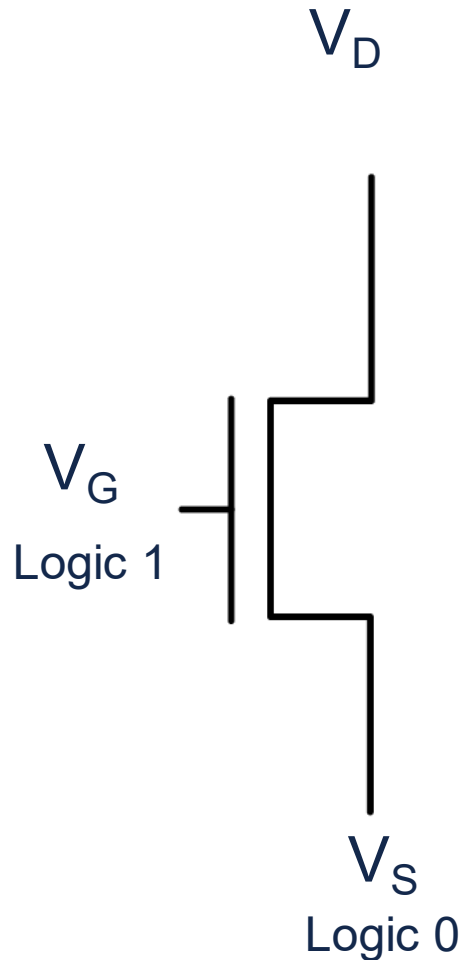
- If the gate voltage is **high (logic 1)**, current **can** flow between source and drain
 - The source and drain will be connected and will have the same voltage
- If the gate voltage is **low**, current **cannot** flow between source and drain
 - The source and drain are not connected



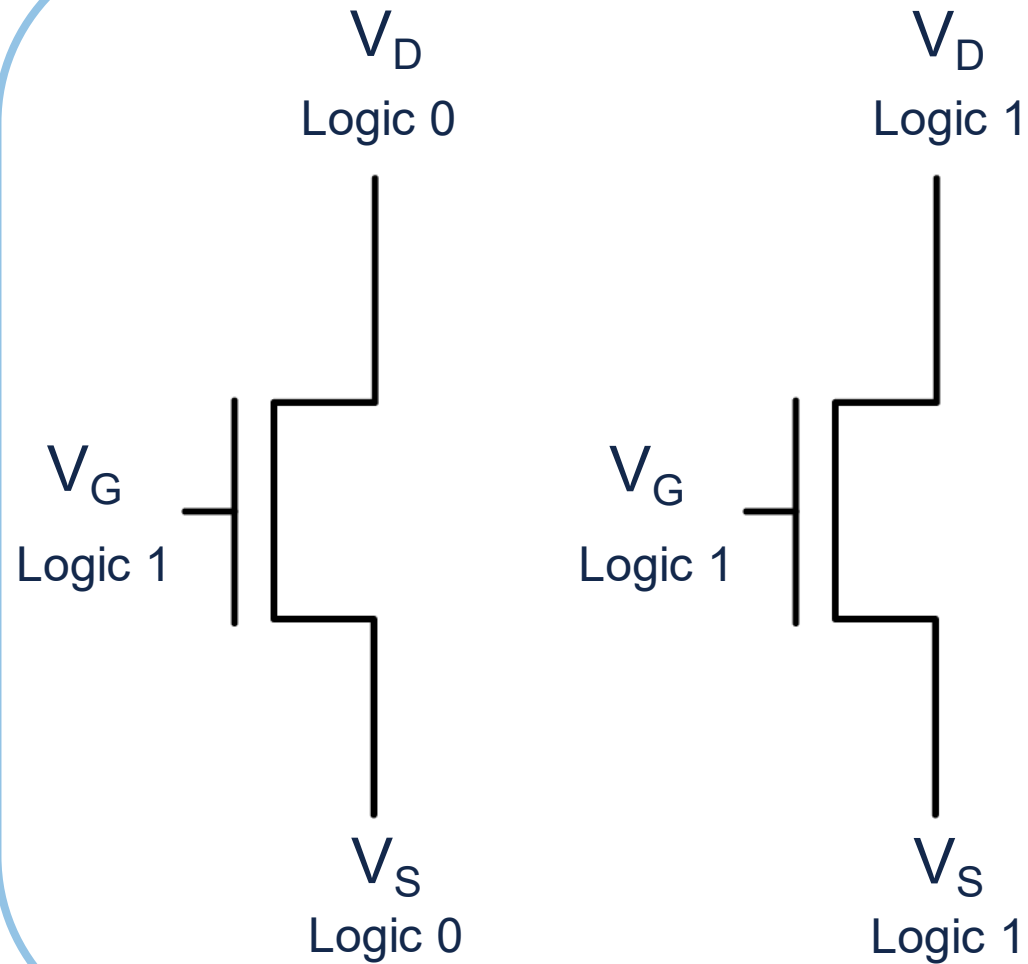
NMOS Transistor



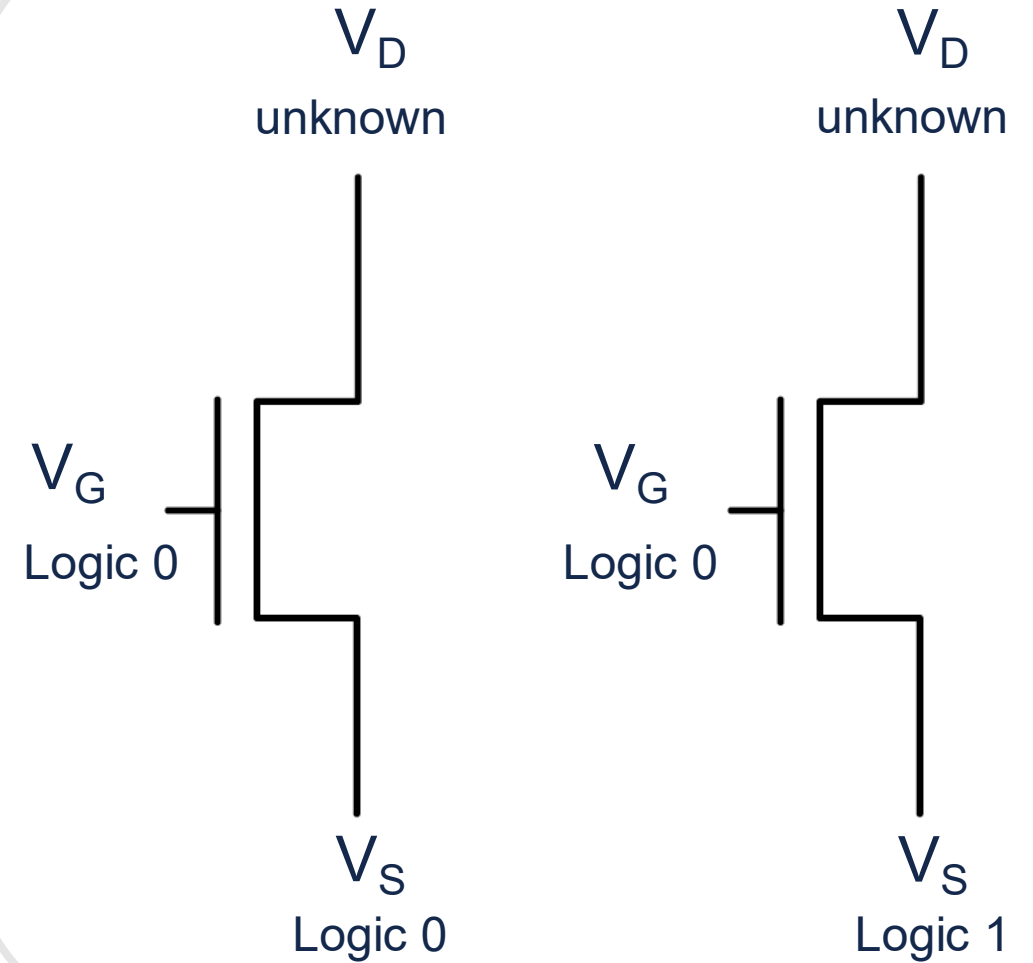
Let's say that we know the values of V_G and V_S . What is the value of V_D in each of the following diagrams?



NMOS Transistor



V_G is high, so source and drain are connected

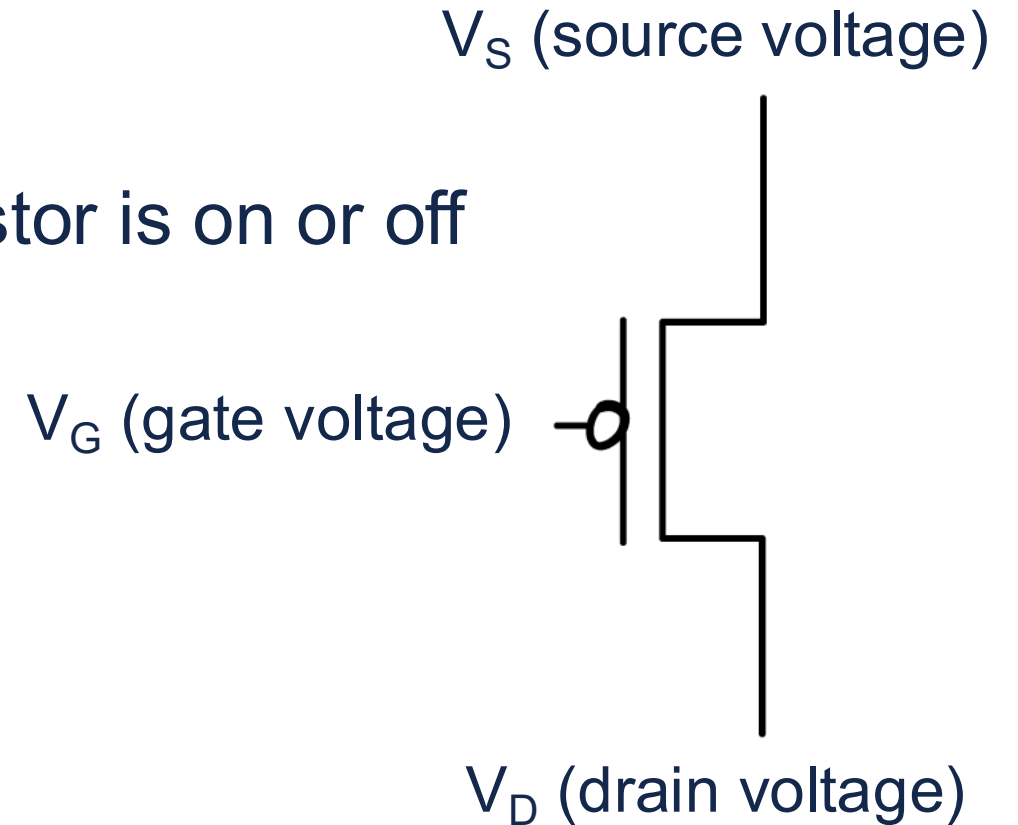


V_G is low, so source and drain are not connected

PMOS Transistor



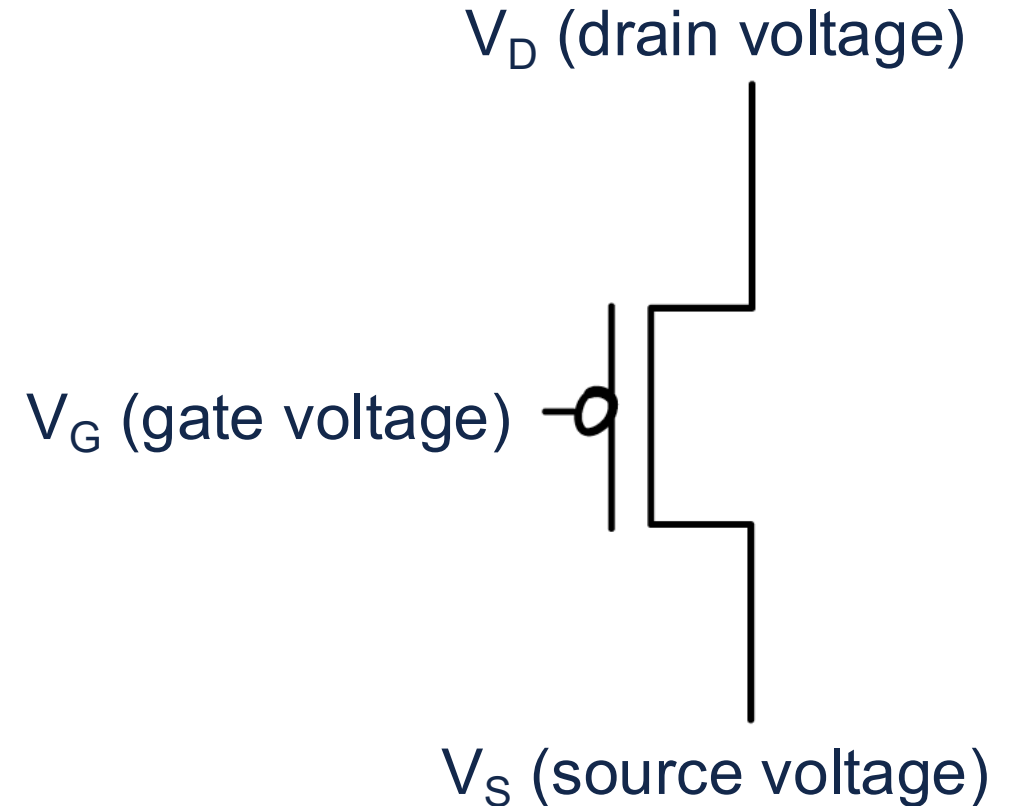
- There are three terminals
 - **Gate**: controls whether the transistor is on or off
 - **Source** and **Drain**: endpoints



PMOS Transistor



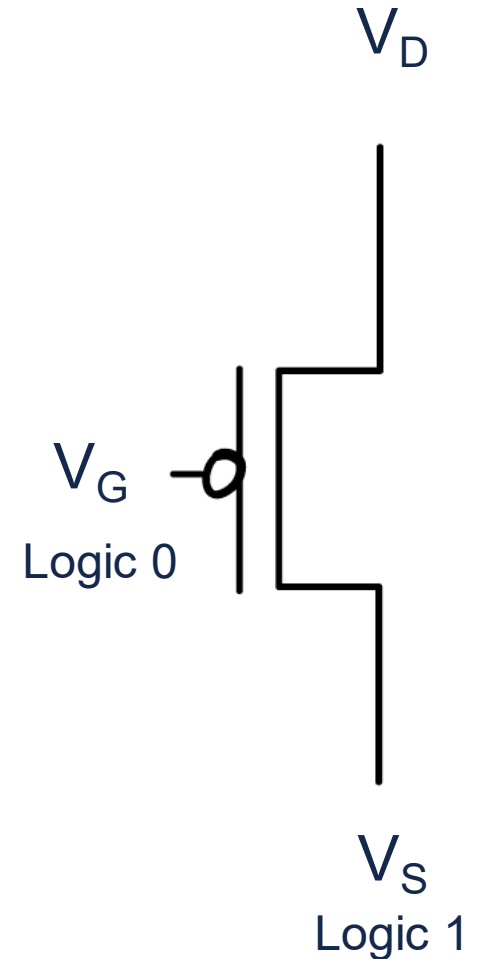
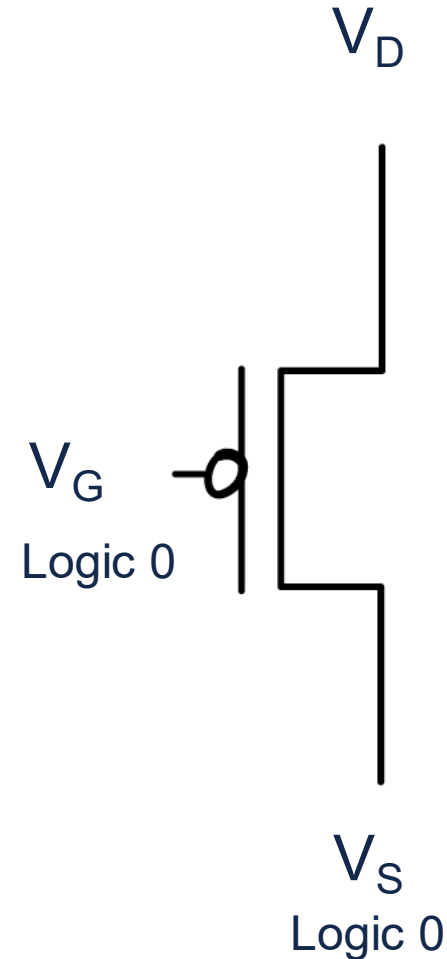
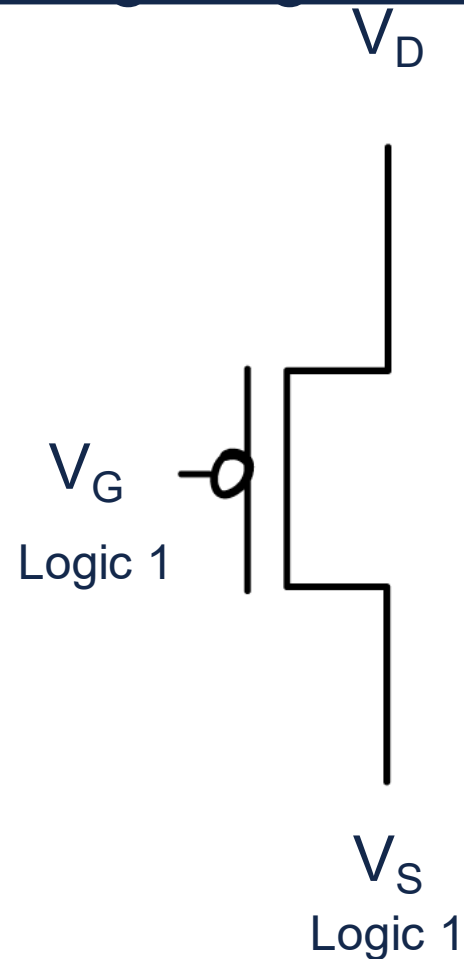
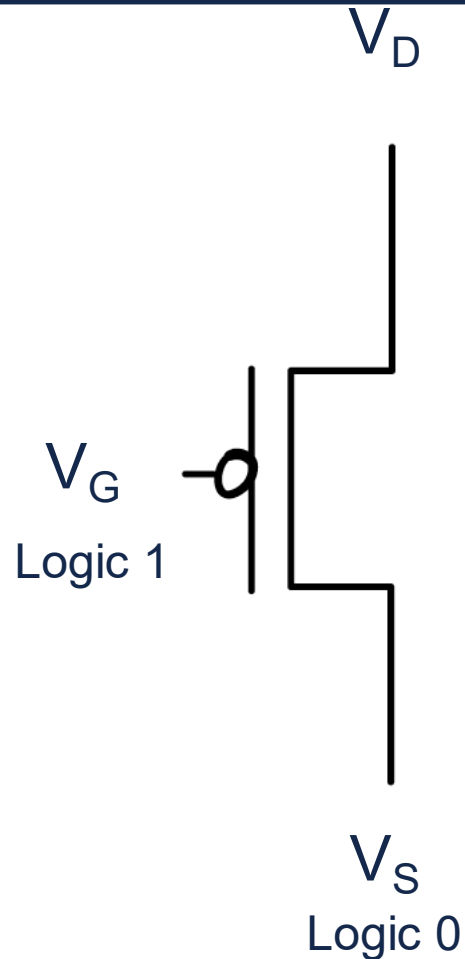
- If the gate voltage is **low (logic 0)**, current **can** flow between source and drain
 - This source and drain will be connected and will have the same voltage
- If the gate voltage is **high**, current **cannot** flow between source and drain
 - The source and drain are not connected



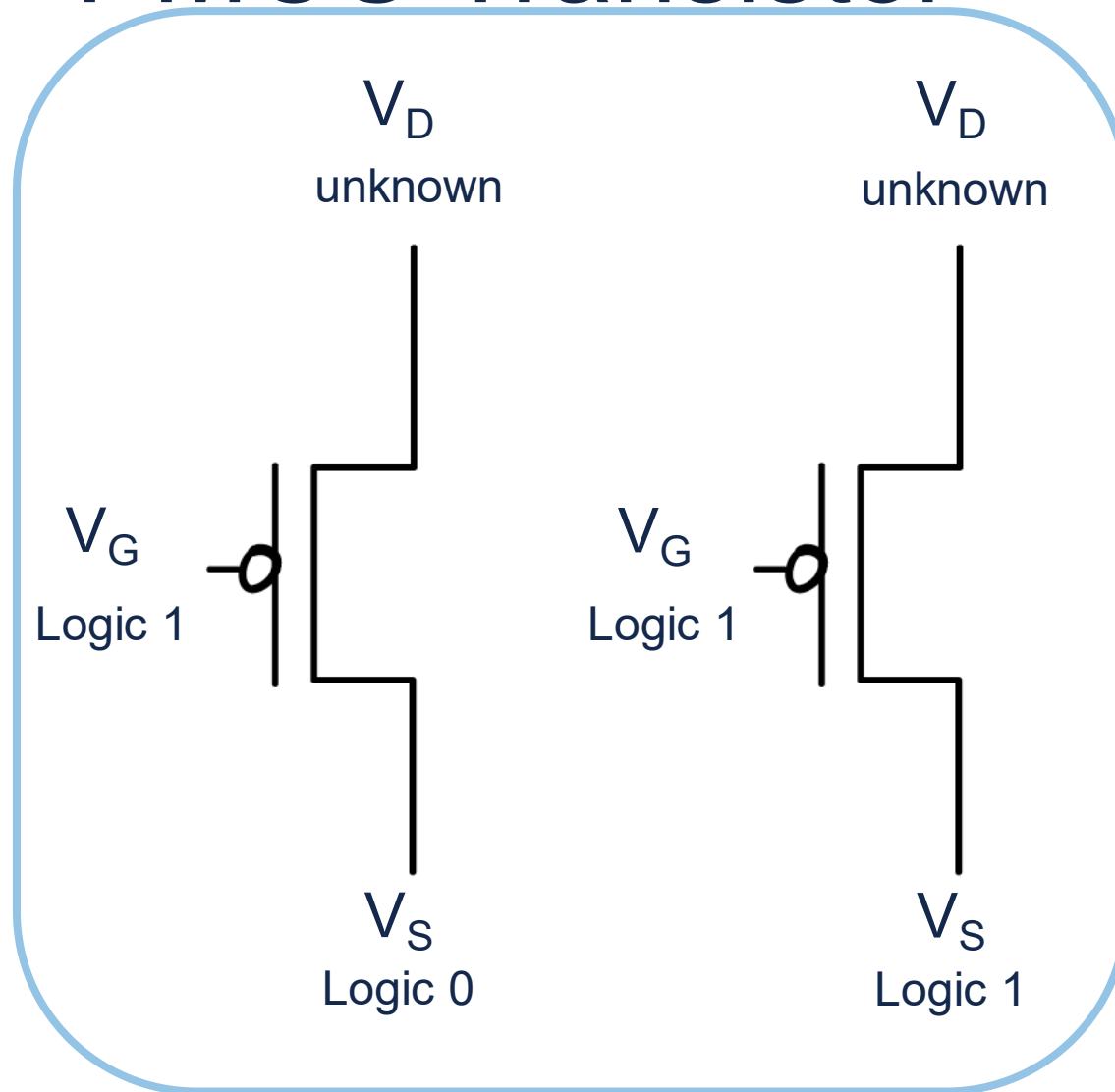
PMOS Transistor



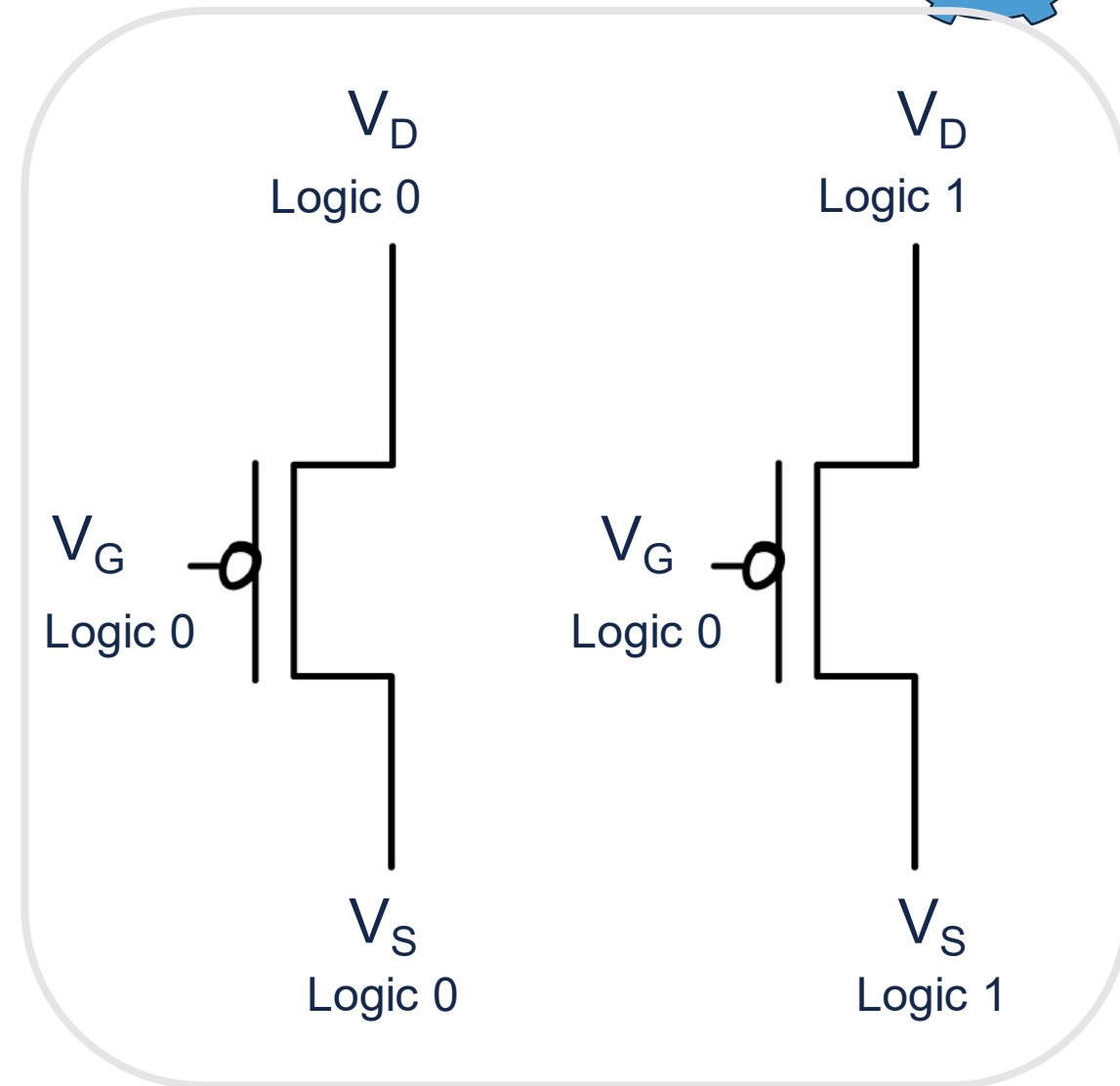
Let's say that we know the values of V_G and V_S . What is the value of V_D in each of the following diagrams?



PMOS Transistor



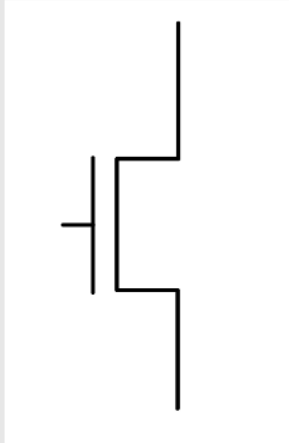
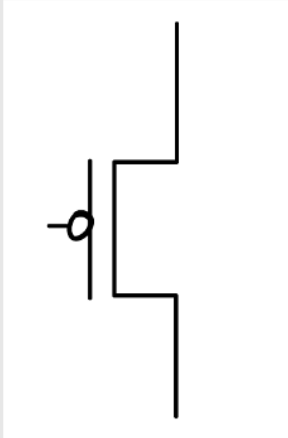
V_G is high, so source and drain are not connected



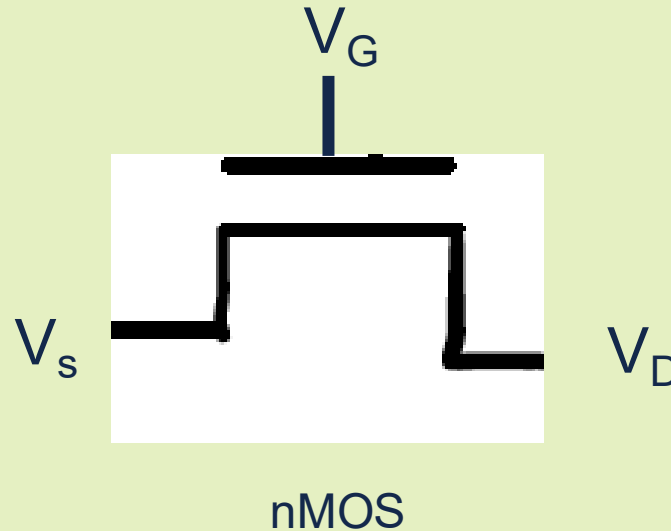
V_G is low, so source and drain are connected

Two Types of Transistors



	<u>nMOS</u>	<u>pMOS</u>
		
Behaves as an open switch when	V_g is low	V_g is high
Behaves as a closed switch when	V_g is high	V_g is low

nMOS Transistors



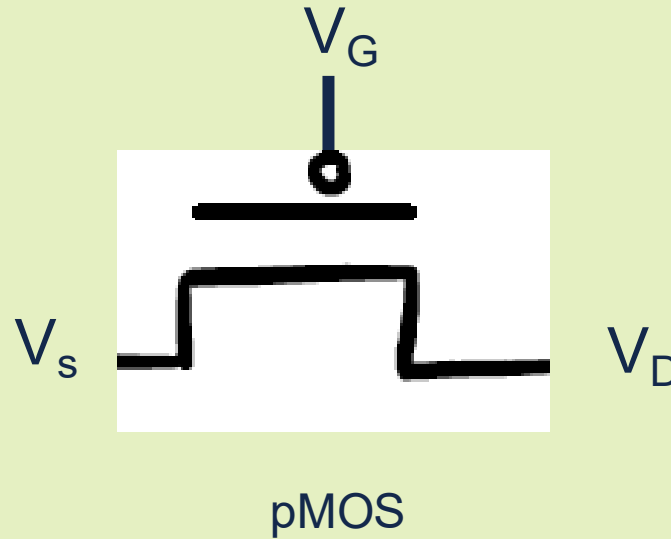
Acts as a(n) _____ (open/closed) switch when the gate voltage (V_G) is low.

Meaning that current _____ (can/cannot) flow between source and drain.

Acts as a(n) _____ (open/closed) switch when the gate voltage (V_G) is high.

Meaning that current _____ (can/cannot) flow between source and drain.

pMOS Transistors



Acts as a(n) _____ (open/closed) switch when the gate voltage (V_G) is low.

Meaning that current _____ (can/cannot) flow between source and drain.

Acts as a(n) _____ (open/closed) switch when the gate voltage (V_G) is high.

Meaning that current _____ (can/cannot) flow between source and drain.