CSE 390B, Autumn 2022

Building Academic Success Through Bottom-Up Computing

Final Project Overview & Operating Systems

Final Project Overview, The Software Stack, Overview of Operating Systems, Project 8 Overview

Lecture Outline

- **Final Project Overview**
  - **E-Portfolio Details and Topics Brainstorming**
- The Software Stack
  - Roadmap of Hardware and Software Components
- Overview of Operating Systems
  - Abstraction, Protection, Processes, Virtual Memory
- Project 8 Overview
  - Micro Jack Details, Tips for Getting Started

Final Project E-Portfolio Overview

- You will create an E-Portfolio that is geared toward a new Allen School student
- Your E-Portfolio is a culminating project in having you reflect on the **metacognitive skills** you've learned and **providing advice** for entering the program
- During our final class, you will give a 6–8-minute presentation on your E-Portfolio

Final Project Due Dates

- Part I: E-Portfolio Outline
  - Due next Thursday (12/8) at 11:59pm
- Part II: Final E-Portfolio
  - Due Tuesday of finals week (12/13) at 4:00pm
- Part III: E-Portfolio Presentations
  - During the scheduled CSE 390B final
  - CSE 390B Final Time: Tuesday, 12/13 from 4:30-6:30pm
  - CSE 390B Final Location: CSE2 G04 (same as usual classroom)

Reflection on Metacognitive Skills

Individually first, take some time to reflect on the following questions, and then discuss in groups:

- Which two metacognitive topics would you consider including in your E-Portfolio and why?
  - Reflect on which ones you've grown the most in, have impacted you the most, were most challenging to grow in, etc.
- What are some examples of yourself demonstrating those two metacognitive skills?
  - Please be specific here! Aim to share these skills as if you are telling a story and showing concrete applications of these skills

Reflection on a Technical Skill

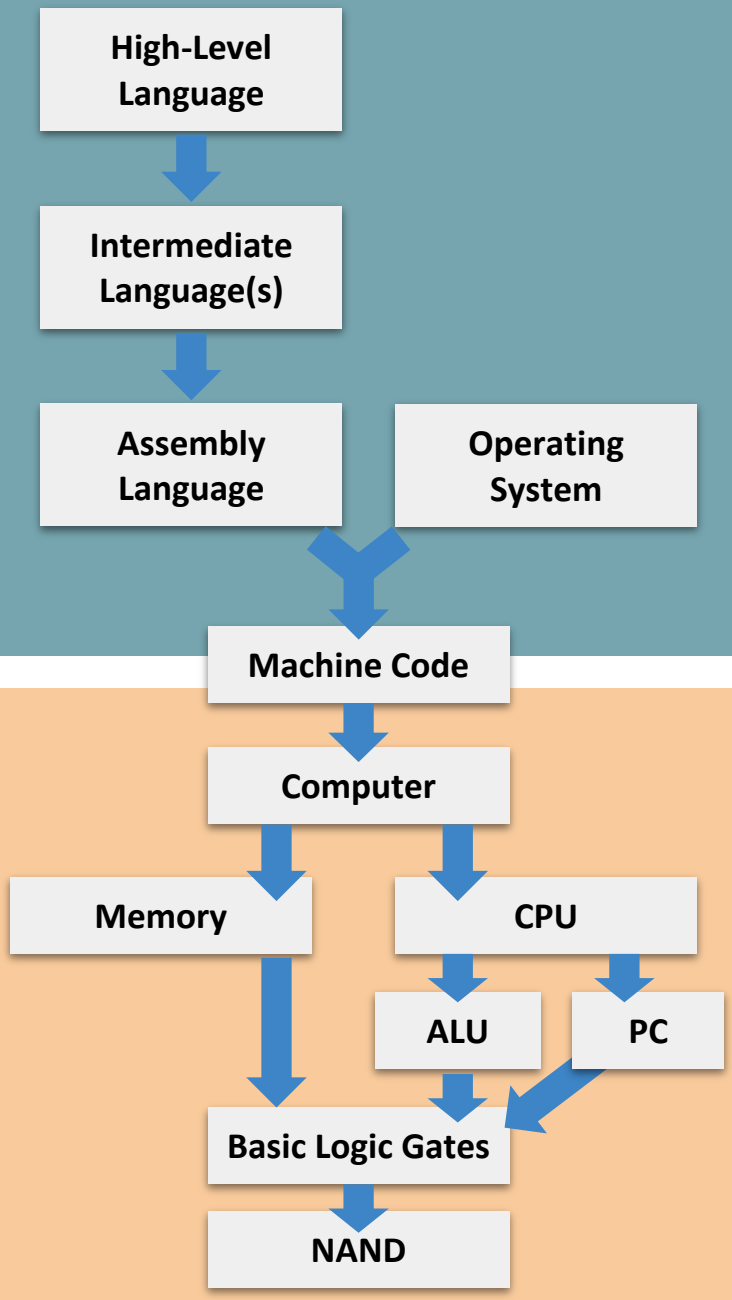
Individually first, take some time to reflect on the following questions, and then discuss in groups:

- What technical topic from CSE 390B would you consider including in your E-Portfolio and why?
  - Reflect on technical skills that helped connect the dots, were most interesting to you, most challenging for you to grasp, etc.
- What is the impact of having knowledge of that technical skill? In other words, why is that technical skill useful?
  - Please be specific here as well — think about how this technical skill would be useful in an academic or personal setting

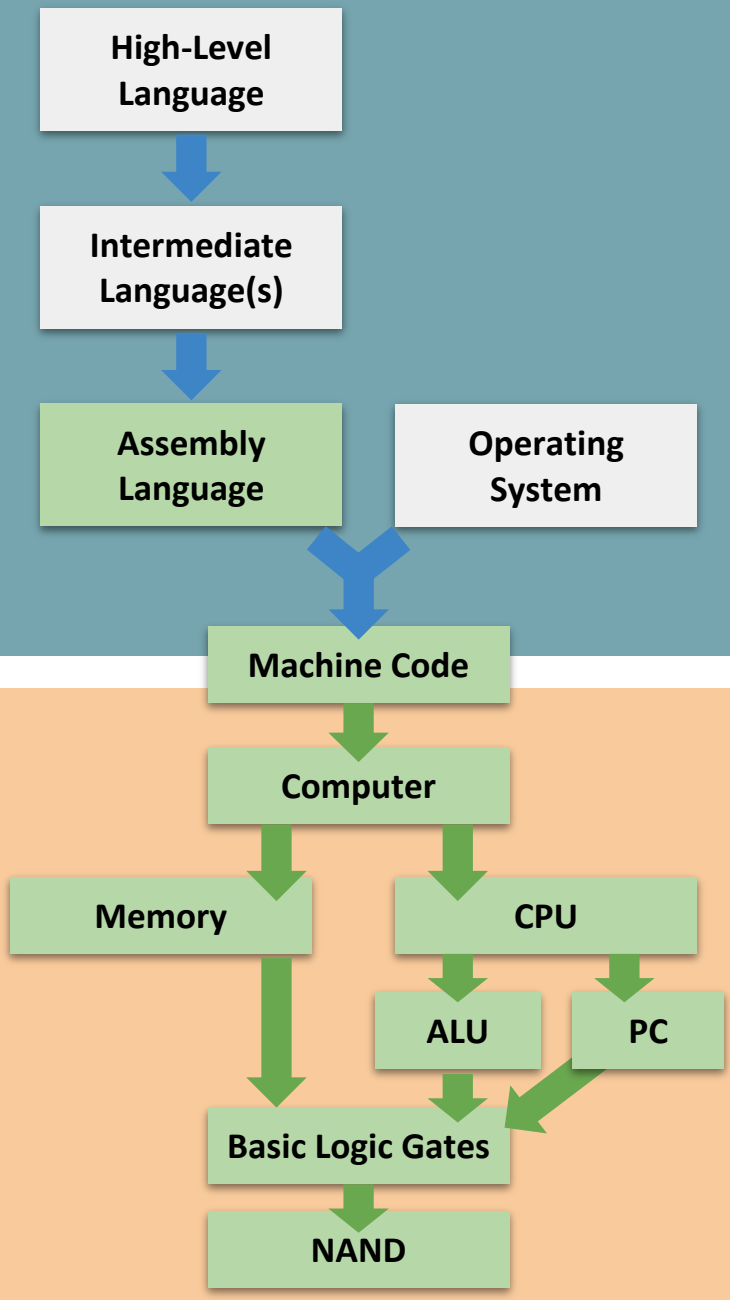
Lecture Outline

- Final Project Overview
  - E-Portfolio Details and Topics Brainstorming
- **The Software Stack**
  - **Roadmap of Hardware and Software Components**
- Overview of Operating Systems
  - Abstraction, Protection, Processes, Virtual Memory
- Project 8 Overview
  - Micro Jack Details, Tips for Getting Started

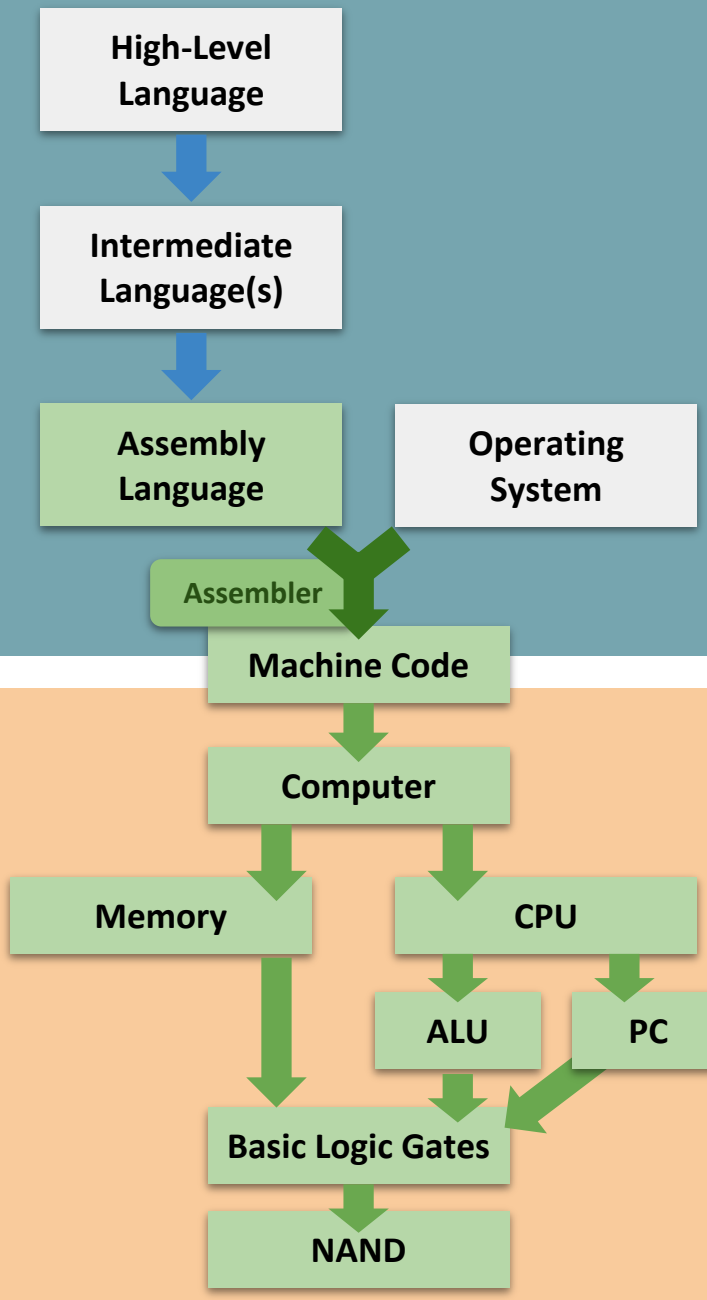
Roadmap


High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
SOFTWARE
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

Roadmap


High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
SOFTWARE
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

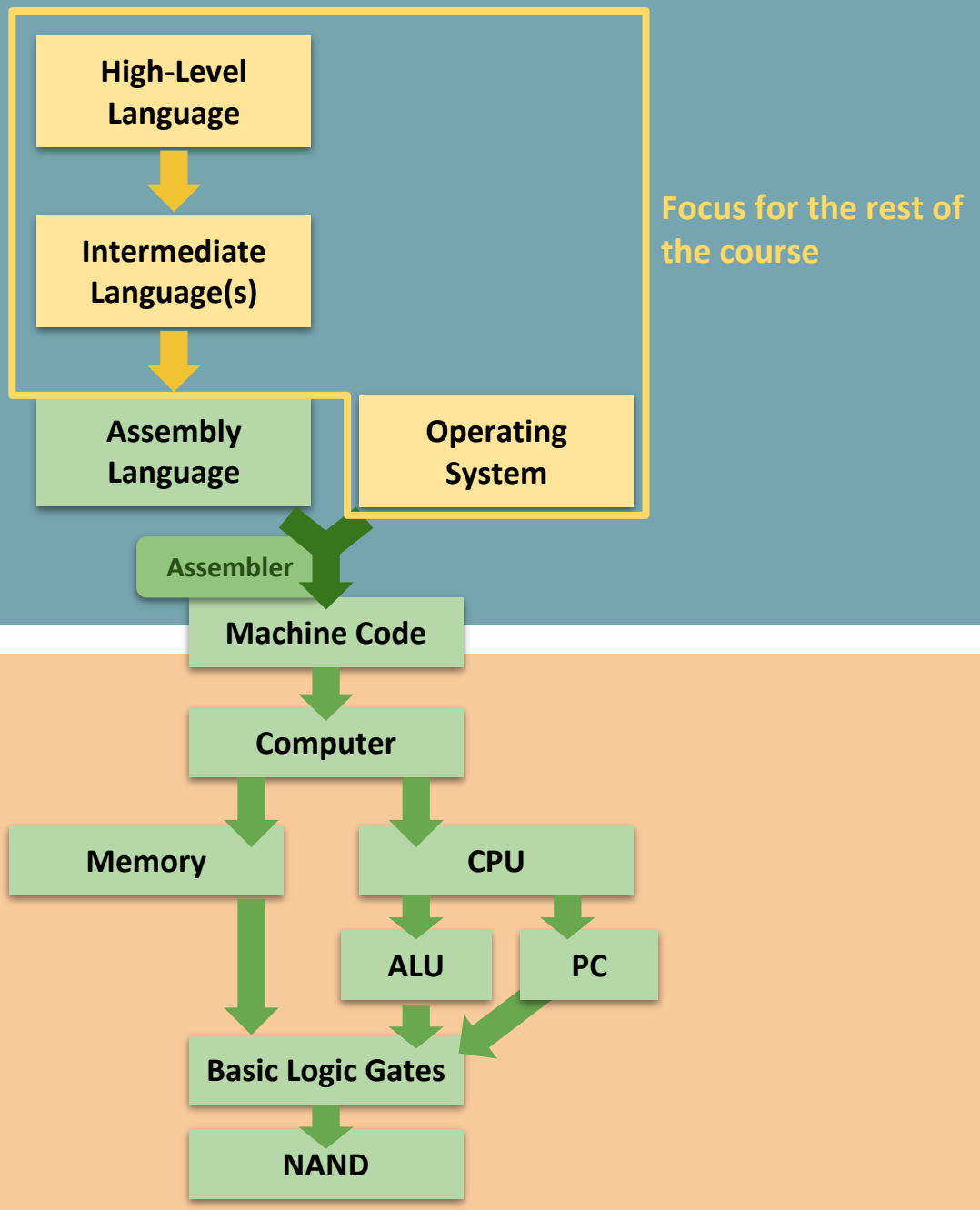
Roadmap



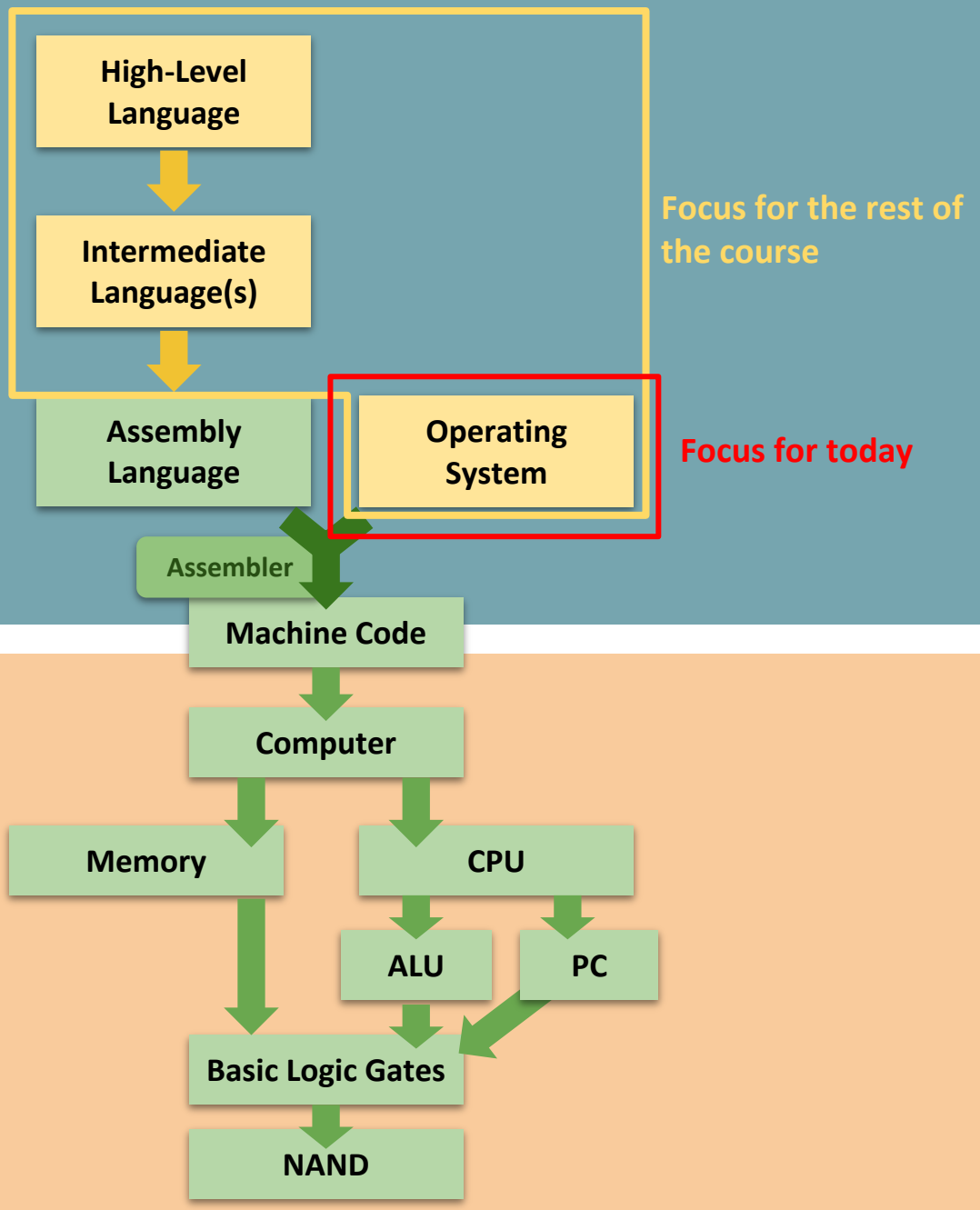
SOFTWARE

HARDWARE

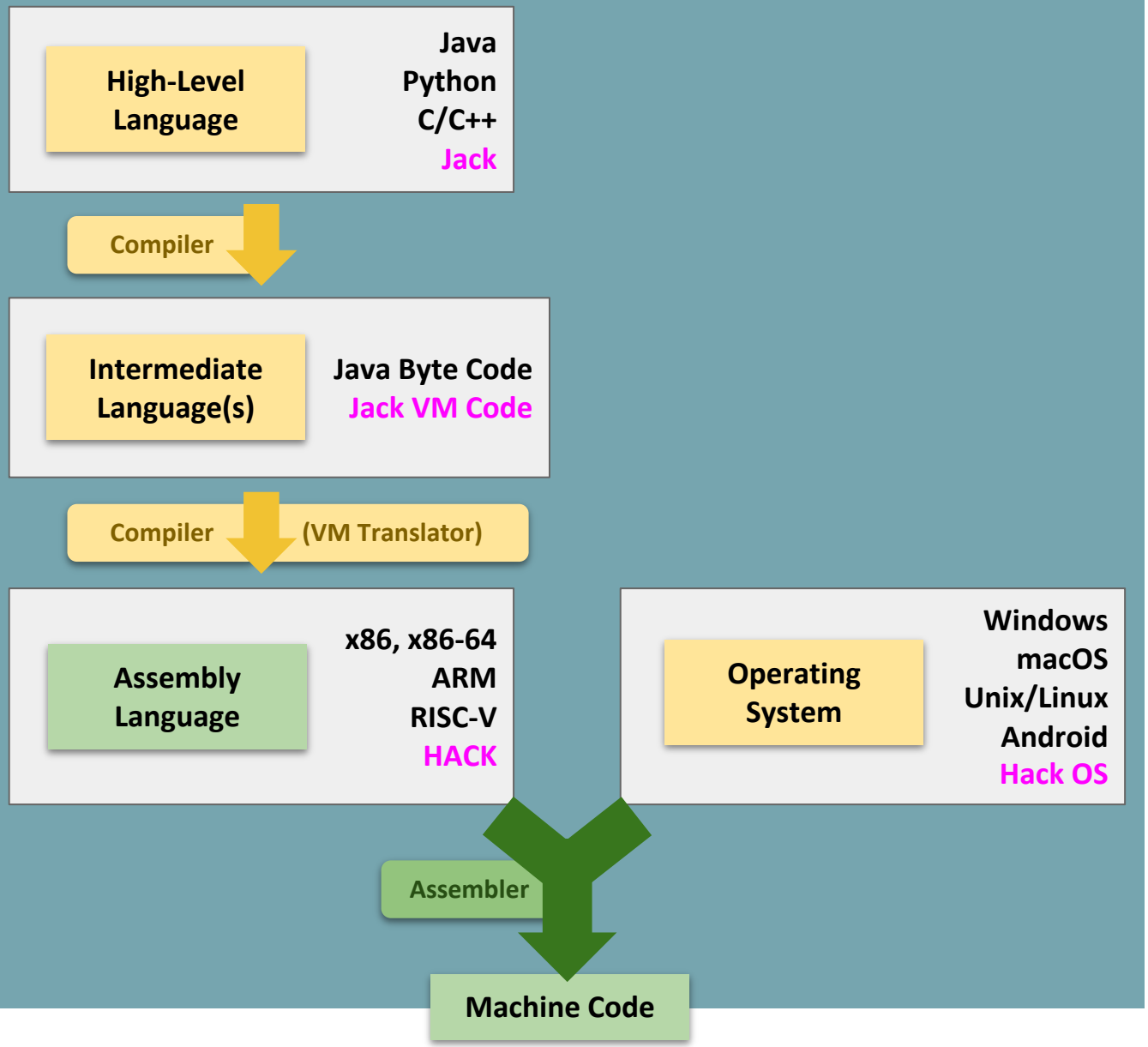
Roadmap


High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
Focus for the rest of the course
SOFTWARE
Assembler
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

Roadmap


High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
Focus for the rest of the course
Focus for today
SOFTWARE
Assembler
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

Software Overview



SOFTWARE

Lecture Outline

- Final Project Overview
  - E-Portfolio Details and Topics Brainstorming
- The Software Stack
  - Roadmap of Hardware and Software Components
- **Overview of Operating Systems**
  - **Abstraction, Protection, Processes, Virtual Memory**
- Project 8 Overview
  - Micro Jack Details, Tips for Getting Started

The Operating System

- The operating system (OS) is just another piece of software
  - A massive, complex piece of software
  - In the end, uses the same machine language your code does
- OS is more trusted than the rest of the software that runs on your computer
- User programs and applications invoke (ask) the OS to perform operations they are not trusted or allowed to
  - Means the OS needs to be secure

Why an Operating System?

- Directly interacts with the hardware

- Benefit: **Abstraction**
  - Provides high-level functionality for messy hardware devices
  - OS must be ported to new hardware, but user-level programs can then be portable

- Benefit: **Protection**
  - OS is trusted to touch hardware; user-level programs are not
  - Prevents user-level programs from causing errors in the hardware
  - Maintains security between programs and user accounts

Operating Systems: Abstraction

- Many abstractions provided by real-world operating systems

- File System
  - File contents = just bits in the "giant array" that is the hard drive ("permanent" storage, as opposed to temporary storage in RAM that disappears when computer is turned off)
  - OS keeps a record of which ones fall into which "files"

Operating Systems: Abstraction

- Many abstractions provided by real-world operating systems

- Network Stack
  - Communicating with network devices ≈ communicating with screen/keyboard memory map
  - OS handles messy, time-sensitive protocols

- Processes
  - Only one process can run at once on a CPU
  - Operating systems can manage resource sharing
  - OS switches very quickly, illusion of running both “at once”

Operating Systems: Protection

- The CPU has different "privilege" levels when it is executing (controlled by a register on the CPU)

- OS code and memory can only be executed by an OS privilege level
  - Your applications run at a lower level and cannot access OS code and memory

- This prevents applications from crashing entire system
  - For example, if your web browser crashes, usually it doesn't crash your entire computer
  - Also helpful for security purposes

Operating Systems: Protection

- Example: Suppose we want only the OS to be allowed to run instruction **`SET_ON_FIRE`**
  - But if the OS is just a machine code program like any other… what's the security hole?

Operating Systems: Protection

- Example: Suppose we want only the OS to be allowed to run instruction **`SET_ON_FIRE`**
  - But if the OS is just a machine code program like any other… what's the security hole?

Your Code

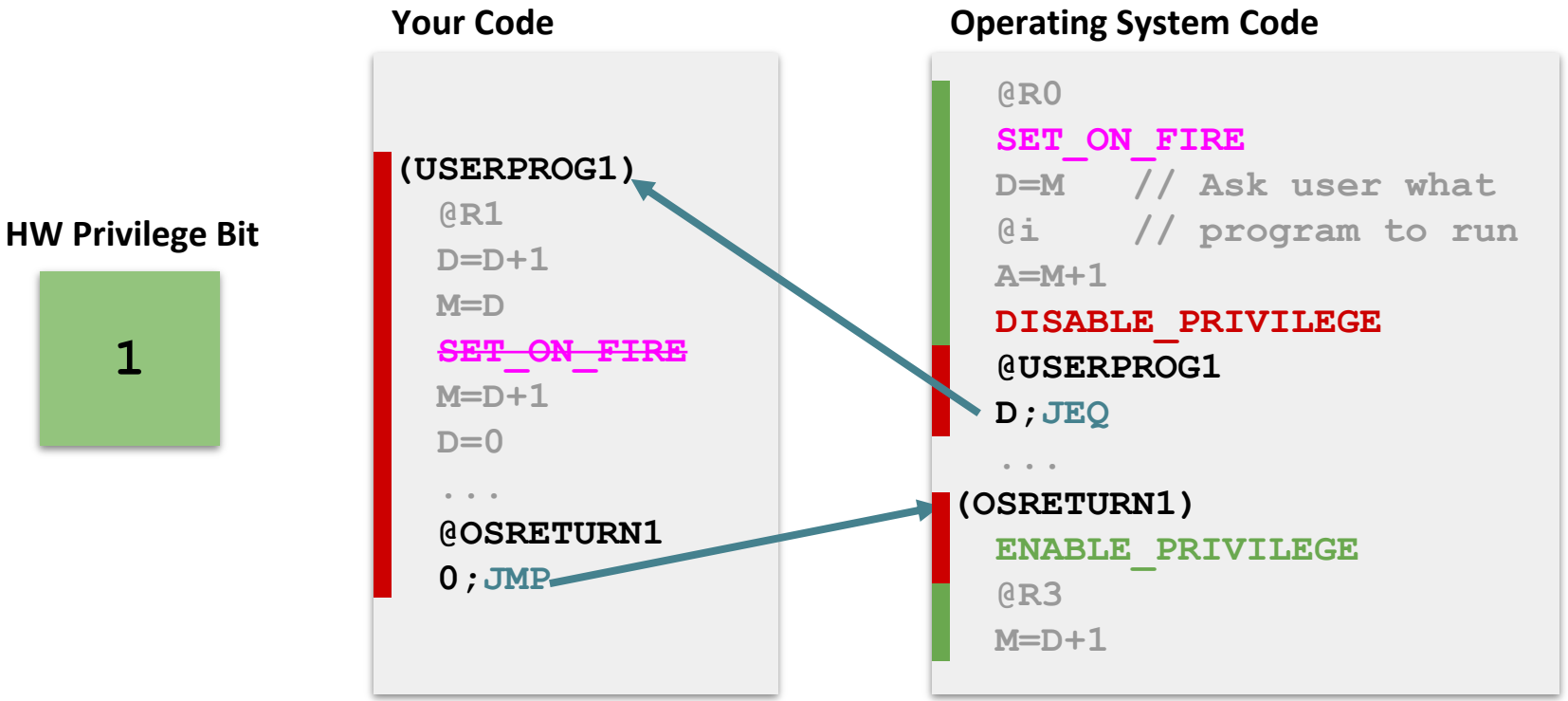
```
(USERPROG1)
  @R1
  D=D+1
  M=D
  SET_ON_FIRE
  M=D+1
  D=0
  ...
  @OSRETURN1
  0;JMP
```

Operating System Code

```
  @R0
  D=M    // Ask user what
  @i     // program to
run
  A=M+1
  M=-D
  @USERPROG1
  D;JEQ
  ...
(OSRETURN1)
  @R3
  M=D+1
```

Operating Systems: Protection

- The fix: hardware bit for "privileged mode"
  - Processor checks before running SET_ON_FIRE
  - OS disables before jumping to user code, re-enables on return
    - (Processor also must check that user code can't enable privilege)



Operating System: Processes

- A "process" is an application running on your computer
  - E.g., your web browser, terminal, Microsoft Word, etc.
- Each app instance contained in one or more processes
  - The OS manages these processes
- Multiple processes are "running" at the same time, but it's just the OS quickly switching between them
- A process only has access to its memory, and cannot access the memory of other processes
  - This is helpful because if one process crashes or is malicious, it makes it more difficult to crash or corrupt other processes too

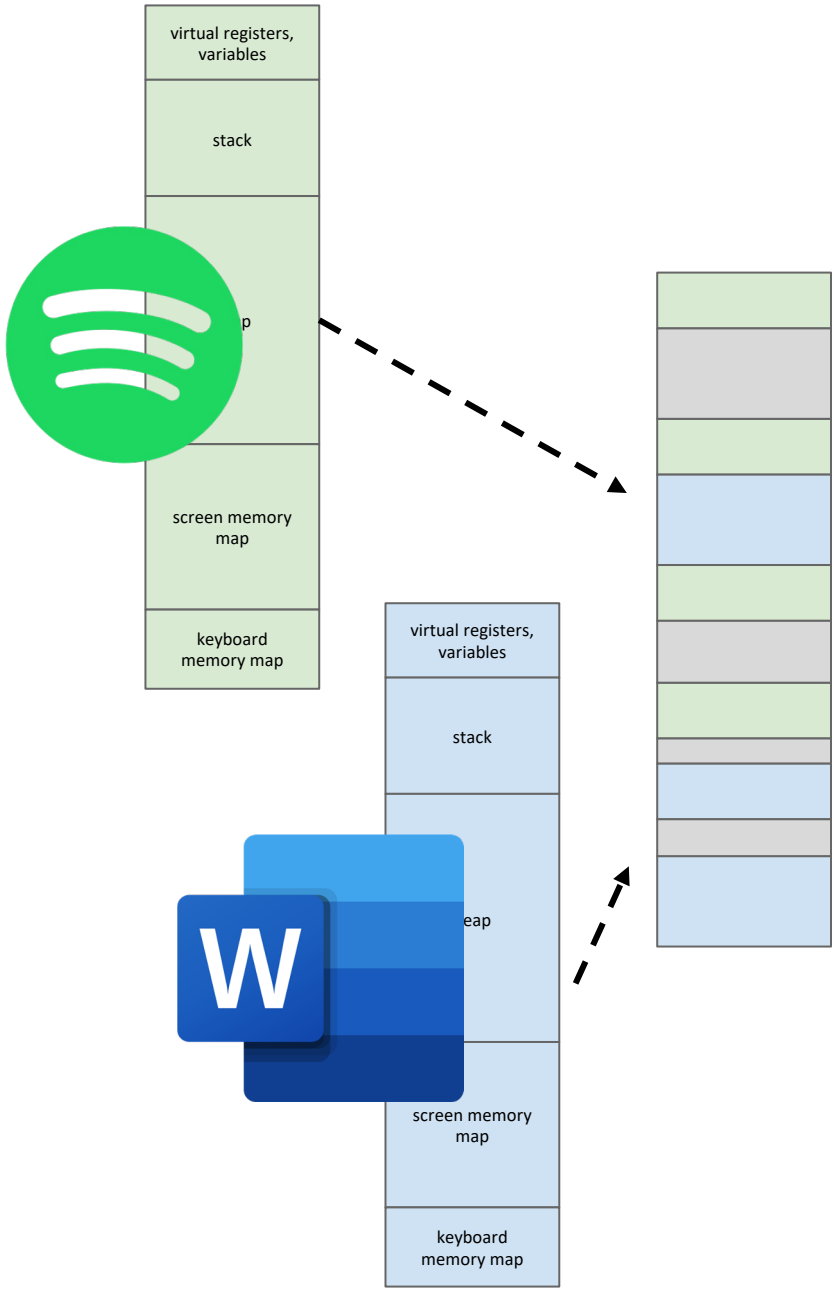
Why *Not* an Operating System?

- The Hack computer we've built is… small
  - Uses the same principles as your laptop CPU
  - But in terms of scale, closer to a microprocessor or small embedded chip

- For embedded systems, often an OS is overkill—instead, designed to be programmed with/run a single program at a time
  - Pro: developer gets complete control over the device
  - Con: re-implement OS features, no protection

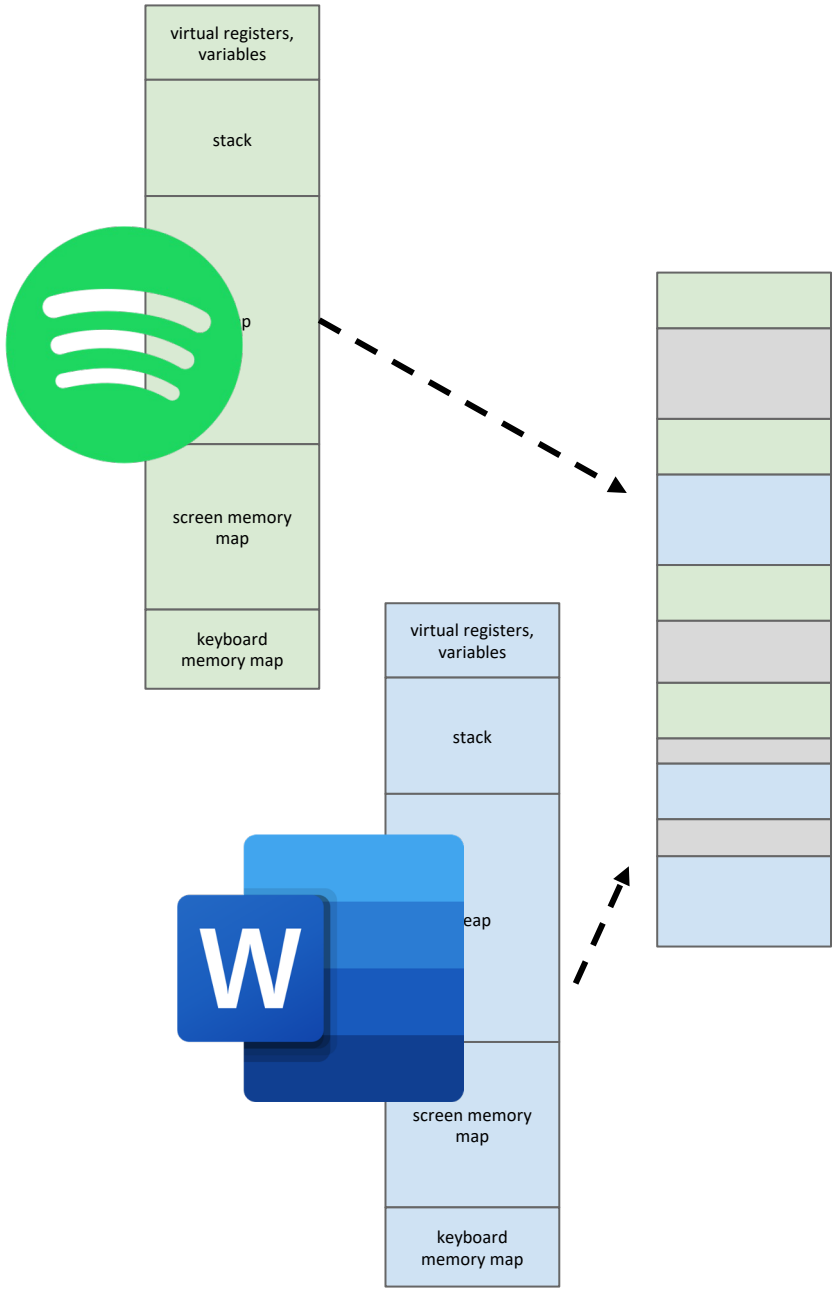
Virtual Memory

- Most OS's allow multiple processes, but shouldn't be able to modify values in another's address space
- OS provides illusion of separate address spaces via virtual memory
  - Really all one physical memory
  - OS & hardware map pieces of virtual memory to pieces of physical memory



Virtual Memory

- **Benefit of security**: Programs only know about their own address space
  - Don't even have a way to describe address of other application's data
- **Drawback is efficiency**
  - Virtual address translation is fast nowadays, but still slower than directly accessing memory



Comparison of Operating Systems

- Three different ways to do essentially the same thing
  - Everyone has their own preference
- Each has their own benefits and tradeoffs
  - Work on varying types of hardware, provide different levels of customization, different features, work better with different software, open source vs. proprietary, etc.
- You could choose to do some research next time you are deciding on a laptop, computer, or OS

Lecture Outline

- Final Project Overview
  - E-Portfolio Details and Topics Brainstorming
- The Software Stack
  - Roadmap of Hardware and Software Components
- Overview of Operating Systems
  - Abstraction, Protection, Processes, Virtual Memory
- **Project 8 Overview**
  - **Micro Jack Details, Tips for Getting Started**

Project 8 Overview

- You will be given starter code for a compiler that reads a micro version of Jack and spits out Hack

- The Scanner & Parser are working
  - Task A: read through comments to understand what's going on

- The Code Generation is buggy and half-finished
  - Task B: find the bugs by practicing deliberate debugging strategies (e.g., step through generated Hack code using CPUEmulator)
  - Task C: Complete the implementation of the compiler

Project 8: Micro Jack

- Stripped-down version of Jack language
  - More manageable but enough features to be interesting
- Available features:
  - Types: **`int`** and **`int[]`**, **`var`**
  - Structures: **`if`**, **`while`**, **`+`**, **`-`**, **`==`**, **`!=`**
- Missing features:
  - Functions, function calls, classes, objects, strings, for loops, array bounds checking, etc.

Any number of variable declarations

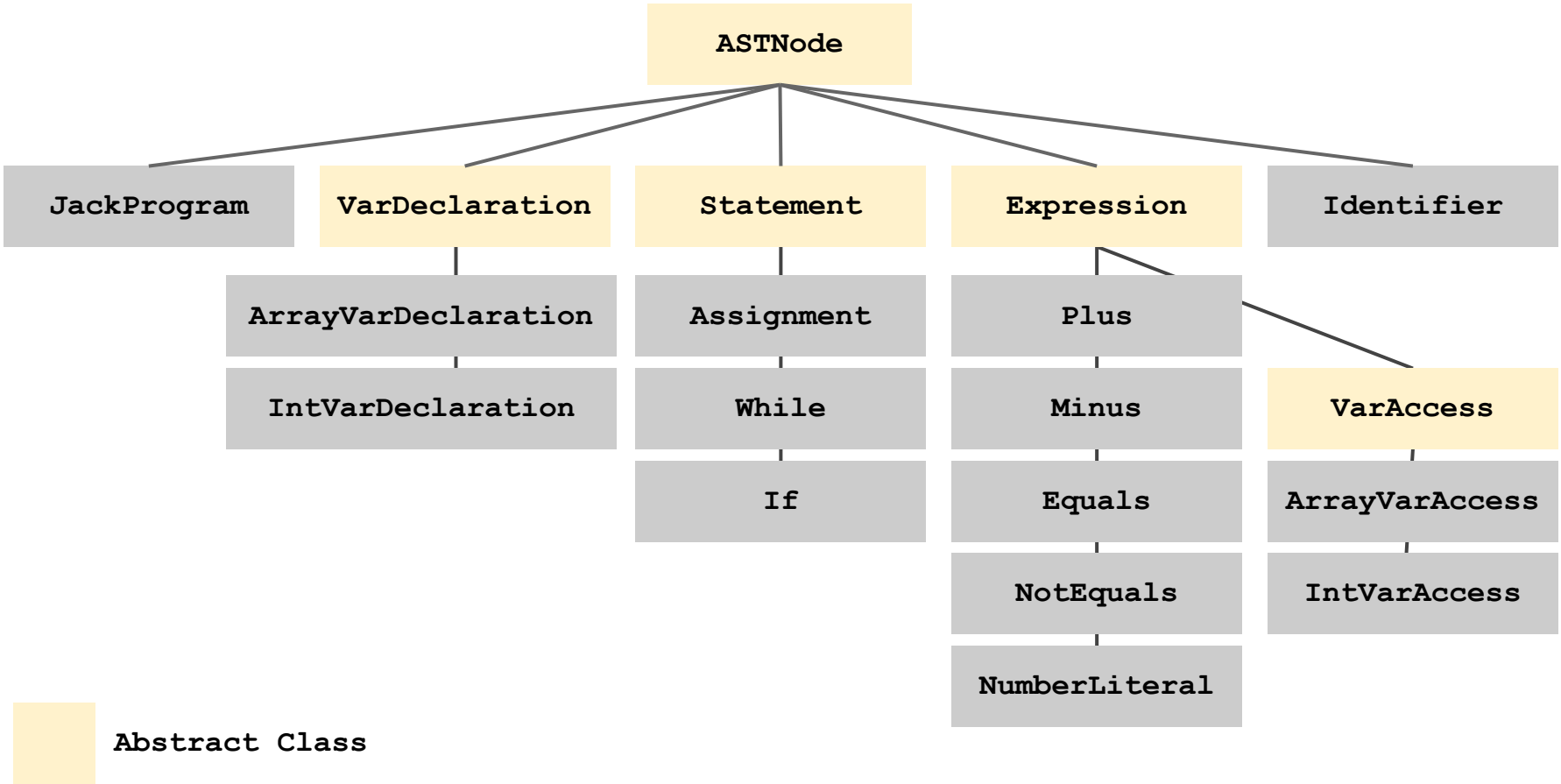
`Basic.jack`

```
var int a, b[1], c;
var int d[10], e;

let a = 1;
let b[0] = 1;
let n = 9;
while (n != 0) {
  let d[n] = a;
  let n = n - 1;
}
let screen[100] = d[0];
```

Then any number of statements

Project 8: The AST Nodes

- You are provided with all AST Node classes needed
  - All your code will be implemented within these classes



Project 8: Generating Code

- Each AST node has a printASM method that should print out Hack instructions to System.out (and recursively call printASM on children)
  - You're provided with **`instr("@R0")`** and **`label("LOOP")`** convenience functions
  - Each can take a comment as a second argument — highly recommended!

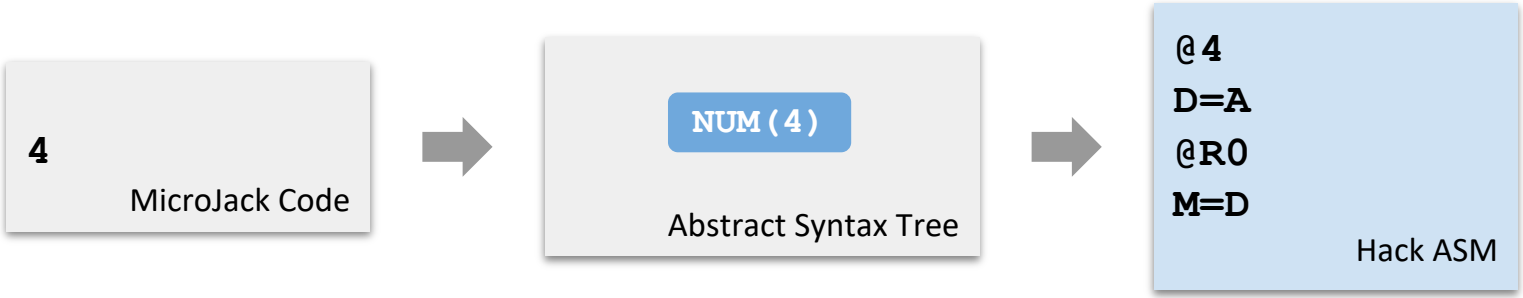
```
public class If extends Statement {
    public Expression condition;
    public List<Statement> statements;

    @Override
    public void printASM(symbolTable) {
        condition.printASM(symbolTable);
        instr("@R0", "Get cond result");
        instr("D=M");

        ...
    }
}
```

Project 8 Overview

- Step 1: Read comments provided in the starter code
- Step 2: Implement **`NumberLiteral.java`** (~4 lines)
- Step 3: Debug **`Plus.java`** (2 bugs)
- Step 4: Implement **`Minus.java`** (~13 lines, similar to **`Plus.java`**)
- Step 5: Implement **`NotEquals.java`** (~21 lines, similar to **`Equals.java`**)
- Step 6: Implement **`ArrayVarAccess.java`** (~3 lines)
- Step 7: Debug **`If.java`** (2 bugs)
- Step 8: Implement **`While.java`** (~14 lines)

Example: Number Literal (Step 1)

- Called a "literal" because it's a literal value embedded in the Micro Jack code
  - Generated Hack Assembly should simply put that value in R0



Example: Number Literal (Step 1)

```
public class NumberLiteral extends Expression {
    public int value;

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr( ? );
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

4

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

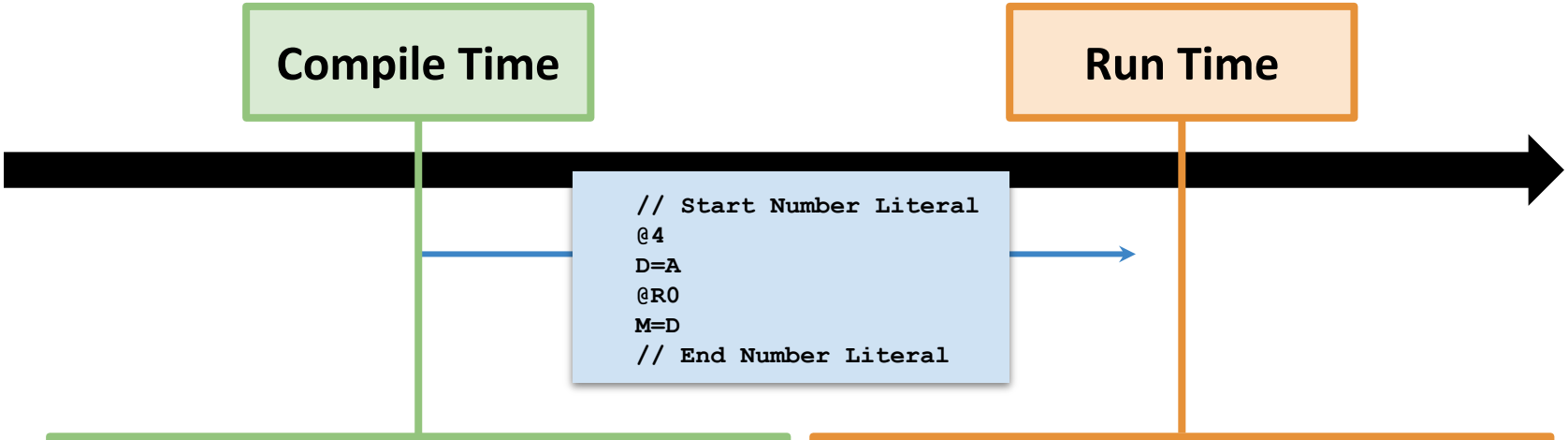
Example: Number Literal (Step 1)

```
public class NumberLiteral extends Expression {
    public int value;

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr("@" + toString());
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

4

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

Example: Number Literal (Step 1)



- Java (the compiler) is running
- 4 is stored as value field inside an NumberLiteral ASTNode
- When executed, **prints code that stores it another way**!

```
public class NumberLiteral extends Expression {
    public int value;

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr("@" + toString());
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

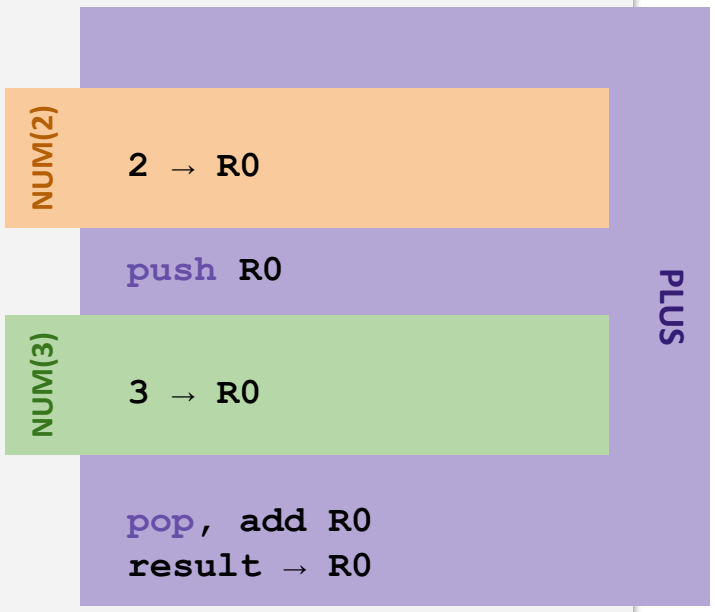
- Hack ASM (the output) is running
- 4 is stored as constant inside an assembly instruction
- When executed, loads 4 from instruction into A register

Example: Plus (Step 2)

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```

Example: Plus (Step 2)

1 Structural Bug: Map to abstract diagram for Plus:




```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");

        left.printASM();

        instr("@R0");
        instr("D=M");

        right.printASM();

        push();

        instr("@R1");
        instr("A=M");

        instr("D=D+A", "perform the addition");

        ...
```

1 Detail Bug: Step through generated code, Check state at each step

Micro Jack Technical Details

- Can't write a negative integer literal
  - Instead, use subtraction from zero: `0 - 1`
- All variable declarations must come before all regular statements
  - Why? Simplifies concept of a "defined" variable
- No defined operator precedence
  - If order matters for an operation, use parentheses

Micro Jack Technical Details

- Arrays are just as you would expect
  - **`arr[index]`** just calculating an address: take address of **`arr`** variable and add index to it as an offset
  - No array bounds checking — you can run off the end of an array

- Booleans are just 0 (false) and non-zero (true)

Project 8: Debugging Tips

- Try walking through the general **`printASM`** code to understand why each line is there
  - Add comments to the assembly as you go! Much easier to understand resulting file
- Find the smallest example you can
  - Provided tests get progressively more complex, but you may want to write your own tiny test case to isolate
  - **`printASM`** methods can get long fast—we've added comments so you can isolate to the section you're working on
- "Play Computer": as you step through the code, write down the state you expect after each instruction, then advance and see if the CPUEmulator agrees

Additional Project 8 Tips

- When debugging assembly, a good first step is to try understanding the code and adding comments to the assembly as you go
  - Much easier to understand resulting file

- A **`printDebug`** method has been implemented for you on all AST nodes
  - Use it to visualize exactly what the parser is giving you, but also as a basis for **`printASM`**
  - Both need to do processing on the current node and strategically recurse on its children

Additional Project 8 Tips

- Pushing and popping from the stack can be intimidating, but formulaic
  - Understand it once, copy and paste afterward
  - **`push()`** and **`pop()`** are already implemented for you

- We provide only a few Micro Jack test files
  - We encourage you to write more of your own (think back to the debugging lecture)
  - Can use **`Sandbox.*`** to write more tests or create your own files

Project 8 Tools Practice

- Practice using the Project 8 tools — try the following:
  - Run **`git pull`** to pull the Project 8 starter code
  - Navigate to the **`src`** directory: **`cd src`**
  - Compile the Java source code of the compiler by running: **`javac $(find . -name "*.java")`**
  - Use your compiler to compile the Jack file for the **`OnlyVars.jack`** program: **`java compiler/Compiler compile ../test/OnlyVars.jack`**
  - Load and run **`OnlyVars.tst`** in the CPUEmulator

- The above steps were taken from the "How to Run Tests" portion of the specification
  - Can refer to this when needed as you work through the project

Post-Lecture 17 Reminders

- This Thursday's Lecture: Final Project E-Portfolio Workshop & Computer Networks

- Project Reminders
  - **Project 7, Part II (Professor Meeting Report) due this Thursday (12/1) at 11:59pm**
  - Project 8 (Debugging & Implementing a Compiler) due next Tuesday (12/6) at 11:59pm
  - Final Project, Part I (E-Portfolio Outline) due next Thursday (12/8) at 11:59pm

- Check Canvas for late updated late days through Project 6