CSE 390B, Spring 2022

Building Academic Success Through Bottom-Up Computing

Assembler & Compilers Overview

Inside the Assembler, The Software Stack, Compilers Overview, Project 6 Tips

Lecture Outline

- **Inside the Assembler**
  - **Producing Machine Code**
  - **Parsing, Symbols, Encoding**
- The Software Stack
  - Roadmap of Hardware and Software Components
- Compilers Overview
  - Roadmap of Hardware and Software Components
- Hack CPU Logic Example: writeM
  - Project 6 CPU Logic Exercise

Producing Machine Code

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

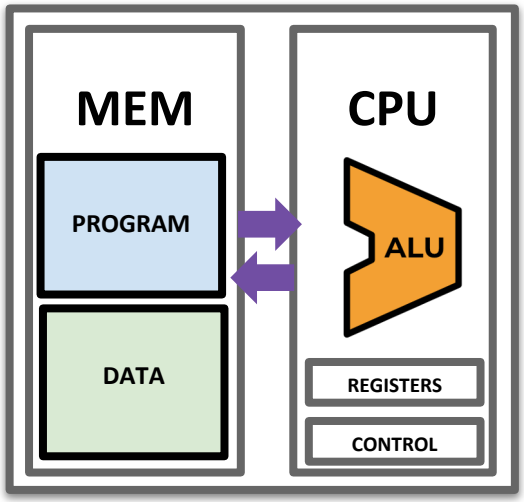
Java

Compile

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

Machine Code Instructions

Load & Execute

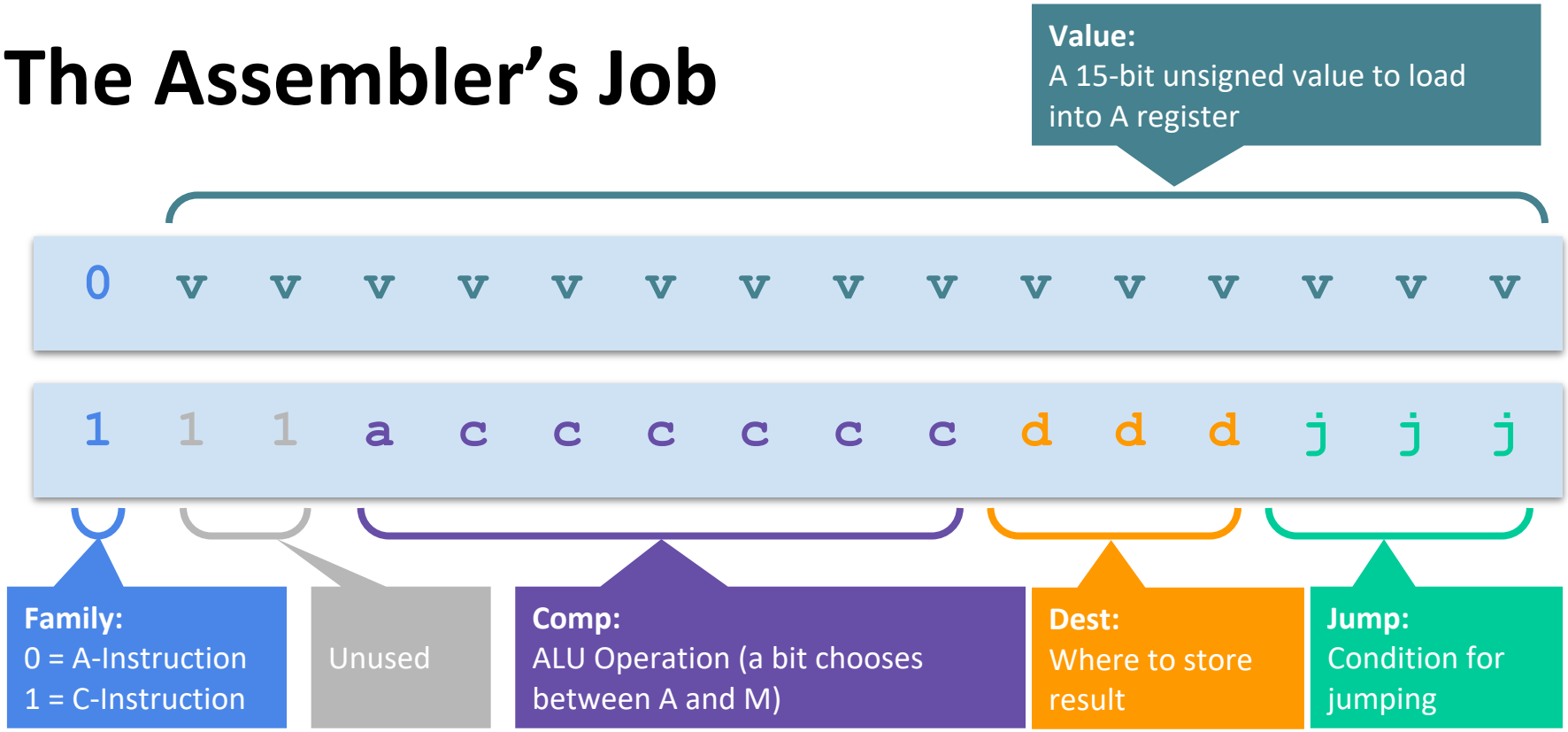


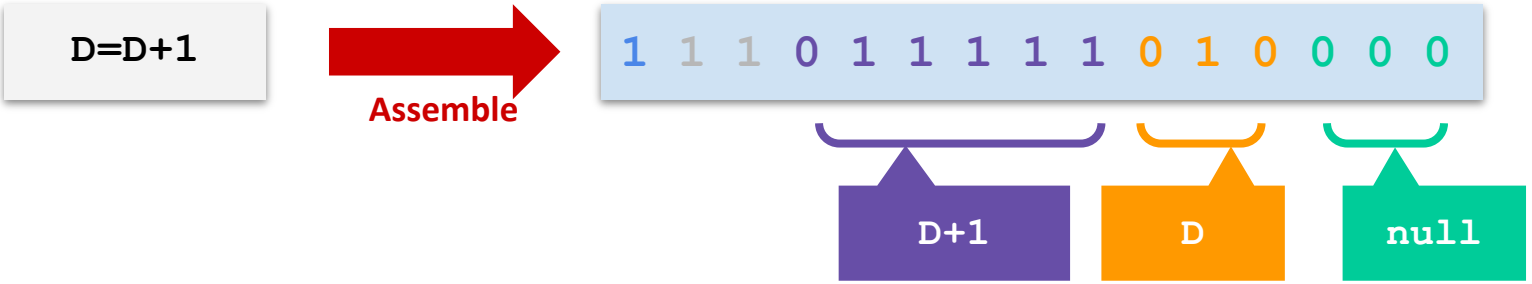

```
movq $5, %rdx
addq %rsx, %rdx
movq %rdx, %rax
ret
```

Assembly Language

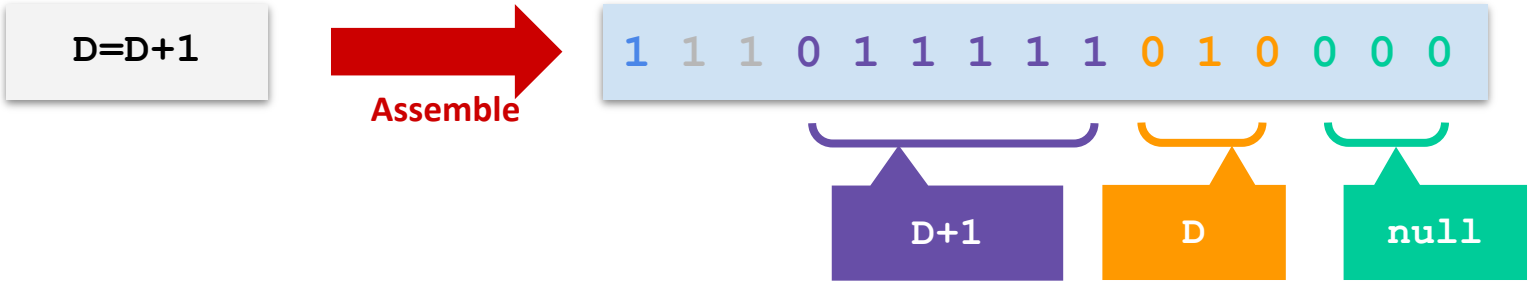
Assemble

The Assembler's Job





The Assembler's Job



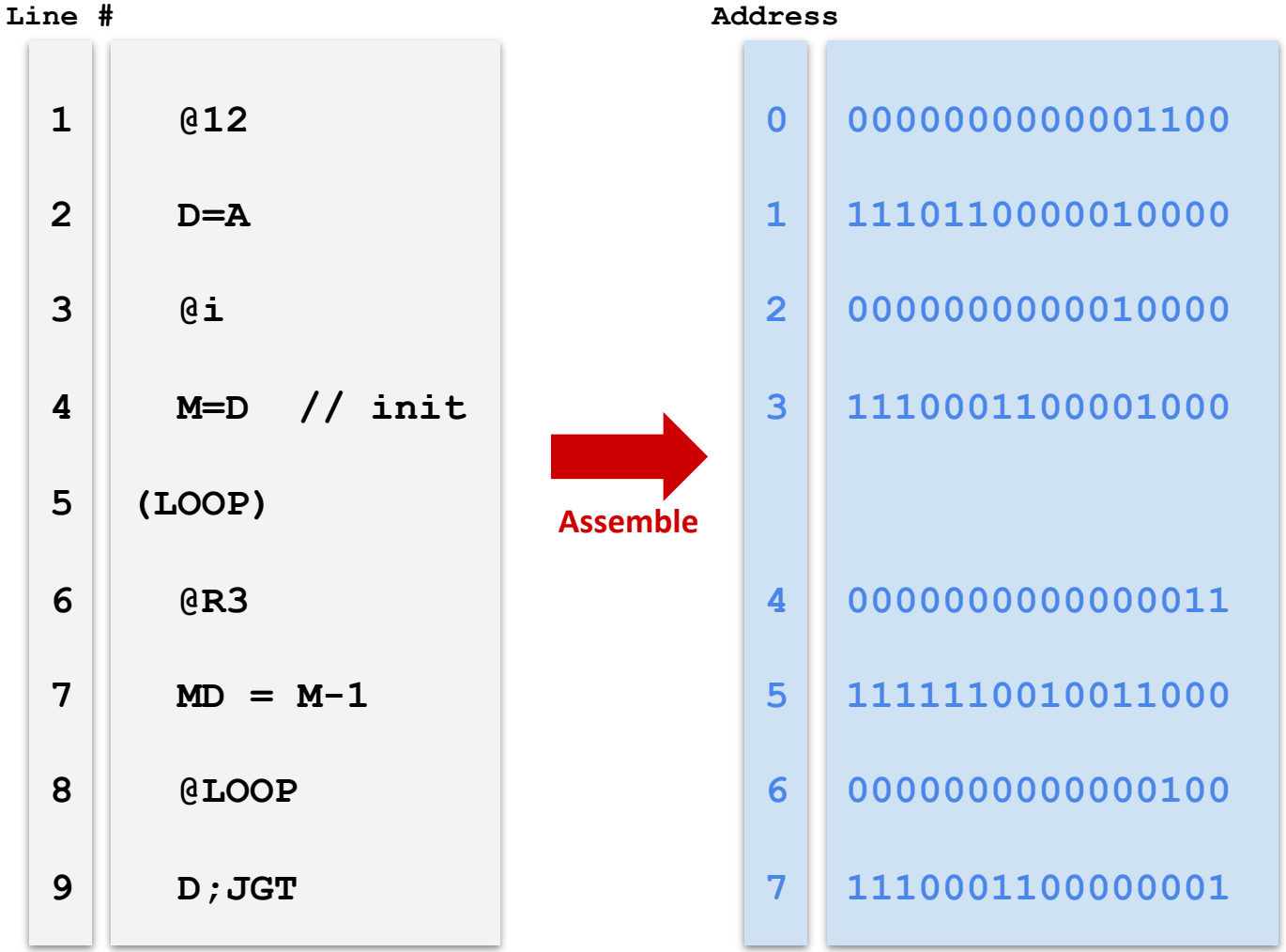
- Look up each value in the corresponding table

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

(when a=0) *comp* mnemonic	c1	c2	c3	c4	c5	c6	(when a=1) *comp* mnemonic
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

What Makes the Assembler's Job Difficult?



Difficulties for the Assembler

- Three broad concerns:

Parsing	Recognizing type of each instruction and 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 #
1     @LOOP
2     0;JMP
3     D=M
4   (LOOP)
5     @var
```

Symbols: Labels

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

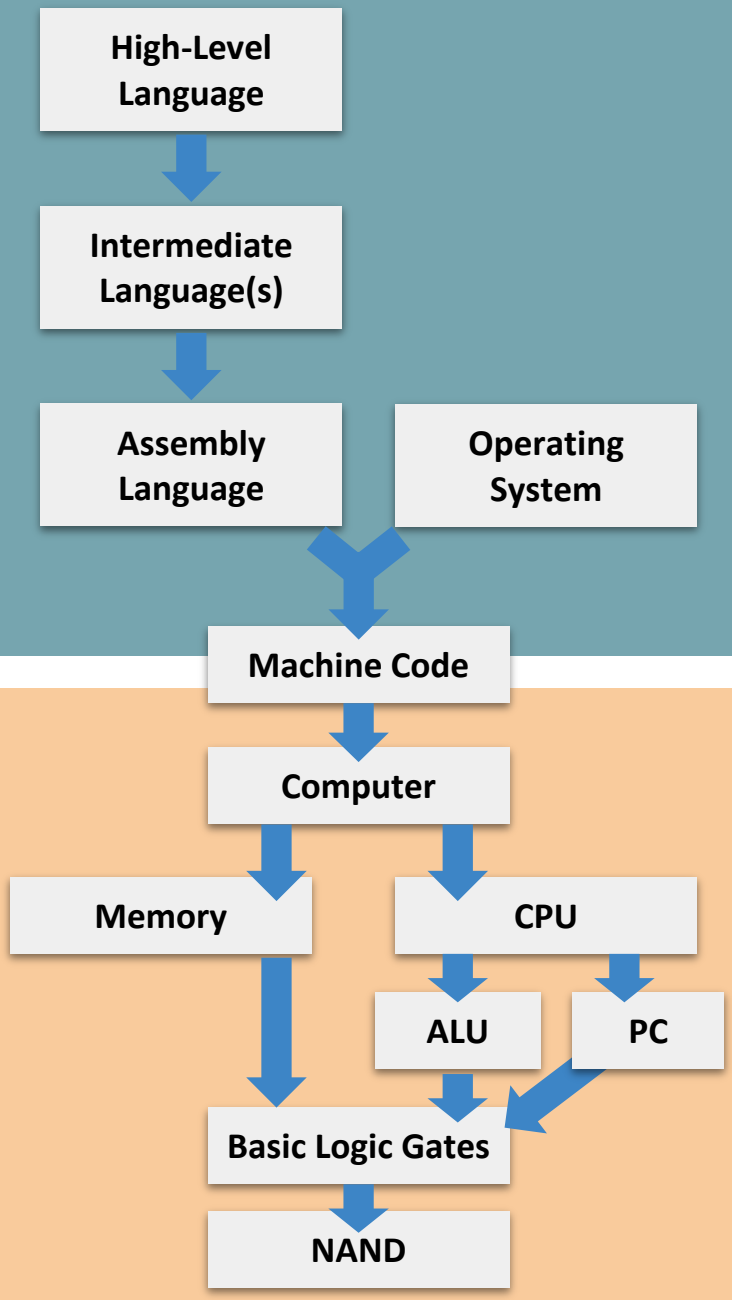
- Solution: Two passes
  - Pass 1: Populate symbol table by moving through file and ignoring anything that isn't a (LABEL) line
  - Pass 2: Go through file again, ignoring (LABEL) lines, encoding C-instructions, and encoding A-instructions according to symbol table lookup

Line #	
1	@LOOP
2	0;JMP
3	D=M
4	(LOOP)
5	@var

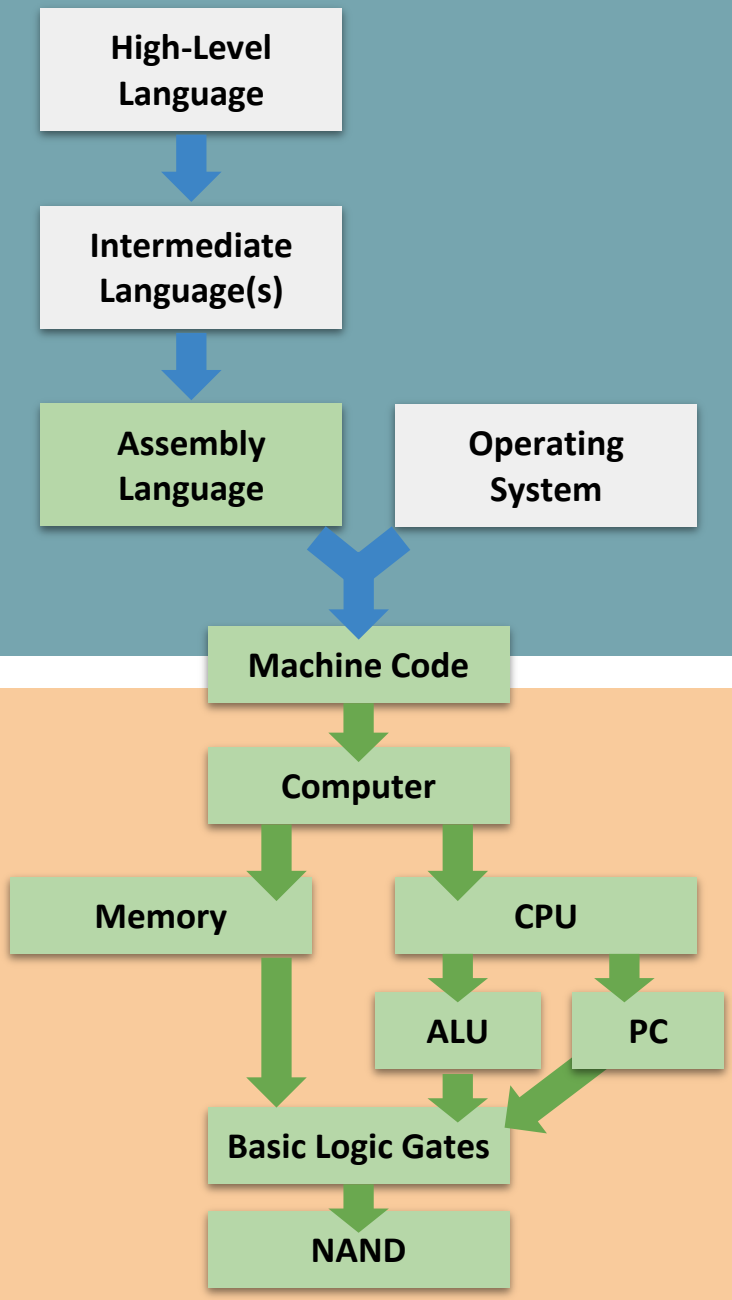
Lecture Outline

- Inside the Assembler
  - Producing Machine Code
  - Parsing, Symbols, Encoding
- **The Software Stack**
  - **Roadmap of Hardware and Software Components**
- Compilers Overview
  - Roadmap of Hardware and Software Components
- Hack CPU Logic Example: writeM
  - Project 6 CPU Logic Exercise

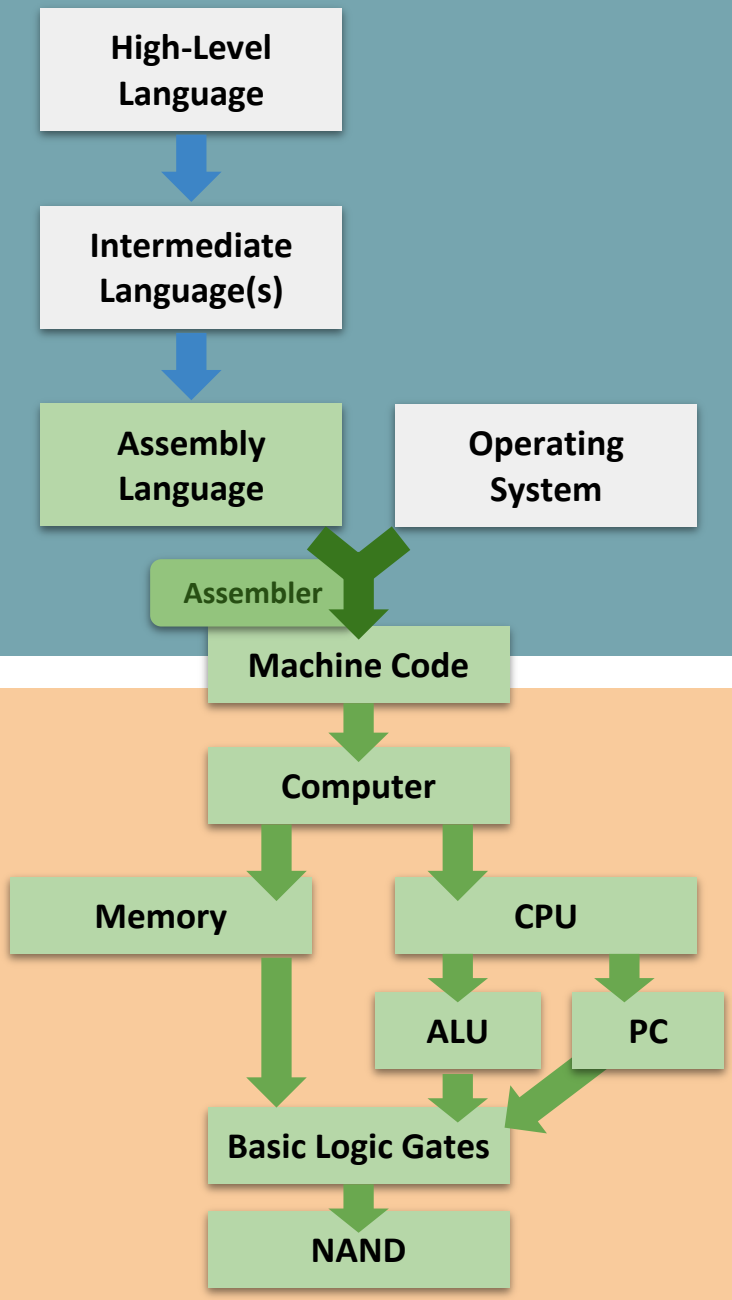
Roadmap


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

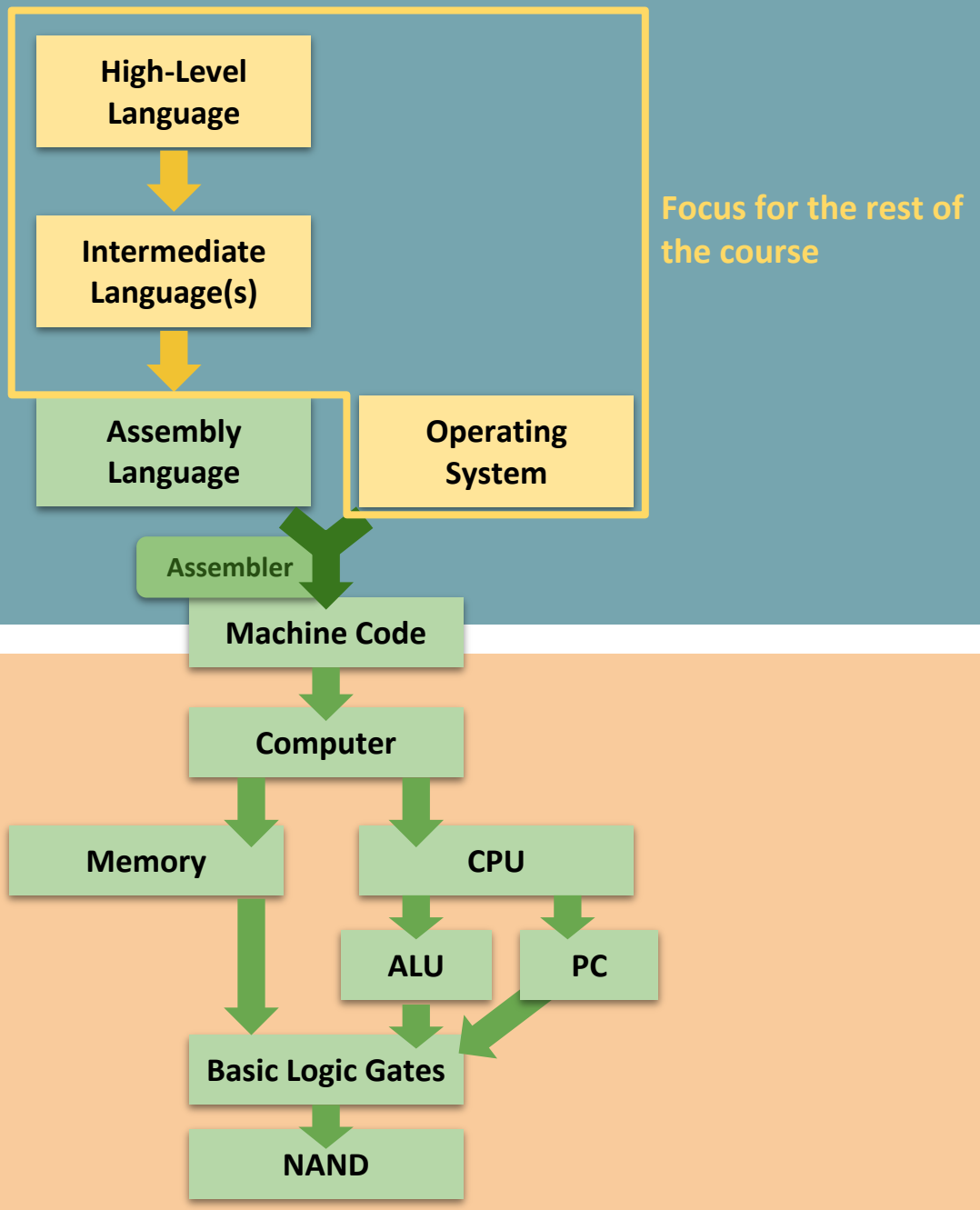
Roadmap


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

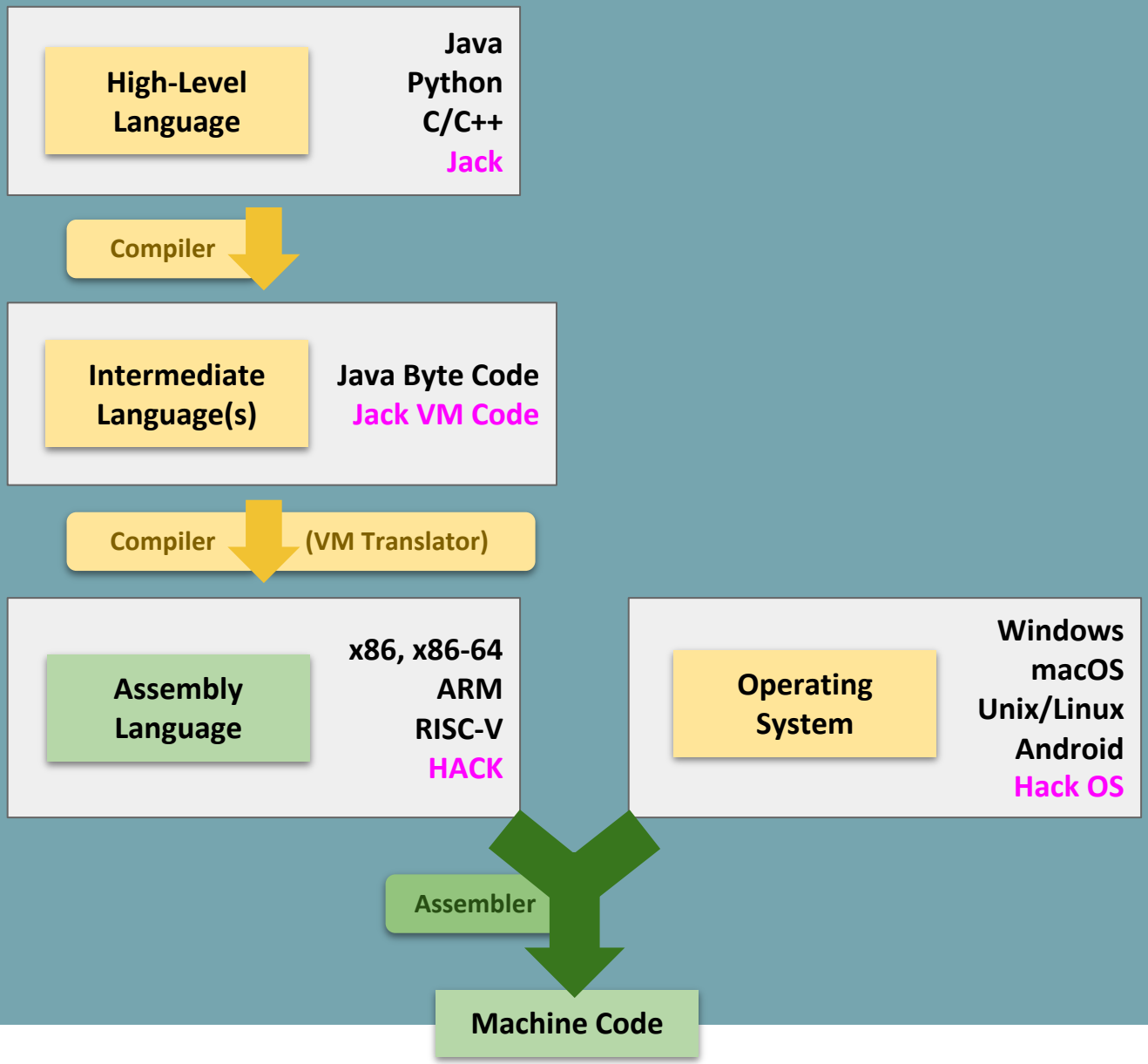
Roadmap


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

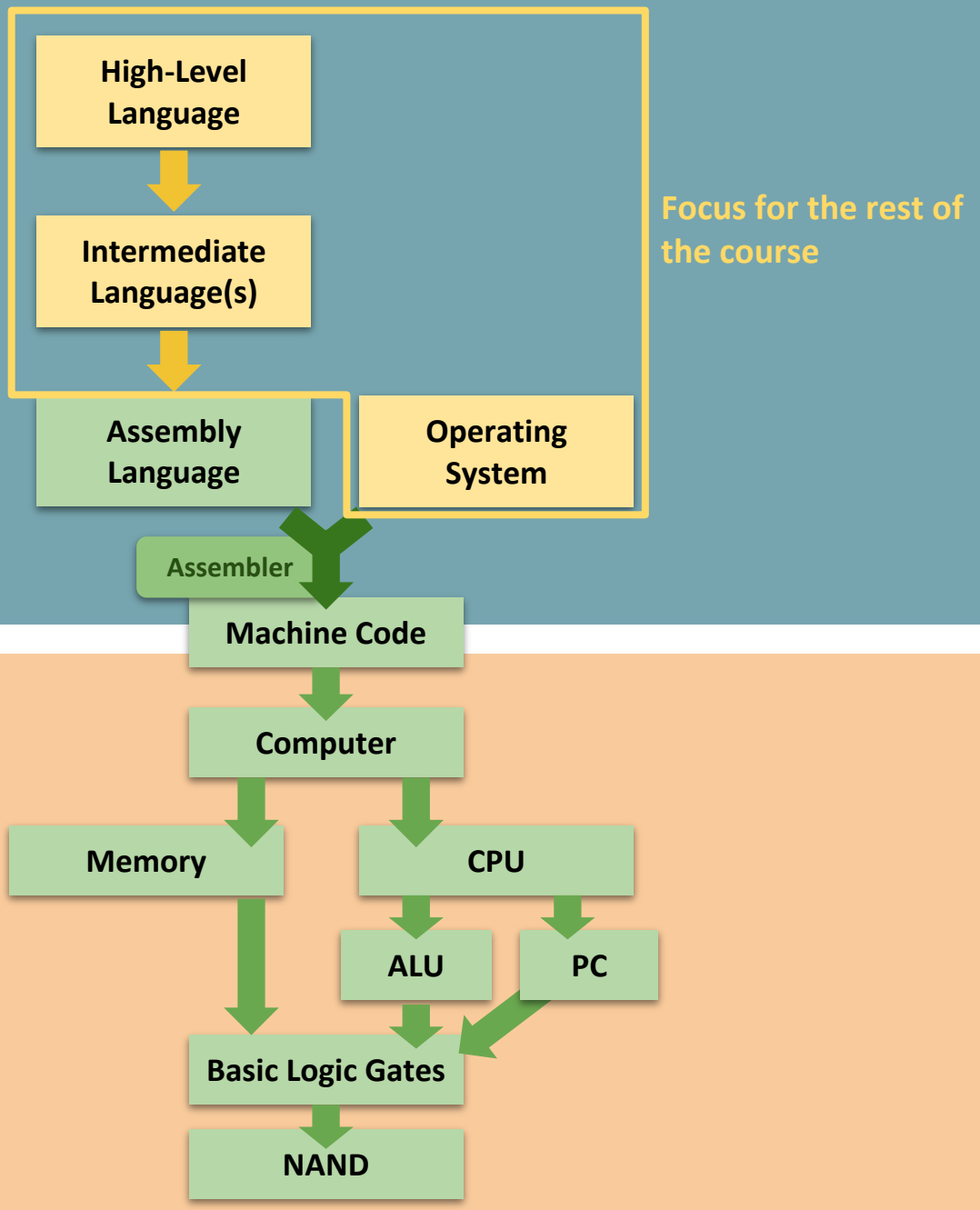


SOFTWARE

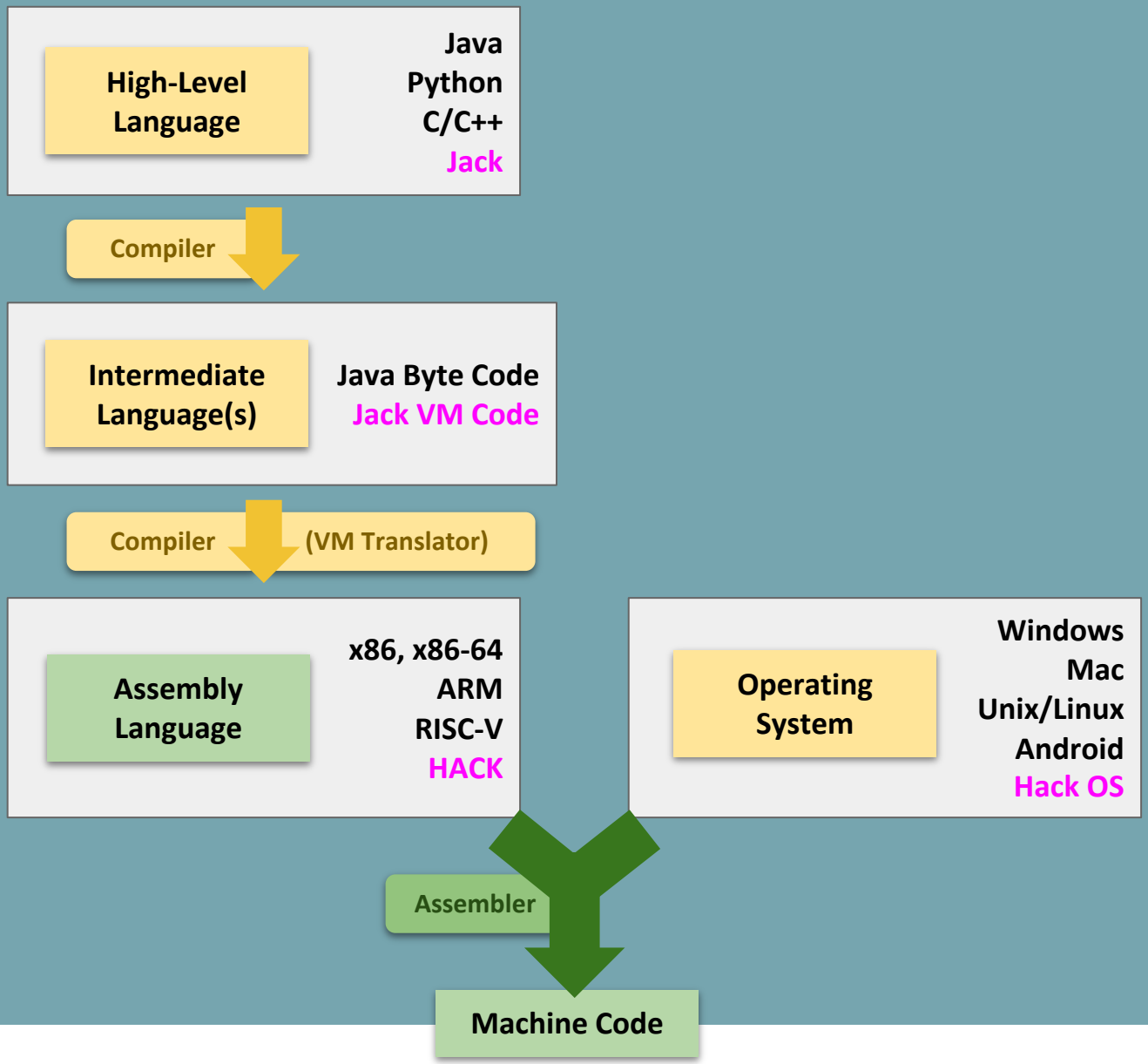
Lecture Outline

- Inside the Assembler
  - Producing Machine Code
  - Parsing, Symbols, Encoding
- The Software Stack
  - Roadmap of Hardware and Software Components
- **Compilers Overview**
  - **Roadmap of Hardware and Software Components**
- Hack CPU Logic Example: writeM
  - Project 6 CPU Logic Exercise

Roadmap

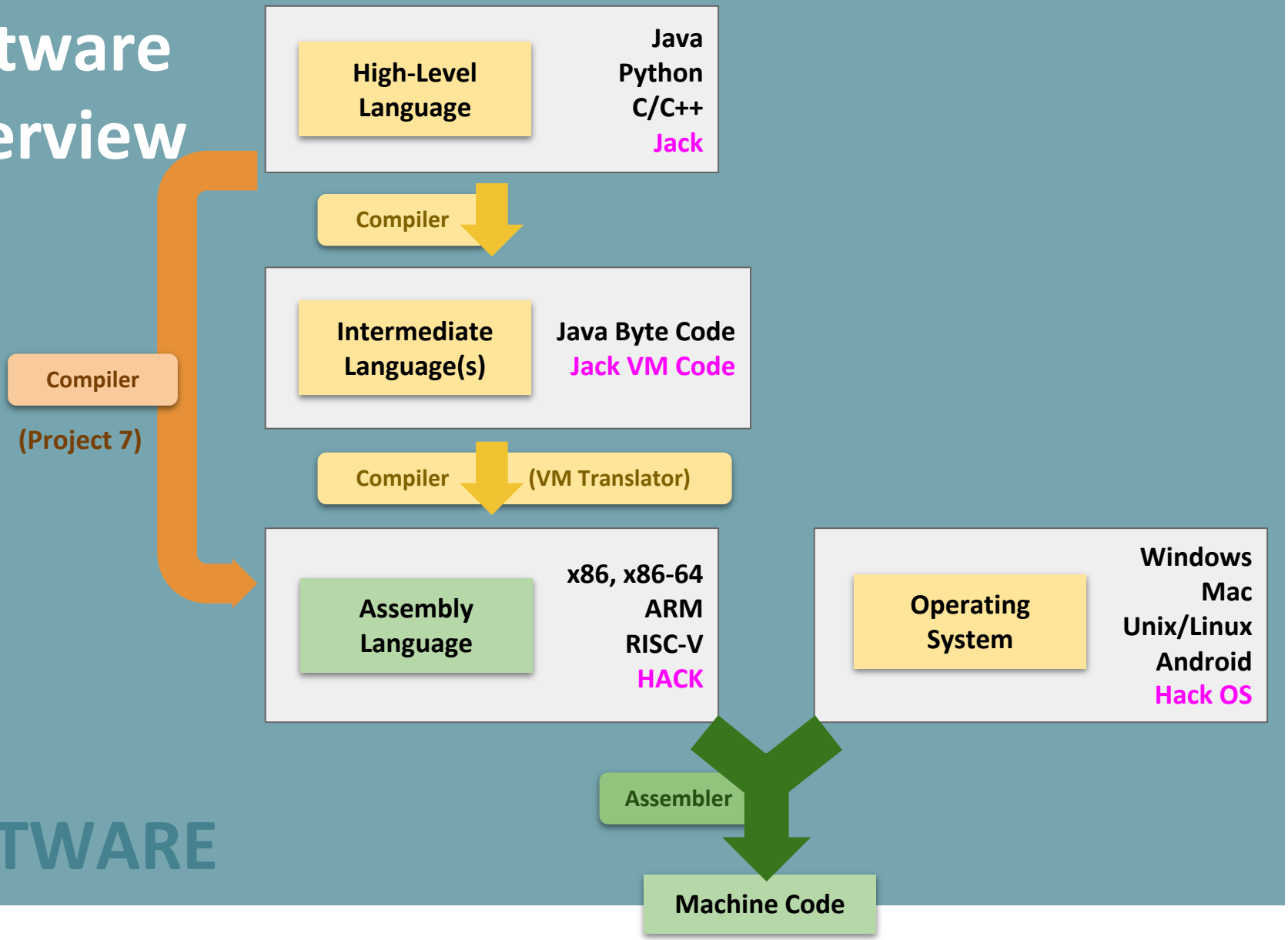

High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
Focus for the rest of the course
SOFTWARE
Assembler
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

Software Overview



SOFTWARE

Software Overview



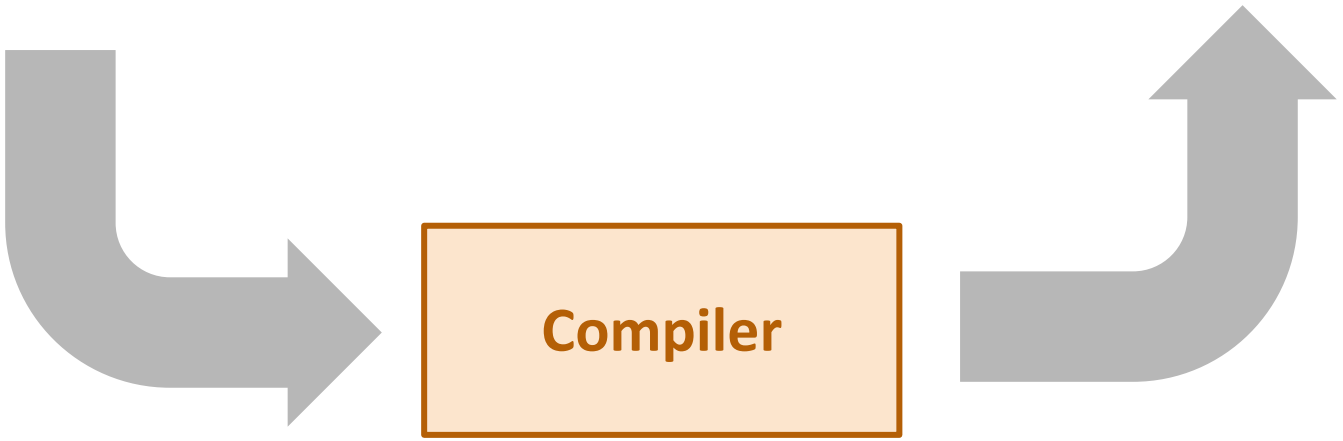
The Compiler: Goal

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



The Compiler: Goal

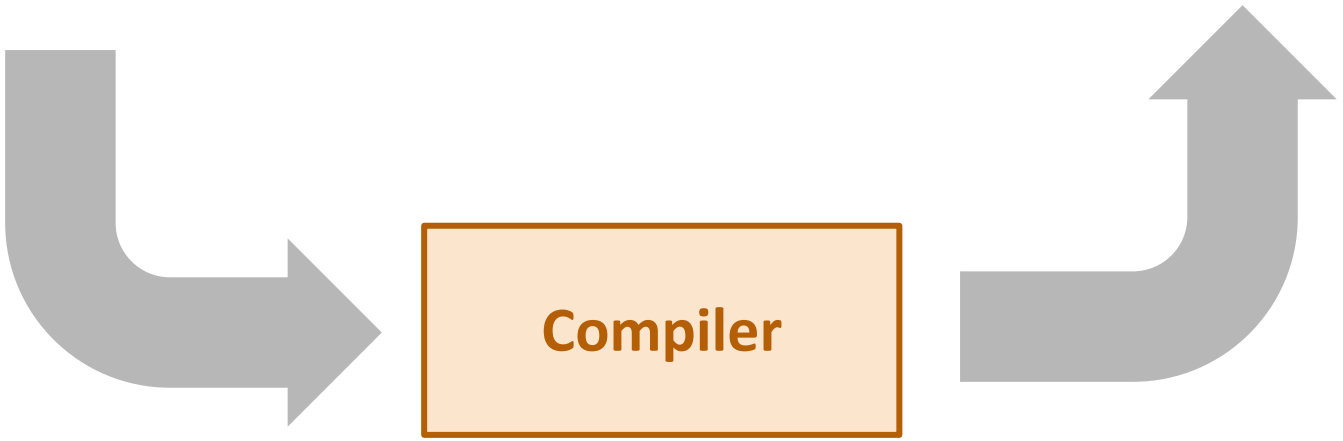
```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

Theory Definition: a string, from the set of strings making up a language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



The Compiler: Goal

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

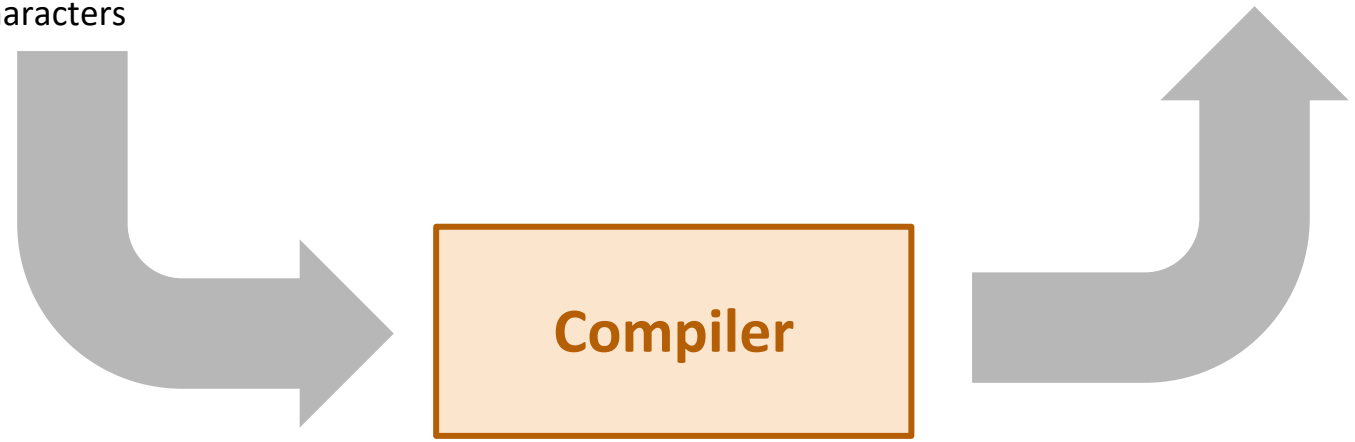
High-Level Language

Theory Definition: a string, from the set of strings making up a language

Practical Definition: a file containing a bunch of characters

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



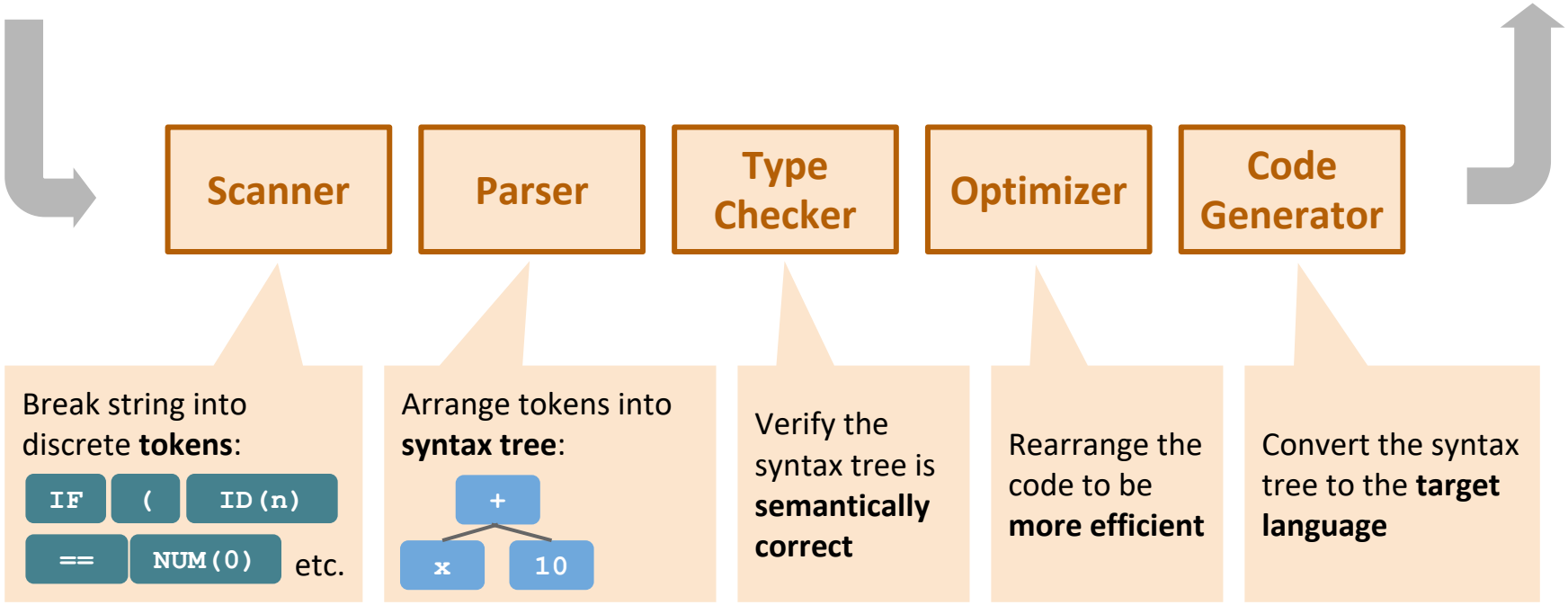
The Compiler: Implementation

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



Lecture Outline

- Inside the Assembler
  - Producing Machine Code
  - Parsing, Symbols, Encoding
- The Software Stack
  - Roadmap of Hardware and Software Components
- Compilers Overview
  - Roadmap of Hardware and Software Components
- **Hack CPU Logic Example: writeM**
  - **Project 6 CPU Logic Exercise**

Hack CPU Logic Example: writeM

- Example: Determine when **`writeM`** should be set to 1

- Step 1: What do we pay attention to?
  - **`writeM`** is related to whether we write to memory or not
  - We need to look up the destination bits specification from Chapter 4

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

Figure 4.4 The *dest* field of the *C*-instruction.

Hack CPU Logic Example: writeM

- Example: Determine when **`writeM`** should be set to 1

- Step 2: Determine logic for specification
  - Read the "Destination Specification" section of Chapter 4
  - Instruction bits:

```
1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3
```

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

Figure 4.4 The *dest* field of the *C*-instruction.

Hack CPU Logic Example: writeM

- Example: Determine when **`writeM`** should be set to 1

- Step 2: Determine logic for specification
  - Read the "Destination Specification" section of Chapter 4
  - Instruction bits:

```
1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3
```

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

Figure 4.4 The *dest* field of the *C*-instruction.

Hack CPU Logic Example: writeM

- Example: Determine when **`writeM`** should be set to 1

- Step 2: Determine logic for specification
  - Read the "Destination Specification" section of Chapter 4
  - Instruction bits:

    `1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3`
  - d3 determines if the output should be written to memory
  - Which bit of our instruction is that?
  - So **`writeM = instruction[3]`**?

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

Hack CPU Logic Example: writeM

- Example: Determine when **`writeM`** should be set to 1

- What's wrong with **`writeM = instruction[3]`**?
  - What happens if we have an A-instruction?
  - We only write to destinations in the case of a C-instruction
  - So, **`writeM = C-instruction & instruction[3]`**
  - Certain actions only occur on certain instruction types
  - You may have to include a check for instruction type in your logic

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

Hack CPU Implementation: Logic Sub Chips

- We provide three sub chips and tests that implement the control logic for the A Register, D Register, and PC
  - **`LoadAReg`** contains logic for loading the A Register
  - **`LoadDReg`** contains logic for loading the D Register
  - **`JumpLogic`** contains logic for determining if the PC should load, jump, or increment

- Implement and test these first, then use them in your CPU implementation
  - Intended to help you narrow the scope of bugs

Lecture 13 Wrap-up

- Midterm will be graded with feedback by Thursday (5/12)
- **Project 6: Mock Exam Problem & Building a Computer due this Thursday (5/12) at 11:59pm PDT**
- Thursday's Lecture Reading: Compilers Overview: Scanning and Parsing
- Please submit the mid-quarter feedback form if you haven't already