CSE 390B, Winter 2022

Building Academic Success Through Bottom-Up Computing

Assembler, Operating System

Inside the Assembler, The Software Stack, Fundamentals of the Operating System (OS)

W UNIVERSITY of WASHINGTON

Lecture Outline

- **Inside the Assembler**
  - **Producing machine code, parsing, symbols, encoding**
- The Software Stack
  - Roadmap of hardware and software components
- Fundamentals of the Operating System (OS)
  - OS abstraction, protection, and memory

Producing Machine Code

```
while (i < 100)
{
  sum += arr[i];
  i++;
}
```

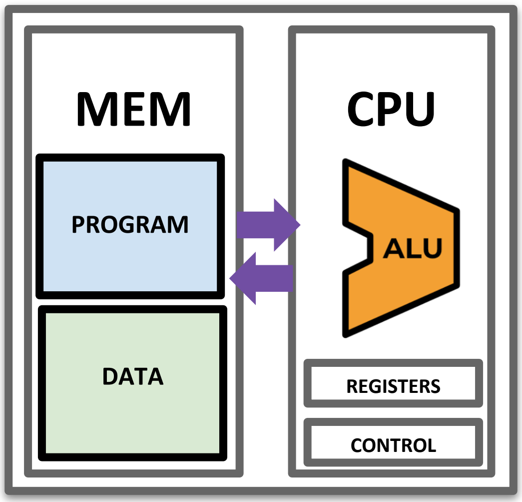
Java

Compile

```
0101110011100110
1011000101010100
1110001011111100
...
```

Machine Code Instructions

Load & Execute

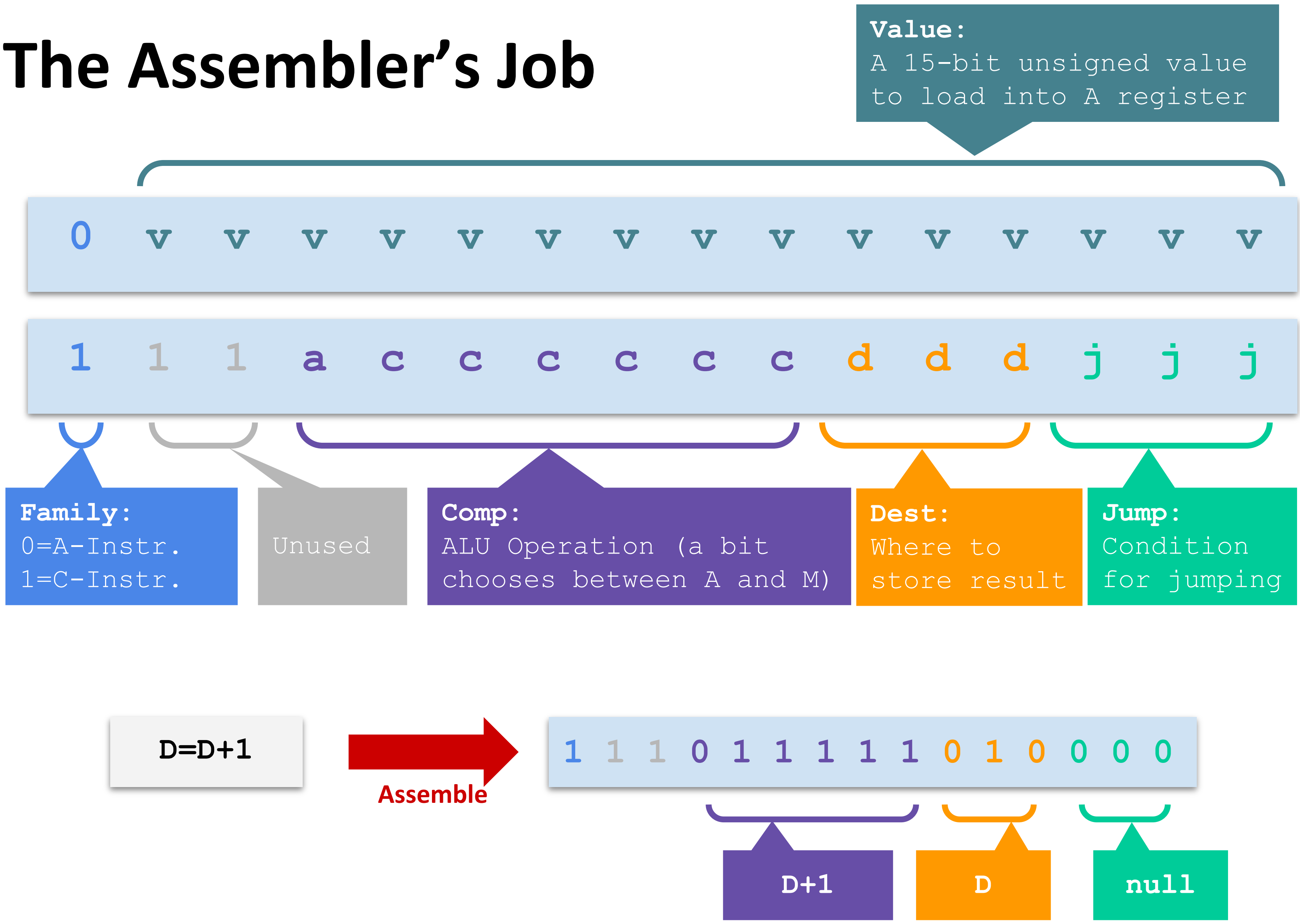



```
movq $5, %rdx
addq %rsx, %rdx
movq %rdx, %rax
ret
```

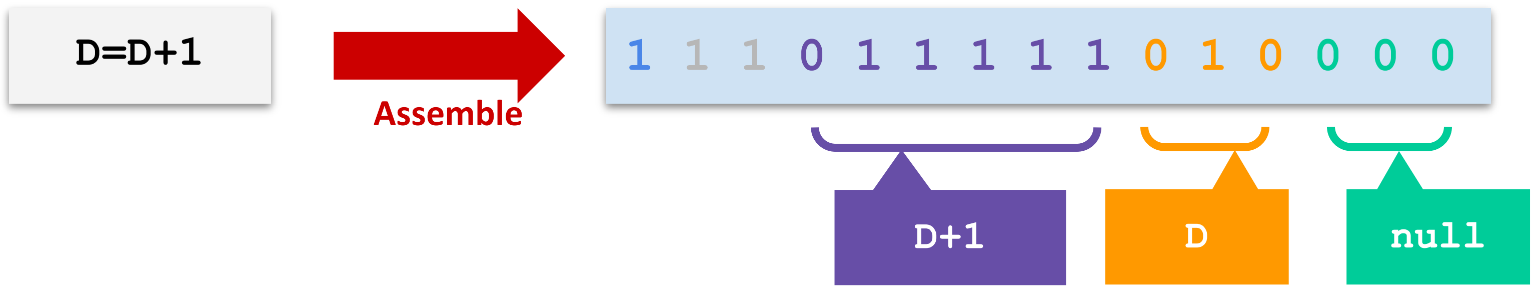
Assembly Language

Assemble

The Assembler's Job



The Assembler's Job



- Look up each value in the corresponding table

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	**Mnemonic**	**Effect**
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

d1	d2	d3	***Mnemonic***	***Destination (where to store the computed value)***
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

<table>
<tr><th>(when a=0)
comp mnemonic</th><th>c1</th><th>c2</th><th>c3</th><th>c4</th><th>c5</th><th>c6</th><th>(when a=1)
comp mnemonic</th></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td></td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td>-1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td></td></tr>
<tr><td>D</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td></td></tr>
<tr><td>A</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>M</td></tr>
<tr><td>!D</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td></td></tr>
<tr><td>!A</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>!M</td></tr>
<tr><td>-D</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td>-A</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>-M</td></tr>
<tr><td>D+1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td>A+1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>M+1</td></tr>
<tr><td>D-1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td></td></tr>
<tr><td>A-1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>M-1</td></tr>
<tr><td>D+A</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>D+M</td></tr>
<tr><td>D-A</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>D-M</td></tr>
<tr><td>A-D</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>M-D</td></tr>
<tr><td>D&A</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>D&M</td></tr>
<tr><td>D|A</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>D|M</td></tr>
</table>

What Makes This Hard?

Line #	
1	`@12`
2	`D=A`
3	`@i`
4	`M=D   // init`
5	`(LOOP)`
6	`@R3`
7	`MD = M-1`
8	`@LOOP`
9	`D;JGT`

Assemble

Address	
0	`0000000000001100`
1	`1110110000010000`
2	`0000000000010000`
3	`1110001100001000`
4	`0000000000000011`
5	`1111110010011000`
6	`0000000000000100`
7	`1110001100000001`

What Makes This Hard?

- Three broad concerns:

PARSING	Recognizing type of each instruction/label, extracting relevant fields, skipping whitespace & comments
SYMBOLS	Mapping from labels to instruction addresses, mapping from code symbols to RAM addresses, creating new symbols, corresponding line numbers to instruction addresses
ENCODING	Converting relevant fields to binary values, converting symbol values to binary values

Bells and Whistles… Why Bother?

- Tradeoff: Adding convenience for programmer makes it harder to build the Assembler
  - E.g., removing symbols from Hack would make Assembler much simpler, still possible to write all the same programs!
  - But language would be far more annoying to use

Bells and Whistles… Why Bother?

- Tradeoff: Adding convenience for programmer makes it harder to build the Assembler
  - E.g., removing symbols from Hack would make Assembler much simpler, still possible to write all the same programs!
  - But language would be far more annoying to use

- **Don't underestimate the importance of convenience!**
  - Put another way: Adding these extra features makes programmers more **productive**

Parsing

- Source code is just a giant string: we need to go character-by-character to understand that string

- Parser presents iterator-like interface:
  - To "advance" one instruction:
    - Move cursor forward, skipping whitespace and comments, until next non-empty line (ending on a newline)
  - To "read" current instruction:
    - Throw away whitespace & comments
    - Determine what type of instruction
    - Pull relevant fields out

Symbols: Labels

- Keep symbol table, mapping symbols (strings) to their values (integers)
  - Initialize with built-in symbols

SYMBOL	VALUE
R0	0
R1	1
...	...
R15	15
SCREEN	16384
KBD	24576

Symbols: Labels

- Keep symbol table, mapping symbols (strings) to their values (integers)
  - Initialize with built-in symbols

SYMBOL	VALUE
R0	0
R1	1
...	...
R15	15
SCREEN	16384
KBD	24576

- Run through instructions, using this pseudocode:

```
If current line is (LABEL):
    Add LABEL → next line number to
    symbol table
If current line is @LABEL:
    Lookup LABEL in symbol table,
    insert value into A instruction
```

Symbols: Labels

- Problem: What if a label's use comes before its definition?

Line #

```
   @LOOP
   0;JMP
   D=M
(LOOP)
   @var
```

Symbols: Labels

- Problem: What if a label's use comes before its definition?

- Solution: Two passes!
  - Pass 1: Populate symbol table by moving through file and ignoring anything that isn't a (LABEL) line
  - Pass 2: Go through file again, ignoring (LABEL) lines, encoding C-instructions, and encoding A-instructions according to symbol table lookup

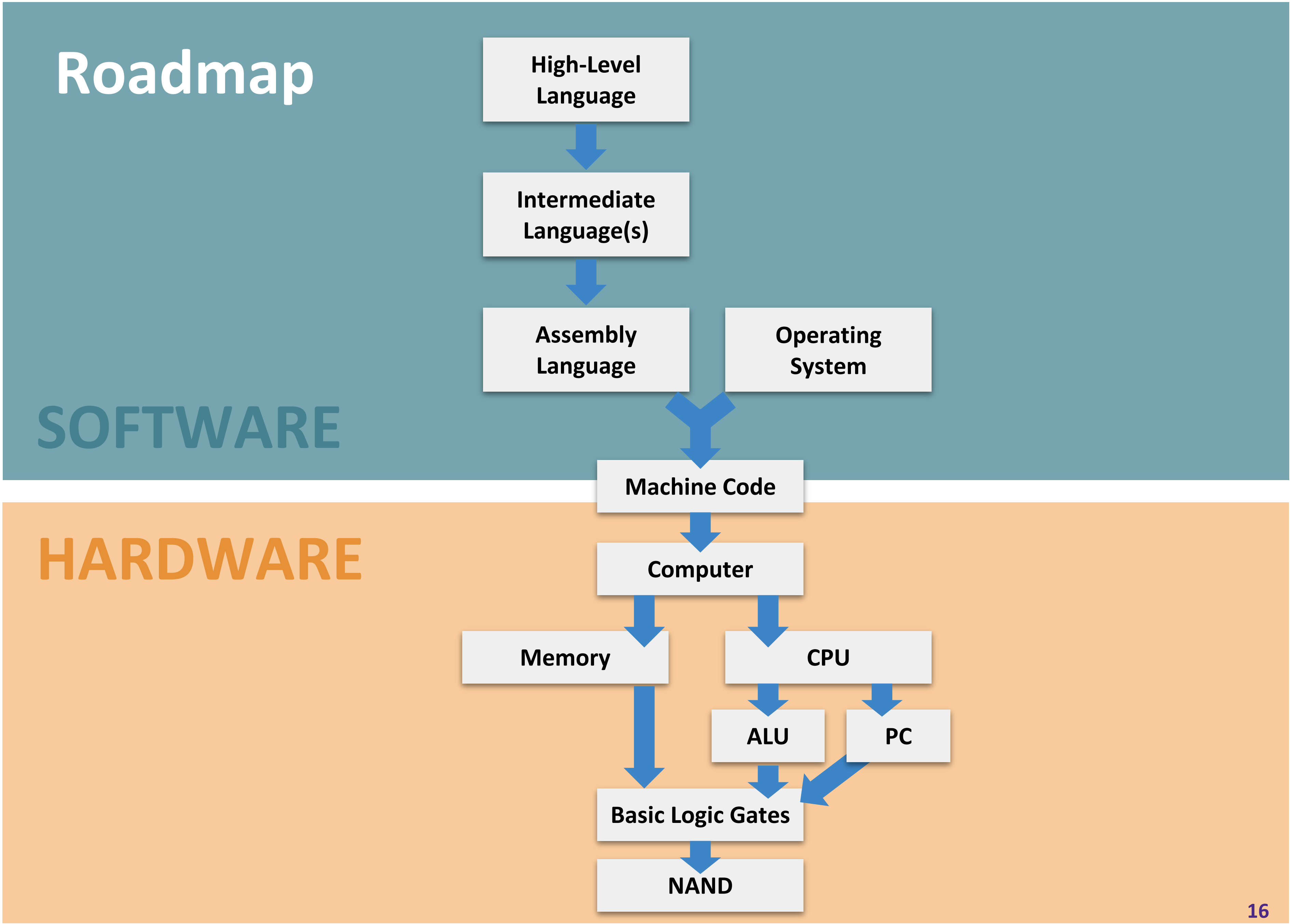
Line #

```
   @LOOP
   0;JMP
   D=M
(LOOP)
   @var
```

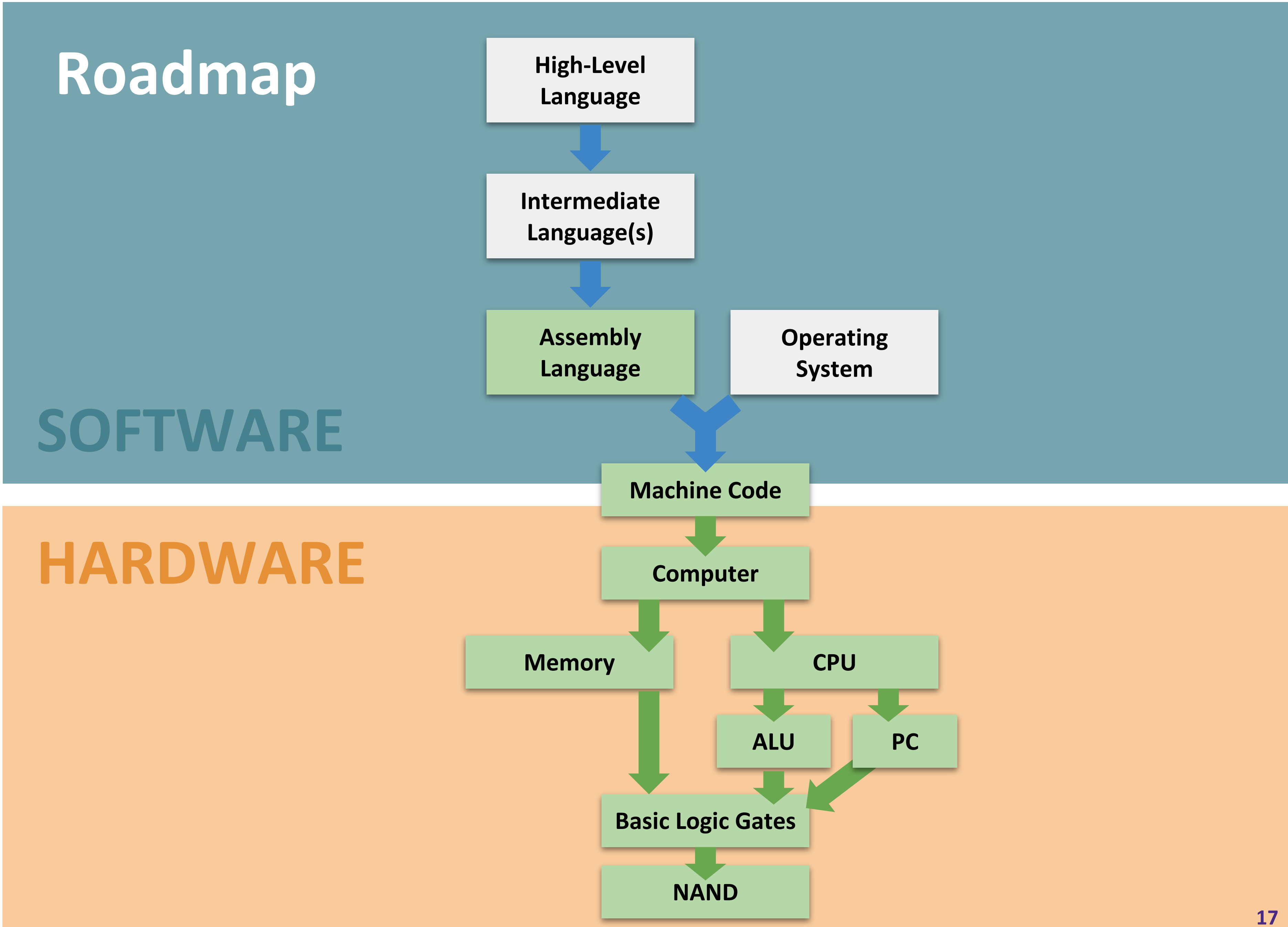
Lecture Outline

- Inside the Assembler
  - Producing machine code, parsing, symbols, encoding
- **The Software Stack**
  - **Roadmap of hardware and software components**
- Fundamentals of the Operating System (OS)
  - OS abstraction, protection, and memory

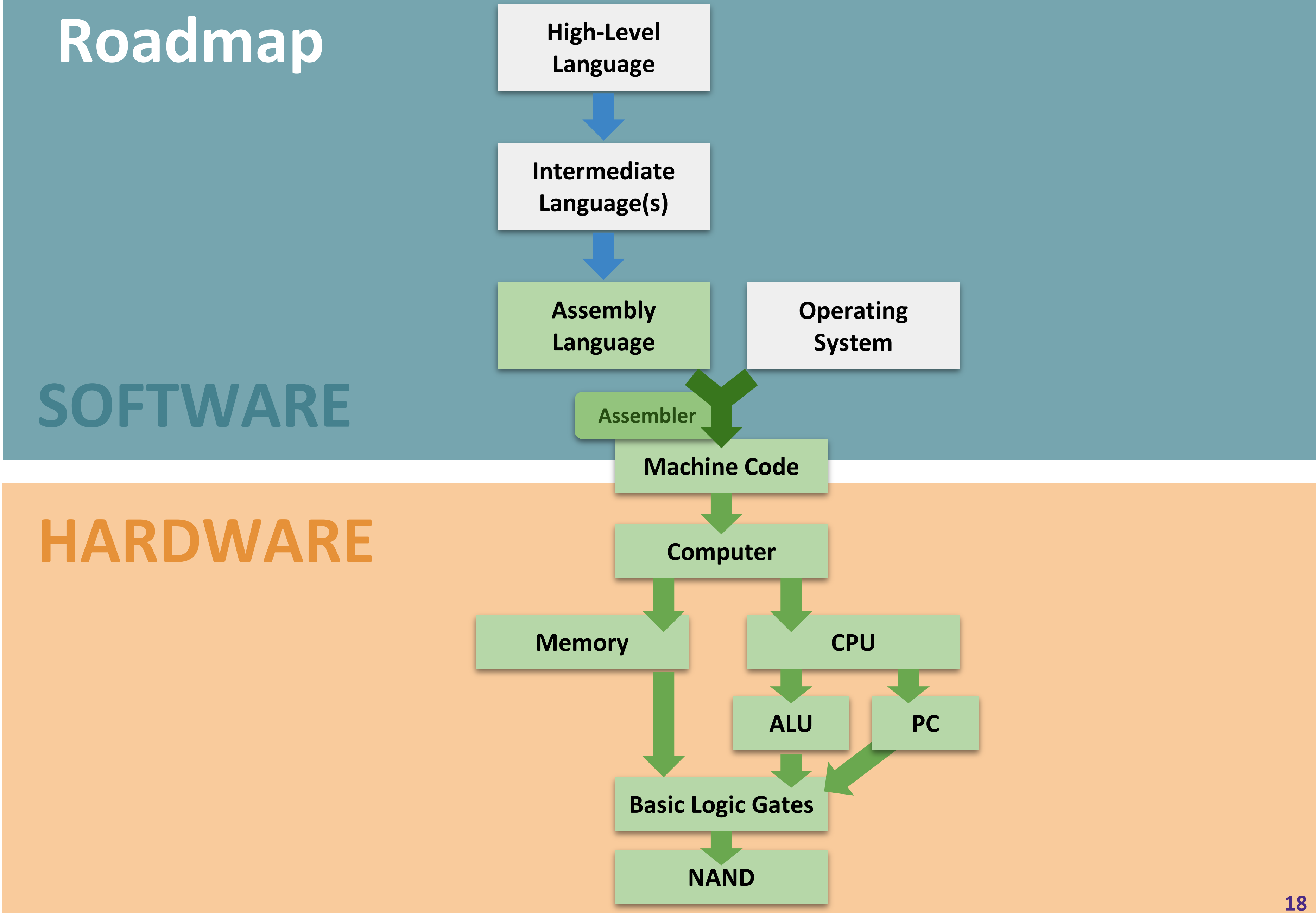
Roadmap


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

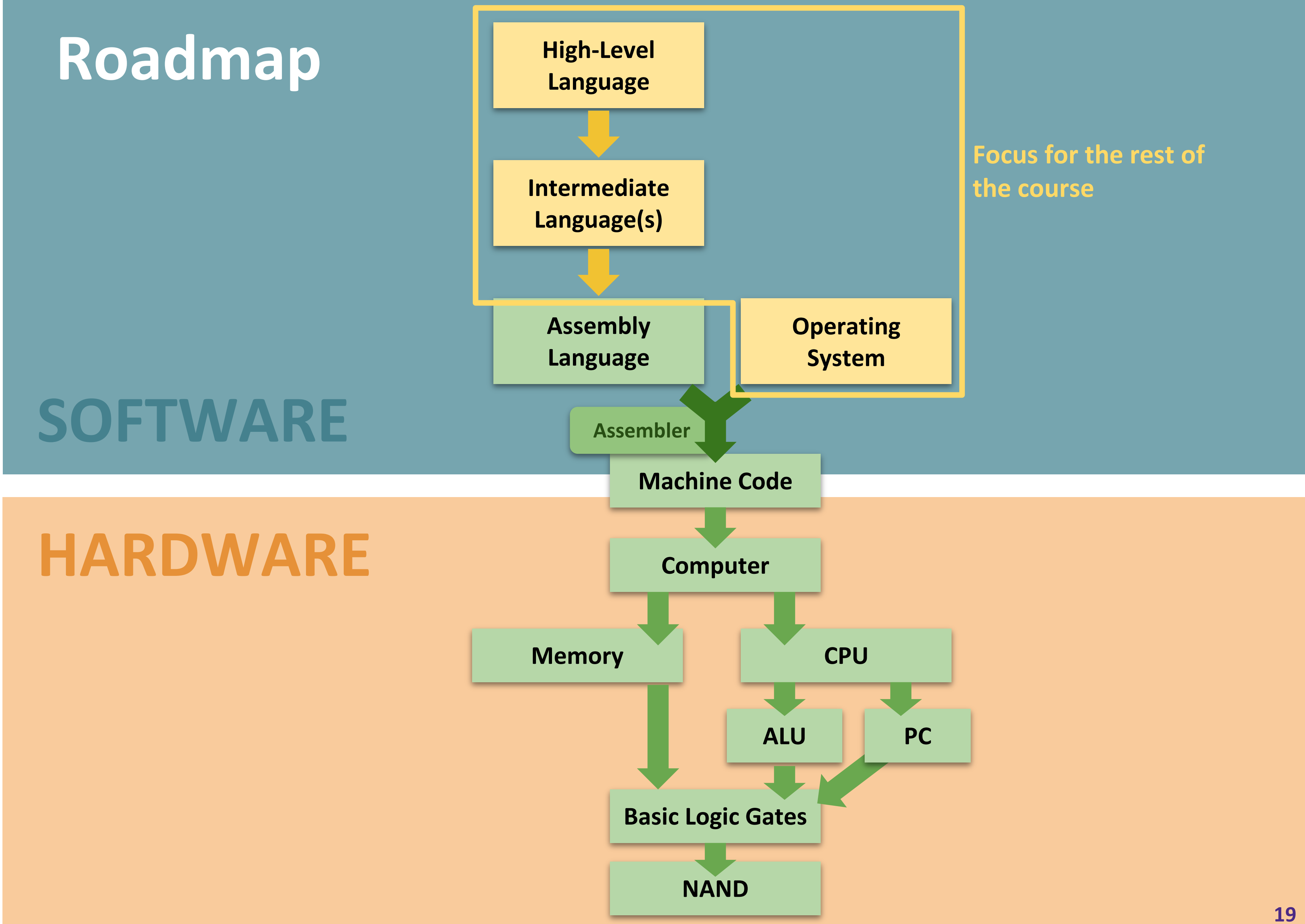
Roadmap


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

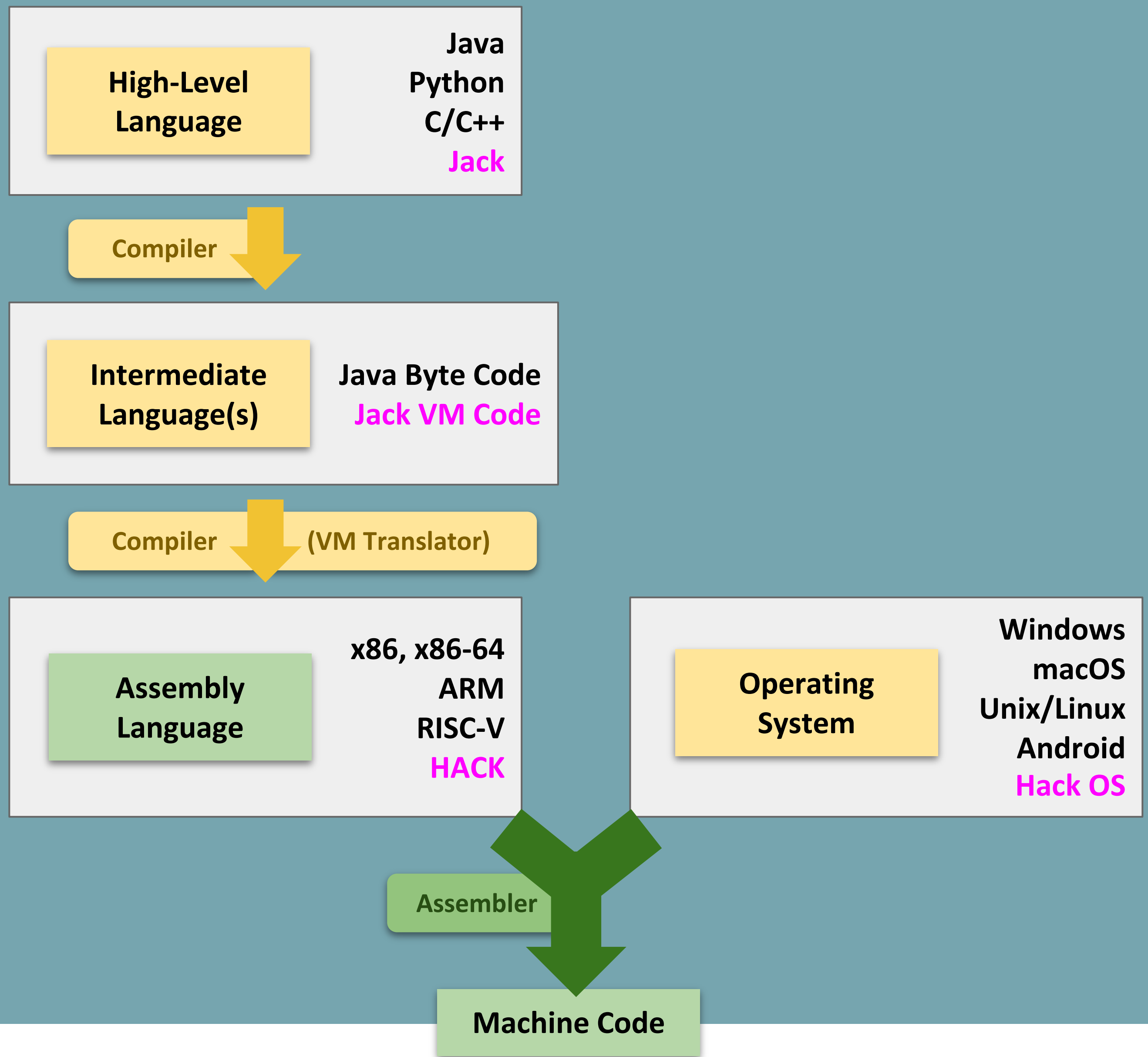
Roadmap


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

Software Overview



SOFTWARE

Lecture Outline

- Inside the Assembler
  - Producing machine code, parsing, symbols, encoding
- The Software Stack
  - Roadmap of hardware and software components
- **Fundamentals of the Operating System (OS)**
  - **OS abstraction, protection, and memory**

The Operating System

- 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/applications invoke (ask) the OS to perform operations they are not trusted or allowed to
  - Means the OS has 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
  - User-level programs cannot "break things"
  - Maintains security between programs and user accounts

Operating System: 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 System: 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
  - OS switches very quickly, illusion of running both “at once”

Operating System: 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 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

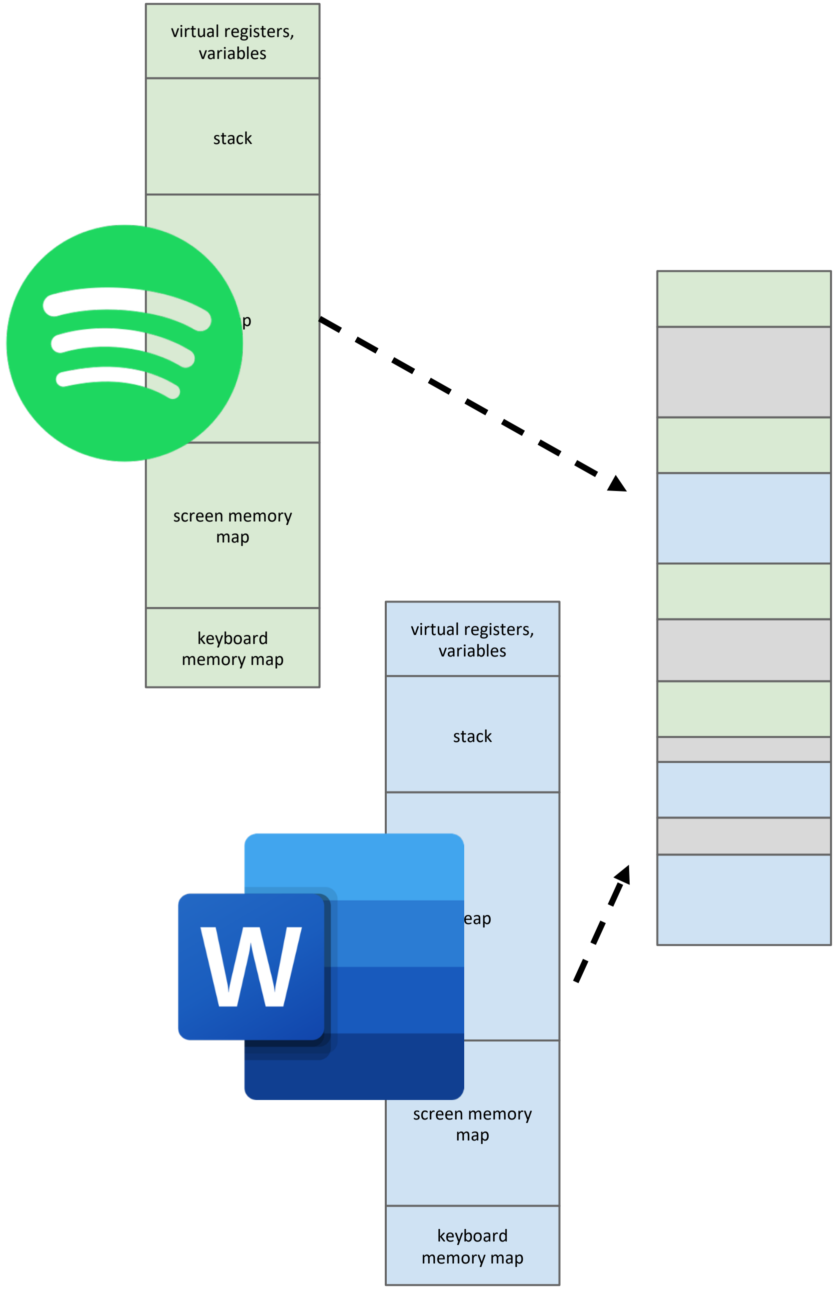
Why *Not* an Operating System?

- 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

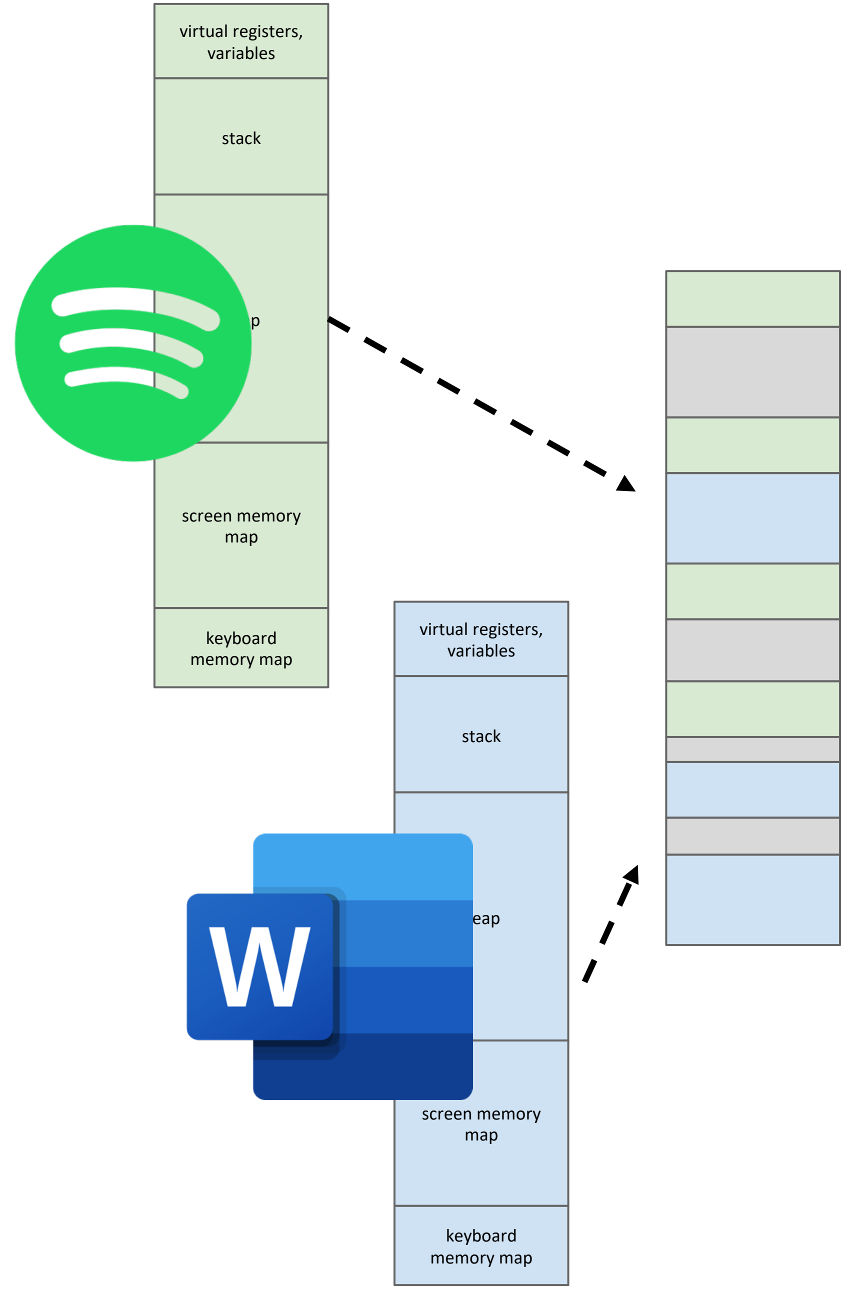
Virtual Memory

- 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

- Pro:
  - Security: programs only know about their own address space
    - Don’t even have a way to describe address of other application’s data
- Con:
  - Efficiency: virtual address translation is fast nowadays but still slower than directly accessing memory (what microprocessors do)



Comparison of Operating Systems

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

Post-Lecture 13 Reminders

- Project 5: Building a Computer Part II and Timed Mocked Exam due this **Thursday (2/17) at 11:59pm PST**
- Midterm will be graded with feedback by Wednesday (2/16) evening
- Thursday’s Lecture: Midterm Debrief and the Compiler
- Please submit the mid-quarter feedback form if you haven’t already!