CS107, Lecture 14

Privacy and Trust, Optimization, & Basic Architecture

Privacy and Trust

- Our learning about assembly and program execution helps us better understand computer security (the protection of data, devices, and networks from disruption, harm, theft, unauthorized access or modification).
- Computer security is important in part because it enables privacy.
- In understanding computer security, it's essential to understand the context in which it comes up (privacy and trust).

Data Breaches

Privacy/trust example: data breaches

- California list of data security breaches: link
- How does a data breach make a customer feel?

Privacy

What is privacy? 4 possible framings in two categories:

Individualist: the value of privacy as an individual right

- Privacy as **control of information** – controlling how our private information is shared with others.
- Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.

Social: the value of privacy for a group

- Privacy as **social good** – social life would be unlivable without privacy.
- Privacy (protection) as based in **trust** – privacy enables trusting relationships

Privacy

Privacy as **control of information** – controlling how our information is communicated to others.

- Consent requires *free* choice with available alternatives and *informed* understanding of what is being offered.
- How many of you just skip past the terms of service for new online services you sign up for?
- Do you feel in control of your information with the services you choose to use? Why or why not? If you're working on a service, how can you respect privacy while achieving product goals?
- Control over personal data being collected (e.g. data exports from services you use, privacy dashboards, device privacy protections)

Privacy

Art. 1 GDPR

Subject-matter and objectives

1. This Regulation lays down rules relating to the protection of natural persons with regard to the processing of personal data and rules relating to the free movement of personal data.

2. This Regulation protects fundamental rights and freedoms of natural persons and in particular their right to the protection of personal data.

3. The free movement of personal data within the Union shall be neither restricted nor prohibited for reasons connected with the protection of natural persons with regard to the processing of personal data.

Media & Entertainment

Instagram launches "Data Download" tool to let you leave

Josh Constine / 9:44 AM PDT • April 24, 2018 Comment

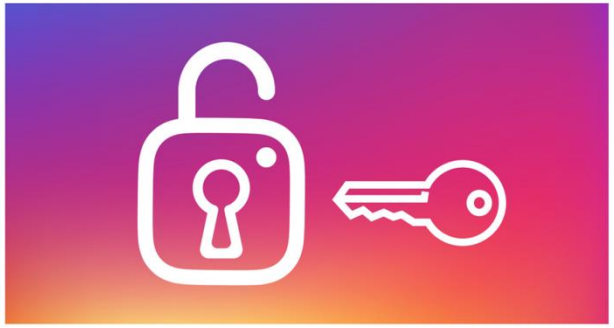


Image Credits: Bryce Durbin/TechCrunch /

TECH / APPLE / GOOGLE

Apple now lets you automatically transfer your iCloud Photo Library to Google Photos

/ Not everything can come along for the ride, though

By Mitchell Clark

Mar 3, 2021 at 1:10 PM PST

Google Report content on Google

Personal Data Removal Request Form

For privacy and data protection reasons (such as pursuant to the EU General Data Protection Regulation) you may have the right to ask for certain personal data relating to you to be removed.

This form is for requesting the removal of specific results for queries that include your name from Google Search. Google LLC is the controller responsible for the processing of personal data carried out in the context of determining the results shown by Google Search, as well as handling delisting requests sent through this form.

Privacy

Privacy as **autonomy** – capacity to choose/decide for ourselves what is valuable.

- Links to autonomy over our own lives and our ability to lead them as we choose.
- Do you feel that your autonomy is always respected when using products and services? Why or why not?

"[P]rivacy is valuable because it acknowledges our respect for persons as autonomous beings with the capacity to love, care and like—in other words, persons with the potential to freely develop close relationships" (Innes 1992)

Individualist Models of Privacy

Privacy as **autonomy** and privacy as **control over information** focus the value of privacy at an individual level.

- Individual privacy can conflict with interests of society or the state.
- Many debates over ”privacy vs. security” – whether one should be sacrificed for the other
  - Apple v. FBI case re: unlocking iPhones (link)
  - Debates around encryption (link)
- Where do your beliefs fall in balancing privacy and security? When (if at all) is it ok to sacrifice one, and how much?

Privacy

Privacy as **social good** – social life would be unlivable without privacy.

- Privacy has a social value in bringing about the kind of society we want to live in.
- What would society look like without privacy?

Privacy

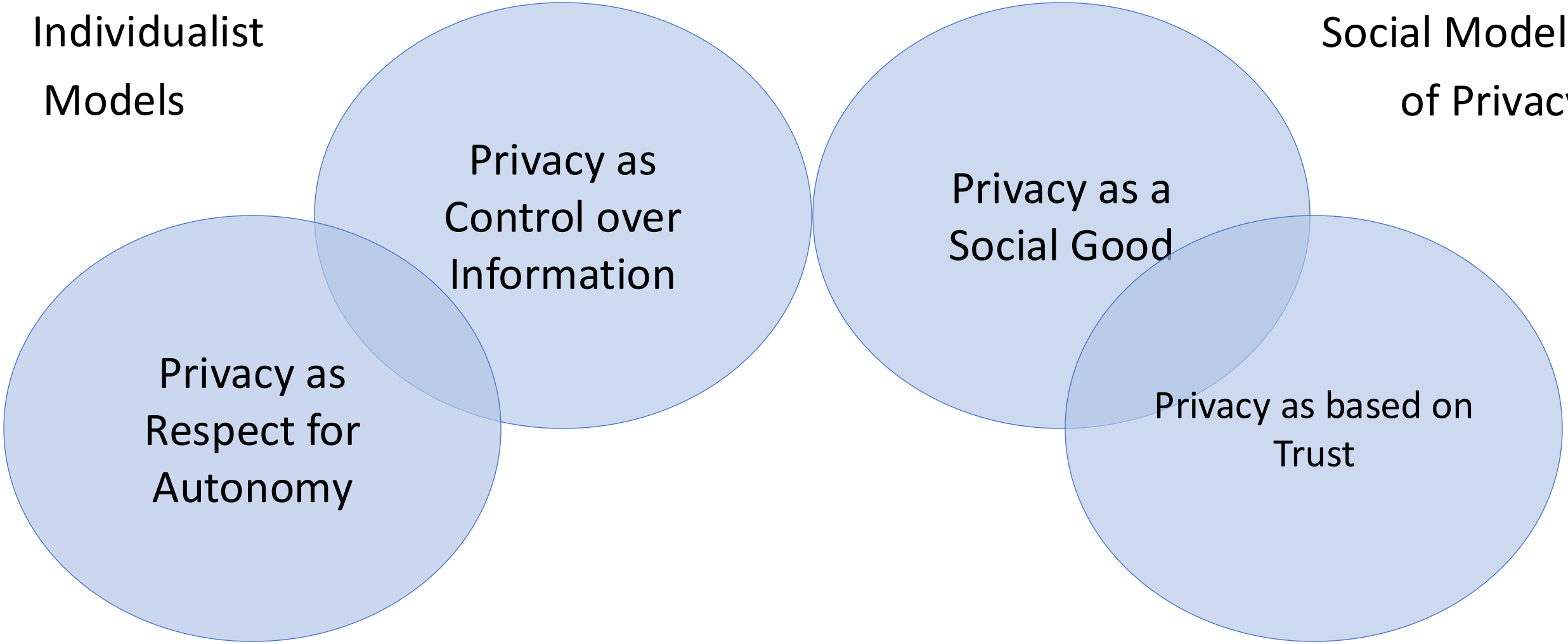
Privacy (protection) as based in **trust** – privacy enables trusting relationships

- Privacy may help enable trusting relationships essential for cooperation.
  - For instance, a *fiduciary*: someone who stands in a legal or ethical relationship of trust with another person (or group). The fiduciary must act for the benefit of and in the best interest of the other person. E.g. tax filer with access to your bank account
    - Should anyone who has access to personal info have a *fiduciary* responsibility? (Richards & Hartzog 2020).
- This model of privacy stresses the essential relationship of trust placed in any holder of personal data and the responsibilities that result from this trust.

Models of Privacy

Individualist Models

Social Models of Privacy



Who Should We Trust?

Both security and privacy rely on trusted people (who administer security, perform penetration tests, submit vulnerabilities to databases, or keep private information secret). The final piece of the security puzzle is understanding trust.

Trust = Reliance + Risk of Betrayal

What makes trust unique to relationships between people is that trust exposes one to being *betrayed or being let down* (Baier 1986).

Penetration Testing & Trust

Penetration testing is the practice of encouraging or hiring security researchers / contractors to find vulnerabilities in one's own code or system.

- Position of trust – tester is given access to the system and encouraged to find exploitable vulnerabilities, expected to share what they have found with you.
- Means *relying on* their skill at finding vulnerabilities and *trusting* that their ethical compass will lead them to tell you and to act as a trustworthy *fiduciary* (guardian of your interests).

In Assignment 5, you have the opportunity to explore this further!

Loss of Privacy

Loss of privacy can cause us various harms, including:

- ***Aggregation*:** combining personal information from various sources to build a profile of someone
- ***Exclusion:*** not knowing how our information is being used, or being unable to access or modify it (Google removing personal info from search – link)
- ***Secondary Use***: using your information for purposes other than what was intended without permission.

Mitigation: Differential Privacy

Differential privacy is a formal measure of privacy for datasets to try and protect individuals from aggregation by making them harder to identify (Dwork 2008).

- Imagine a large database, e.g., a medical database, with personal information and records of past activity tied to a name.
- The records might be useful for research purposes, or to train a machine learning model to predict future health outcomes, but what if giving access to the records exposed the privacy of individual person's health records?
- Differential privacy adds inconsequential noise (e.g., changing a birthday from 2001 to 2002) or removes records to make individuals harder to identify while preserving the utility of the dataset overall.

Trust Models

In every evaluation of privacy, we can ask: who is trusted? Who is distrusted? Does this model concentrate trust (and therefore power) in a single individual or small group, or does it distribute trust?

Differential Privacy's Trust Model

Differential privacy assumes that the only threat to privacy is an *external user querying the database* who must be prevented from aggregating data that could identify a user.

- In other words, the *trust model* of differential privacy is that the database owners and maintainers are to be fully trusted, and no one else.
- But is that the only threat? Differential privacy does not protect against improper use by people with full access to data or against leaks of the whole database, which may be the primary data exposure risks.

Differential privacy also does not question the assumption that amassing & storing large amounts of personal data is worth the risk of inevitable leaks (Rogaway 2015).

GCC Optimizations

Optimization

Most of what you need to do with optimization can be summarized by:

1) If doing something seldom and only on small inputs, do whatever is simplest to code, understand, and debug
2) If doing things a lot, or on big inputs, make the primary algorithm's Big-O cost reasonable
3) **Let gcc do its magic from there**
4) Optimize explicitly as a last resort

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if success
Body
Update
Jump to test
```

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Test
No jump
Body
Update
Jump to test
Test
No jump
Body
Update
Jump to test
...
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

```
for (int i = 0; i < n; i++)          // n = 100

Initialization
Jump to test
Test
Jump to body
Body
Update
Test
Jump to body
Body
Update
Test
Jump to body
...
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

GCC For Loop Output

GCC Common For Loop Output

```
Initialization
Test
Jump past loop if passes
Body
Update
Jump to test
```

Possible Alternative

```
Initialization
Jump to test
Body
Update
Test
Jump to body if success
```

Which instructions are better when n = 0? n = 1000?

for (int i = 0; i < n; i++)

Optimizing Instruction Counts

- Both versions have the same **static** **instruction count** (# of written instructions).
- But they have different **dynamic** **instruction counts** (# of executed instructions when program is run).
  - If n = 0, left (GCC common output) is best b/c fewer instructions
  - If n is large, right (alternative) is best b/c fewer instructions
- The compiler may emit a static instruction count that is several times longer than an alternative, but it may be more efficient if loop executes many times.
- Does the compiler *know* that a loop will execute many times? (in general, no)
- So what if our code had loops that always execute a small number of times? How do we know when gcc makes a bad decision?
- (take EE108, EE180, CS316 for more!)

Optimizations

- **Conditional Moves** can sometimes eliminate “branches” (jumps), which are particularly inefficient on modern computer hardware.
- Processors try to *predict* the future execution of instructions for maximum performance. This is difficult to do with jumps.

GCC Optimization

- Today, we'll be comparing two levels of optimization in the gcc compiler:
  - `gcc -O0  // mostly just literal translation of C`
  - `gcc -O2  // enable nearly all reasonable optimizations`
  - (we also use –Og, like –O0 but more debugging friendly)

- There are other custom and more aggressive levels of optimization, e.g.:
  - `-O3     //more aggressive than O2, trade size for speed`
  - `-Os     //optimize for size`
  - `-Ofast  //disregard standards compliance (!!)`

- Exhaustive list of gcc optimization-related flags:
  - https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compiler optimizations

How many GCC optimization levels are there?

Asked 11 years, 3 months ago Active 5 months ago Viewed 62k times

109

35

How many GCC optimization levels are there?

I tried gcc -O1, gcc -O2, gcc -O3, and gcc -O4

If I use a really large number, it won't work.

However, I have tried

```
gcc -O100
```

and it compiled.

How many optimization levels are there?

Gcc supports numbers up to 3. Anything above is interpreted as 3

https://stackoverflow.com/questions/1778538/how-many-gcc-optimization-levels-are-there

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Constant Folding

Constant Folding pre-calculates constants at compile-time where possible.

```
int seconds = 60 * 60 * 24 * n_days;
```

Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = sqrt(2.0);
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

Constant Folding: Before (-O0)

```
00000000000011b9 <fold>:
    11b9:   55                      push   %rbp
    11ba:   48 89 e5                mov    %rsp,%rbp
    11bd:   41 54                   push   %r12
    11bf:   53                      push   %rbx
    11c0:   48 83 ec 30             sub    $0x30,%rsp
    11c4:   89 7d cc                mov    %edi,-0x34(%rbp)
    11c7:   c7 45 ec 07 01 00 00    movl   $0x107,-0x14(%rbp)
    11ce:   8b 45 ec                mov    -0x14(%rbp),%eax
    11d1:   48 98                   cltq
    11d3:   89 c2                   mov    %eax,%edx
    11d5:   89 d0                   mov    %edx,%eax
    11d7:   c1 e0 02                shl    $0x2,%eax
    11da:   01 d0                   add    %edx,%eax
    11dc:   89 45 e8                mov    %eax,-0x18(%rbp)
    11df:   48 8b 05 2a 0e 00 00    mov    0xe2a(%rip),%rax        # 2010 <_IO_stdin_used+0x10>
    11e6:   66 48 0f 6e c0          movq   %rax,%xmm0
    11eb:   e8 b0 fe ff ff          callq  10a0 <sqrt@plt>
    11f0:   f2 0f 2c c0             cvttsd2si %xmm0,%eax
    11f4:   89 45 e4                mov    %eax,-0x1c(%rbp)
    11f7:   8b 45 ec                mov    -0x14(%rbp),%eax
    11fa:   0f af 45 cc             imul   -0x34(%rbp),%eax
    11fe:   41 89 c4                mov    %eax,%r12d
    1201:   b8 15 00 00 00          mov    $0x15,%eax
    1206:   99                      cltd
    1207:   f7 7d e4                idivl  -0x1c(%rbp)
    120a:   89 c2                   mov    %eax,%edx
    120c:   8b 45 ec                mov    -0x14(%rbp),%eax
    120f:   01 d0                   add    %edx,%eax
    1211:   48 63 d8                movslq %eax,%rbx
    1214:   48 8d 3d ed 0d 00 00    lea    0xded(%rip),%rdi        # 2008 <_IO_stdin_used+0x8>
    121b:   e8 20 fe ff ff          callq  1040 <strlen@plt>
    1220:   8b 55 e8                mov    -0x18(%rbp),%edx
    1223:   48 63 d2                movslq %edx,%rdx
    1226:   48 0f af c2             imul   %rdx,%rax
    122a:   48 01 d8                add    %rbx,%rax
    122d:   48 83 e8 37             sub    $0x37,%rax
    1231:   48 c1 e8 02             shr    $0x2,%rax
    1235:   44 01 e0                add    %r12d,%eax
    1238:   48 83 c4 30             add    $0x30,%rsp
    123c:   5b                      pop    %rbx
    123d:   41 5c                   pop    %r12
    123f:   5d                      pop    %rbp
    1240:   c3                      retq
```

Constant Folding: After (-O2)

```
0000000000011b0 <fold>:
    11b0:   69 c7 07 01 00 00       imul   $0x107,%edi,%eax
    11b6:   05 a5 06 00 00          add    $0x6a5,%eax
    11bb:   c3                      retq
```

What is the consequence of this for you as a programmer? What should you do differently or the same knowing that compilers can do this for you?

GCC Optimizations

- Constant Folding
- **Common Sub-expression Elimination**
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```

Common Sub-Expression Elimination

Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

```
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
// = 2 * a * a + param1 * a * a
```

```
0000000000011b0 <subexp>:  // param1 in %edi, param2 in %esi
    11b0:  lea    0x107(%rsi),%eax      // %eax stores a
    11b6:  imul   %eax,%edi             // param1 * a
    11b9:  lea    (%rdi,%rax,2),%esi    // 2 * a + param1 * a
    11bc:  imul   %esi,%eax             // a * (2 * a + param1 * a)
    11bf:  retq
```

Common Sub-Expression Elimination

Why should we bother saving repeated calculations in variables if the compiler has common subexpression elimination?

- The compiler may not always be able to optimize every instance. Plus, it can help reduce redundancy!
- Makes code more readable!

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- Tail Recursion
- Loop Unrolling

Dead Code

Dead code elimination removes code that doesn't serve a purpose:

```
if (param1 < param2 && param1 > param2) {
    printf("This test can never be true!\n");
}

// Empty for loop
for (int i = 0; i < 1000; i++);

// If/else that does the same operation in both cases
if (param1 == param2) {
    param1++;
} else {
    param1++;
}

// If/else that more trickily does the same operation in both cases
if (param1 == 0) {
    return 0;
} else {
    return param1;
}
```

Dead Code: Before (-O0)

```
00000000000011a9 <dead_code>:
    11a9: 55                          push   %rbp
    11aa: 48 89 e5                    mov    %rsp,%rbp
    11ad: 48 83 ec 20                 sub    $0x20,%rsp
    11b1: 89 7d ec                    mov    %edi,-0x14(%rbp)
    11b4: 89 75 e8                    mov    %esi,-0x18(%rbp)
    11b7: 8b 45 ec                    mov    -0x14(%rbp),%eax
    11ba: 3b 45 e8                    cmp    -0x18(%rbp),%eax
    11bd: 7d 19                       jge    11d8 <dead_code+0x2f>
    11bf: 8b 45 ec                    mov    -0x14(%rbp),%eax
    11c2: 3b 45 e8                    cmp    -0x18(%rbp),%eax
    11c5: 7e 11                       jle    11d8 <dead_code+0x2f>
    11c7: 48 8d 3d 36 0e 00 00        lea    0xe36(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    11ce: b8 00 00 00 00              mov    $0x0,%eax
    11d3: e8 68 fe ff ff              callq  1040 <printf@plt>
    11d8: c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%rbp)
    11df: eb 04                       jmp    11e5 <dead_code+0x3c>
    11e1: 83 45 fc 01                 addl   $0x1,-0x4(%rbp)
    11e5: 81 7d fc e7 03 00 00        cmpl   $0x3e7,-0x4(%rbp)
    11ec: 7e f3                       jle    11e1 <dead_code+0x38>
    11ee: 8b 45 ec                    mov    -0x14(%rbp),%eax
    11f1: 3b 45 e8                    cmp    -0x18(%rbp),%eax
    11f4: 75 06                       jne    11fc <dead_code+0x53>
    11f6: 83 45 ec 01                 addl   $0x1,-0x14(%rbp)
    11fa: eb 04                       jmp    1200 <dead_code+0x57>
    11fc: 83 45 ec 01                 addl   $0x1,-0x14(%rbp)
    1200: 83 7d ec 00                 cmpl   $0x0,-0x14(%rbp)
    1204: 75 07                       jne    120d <dead_code+0x64>
    1206: b8 00 00 00 00              mov    $0x0,%eax
    120b: eb 03                       jmp    1210 <dead_code+0x67>
    120d: 8b 45 ec                    mov    -0x14(%rbp),%eax
    1210: c9                          leaveq
    1211: c3                          retq
```

Dead Code: After (-O2)

```
00000000000011b0 <dead_code>:
    11b0:   8d 47 01                lea    0x1(%rdi),%eax
    11b3:   c3                      retq
```

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- **Strength Reduction**
- Code Motion
- Tail Recursion
- Loop Unrolling

Strength Reduction

Strength reduction changes divide to multiply, multiply to add/shift, and mod to AND to avoid using instructions that cost many cycles (multiply and divide).

```
int a = param2 * 32;
int b = a * 7;
int c = b / 2;
int d = param2 % 2;

for (int i = 0; i <= param2; i++) {
    c += param1[i] + 0x107 * i;
}
return c + d;
```

Shifting into Shifts

- `int a = param2 * 32;`

  Becomes:

- `int a = param2 * 32;`

- `int b = a * 7;`

  Becomes:

- `int b = a + (a << 2) + (a << 1);`

- `int c = b / 2;`

  Becomes

- `int c = b >> 1`

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- Tail Recursion
- Loop Unrolling

Code Motion

Code motion moves code outside of a loop if possible.

```
for (int i = 0; i < n; i++) {
  sum += arr[i] + foo * (bar + 3);
}
```

Common subexpression elimination deals with expressions that appear multiple times in the code. Here, the expression appears once, but is calculated each loop iteration, even though none of its values change during the loop.

Code Motion

Code motion moves code outside of a loop if possible.

```
int temp = foo * (bar + 3);
for (int i = 0; i < n; i++) {
  sum += arr[i] + temp;
}
```

Moving it out of the loop allows the computation to happen only once.

Practice: GCC Optimization

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? What (if anything) can GCC do?

Practice: GCC Optimization

```
int char_sum(char *s) {
    int sum = 0;
    for (size_t i = 0; i < strlen(s); i++) {
        sum += s[i];
    }
    return sum;
}
```

What is the bottleneck? What (if anything) can GCC do?

strlen is called every loop iteration – code motion can pull it out of the loop

Tail Recursion

Tail recursion is an example of where GCC can identify recursive patterns that can be more efficiently implemented iteratively.

```
long factorial(int n) {
  if (n <= 1) {
      return 1;
  }
  else return n * factorial(n - 1);
}
```

Tail recursion example: Lab6 bonus

Recall the factorial problem from assembly lectures:

```
unsigned int factorial(unsigned int n) {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

What happens with **`factorial(-1)`**?

- Infinite recursion → Literal stack overflow!
- Compiled with `-0g`!

Factorial: -Og vs -O2

```
401146 <+0>:  cmp    $0x1,%edi
401149 <+3>:  jbe    0x40115b <factorial+21>
40114b <+5>:  push   %rbx
40114c <+6>:  mov    %edi,%ebx
40114e <+8>:  lea    -0x1(%rdi),%edi
401151 <+11>:callq  0x401146 <factorial>
401156 <+16>:imul   %ebx,%eax
401159 <+19>:pop    %rbx
40115a <+20>:retq
40115b <+21>:mov    $0x1,%eax
401160 <+26>:retq
```

-O2:

- What happened?
- Did the compiler "fix" the infinite recursion?

```
4011e0 <+0>:  mov    $0x1,%eax
4011e5 <+5>:  cmp    $0x1,%edi
4011e8 <+8>:  jbe    0x4011fd <factorial+29>
4011ea <+10>:nopw   0x0(%rax,%rax,1)
4011f0 <+16>:mov    %edi,%edx
4011f2 <+18>:sub    $0x1,%edi
4011f5 <+21>:imul   %edx,%eax
4011f8 <+24>:cmp    $0x1,%edi
4011fb <+27>:jne    0x4011f0 <factorial+16>
4011fd <+29>:retq
```

Breaking Down the –O2

```
4011e0 <+0>:  mov    $0x1,%eax          # Initialize %eax with 1.
4011e5 <+5>:  cmp    $0x1,%edi          # Compare input value (%edi) with 1.
4011e8 <+8>:  jbe    0x4011fd <factorial+29>   # If input <= 1 (unsigned check), jump to return.
4011ea <+10>: nopw   0x0(%rax,%rax,1)   # No operation (probably for alignment).
4011f0 <+16>: mov    %edi,%edx          # Copy current value of %edi to %edx.
4011f2 <+18>: sub    $0x1,%edi          # Decrement %edi.
4011f5 <+21>: imul   %edx,%eax          # Multiply %eax by %edx and store result in %eax.
4011f8 <+24>: cmp    $0x1,%edi          # Compare decremented value of %edi with 1.
4011fb <+27>: jne    0x4011f0 <factorial+16>   # If %edi is not 1, repeat the multiplication.
4011fd <+29>: retq                      # Return with the result in %eax.
```

-02:

- Recursive -> Iterative
- No Stack Overflow, Saves Memory and Operations

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- Code Motion
- Tail Recursion
- **Loop Unrolling**

Loop Unrolling

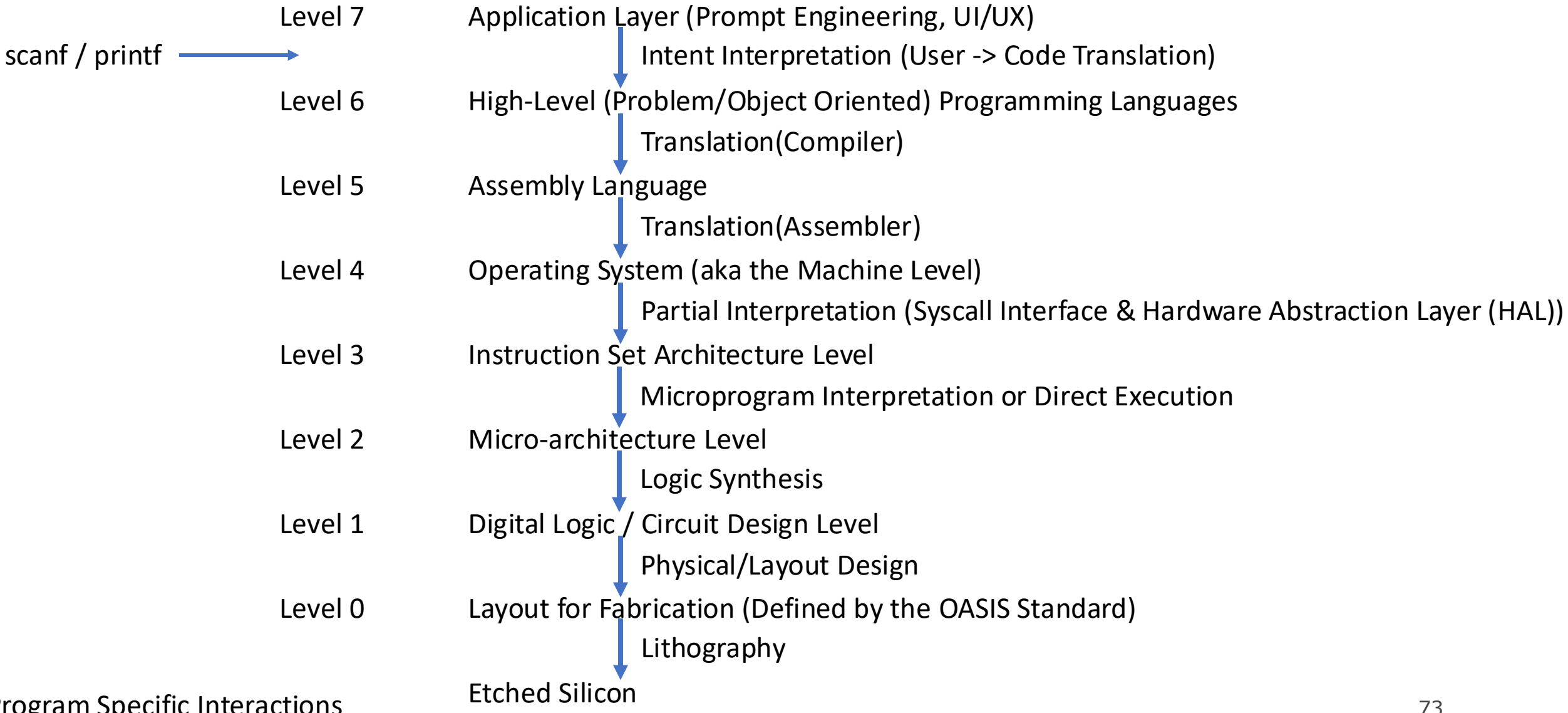
Loop Unrolling: Do **n** loop iterations' worth of work per actual loop iteration, so we save ourselves from doing the loop overhead (test and jump) every time, and instead incur overhead only every n-th time.

```
for (int i = 0; i <= n - 4; i += 4) {
     sum += arr[i];
     sum += arr[i + 1];
     sum += arr[i + 2];
     sum += arr[i + 3];
} // after the loop handle any leftovers
```

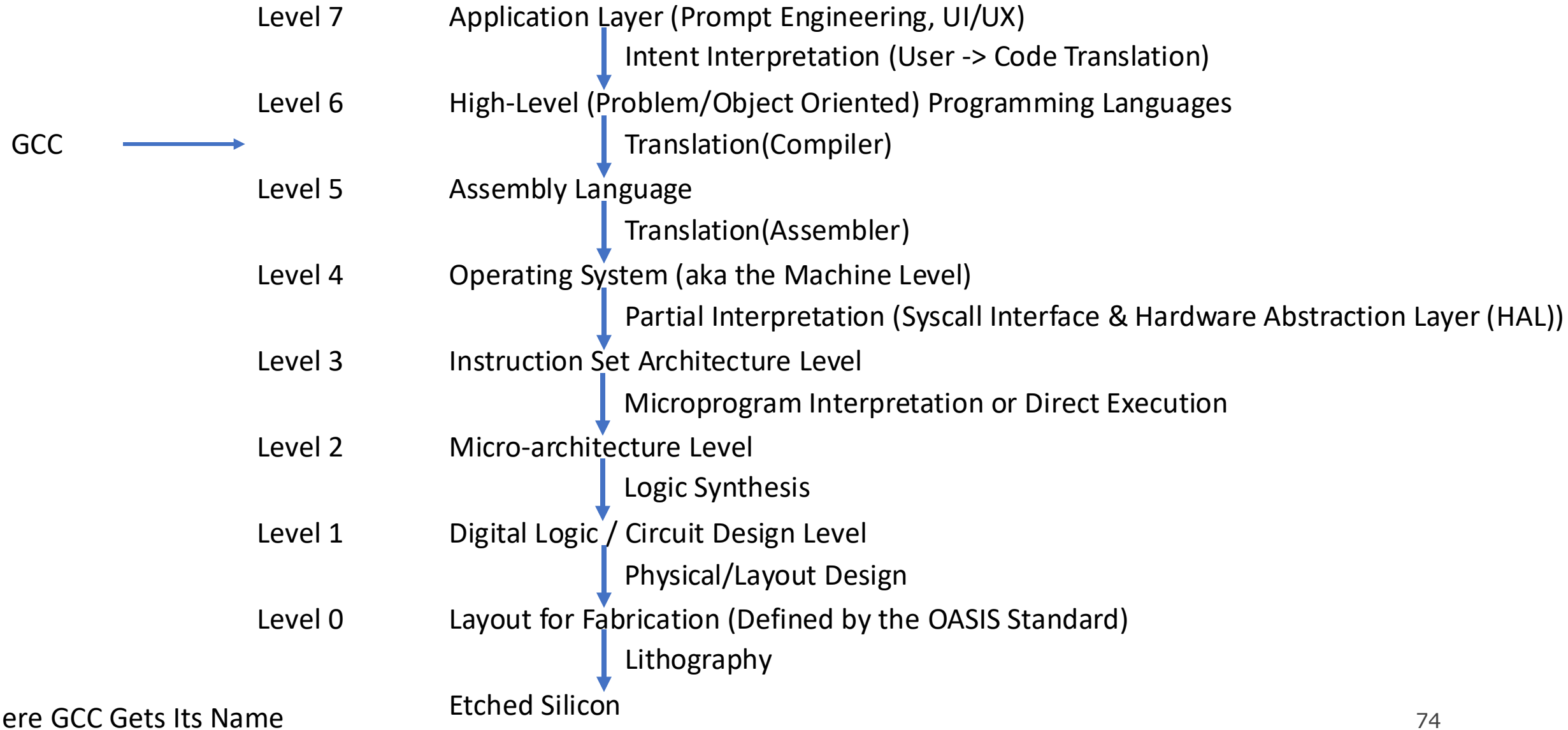
Some Extra Reading

Into the Architecture!

Programming Levels

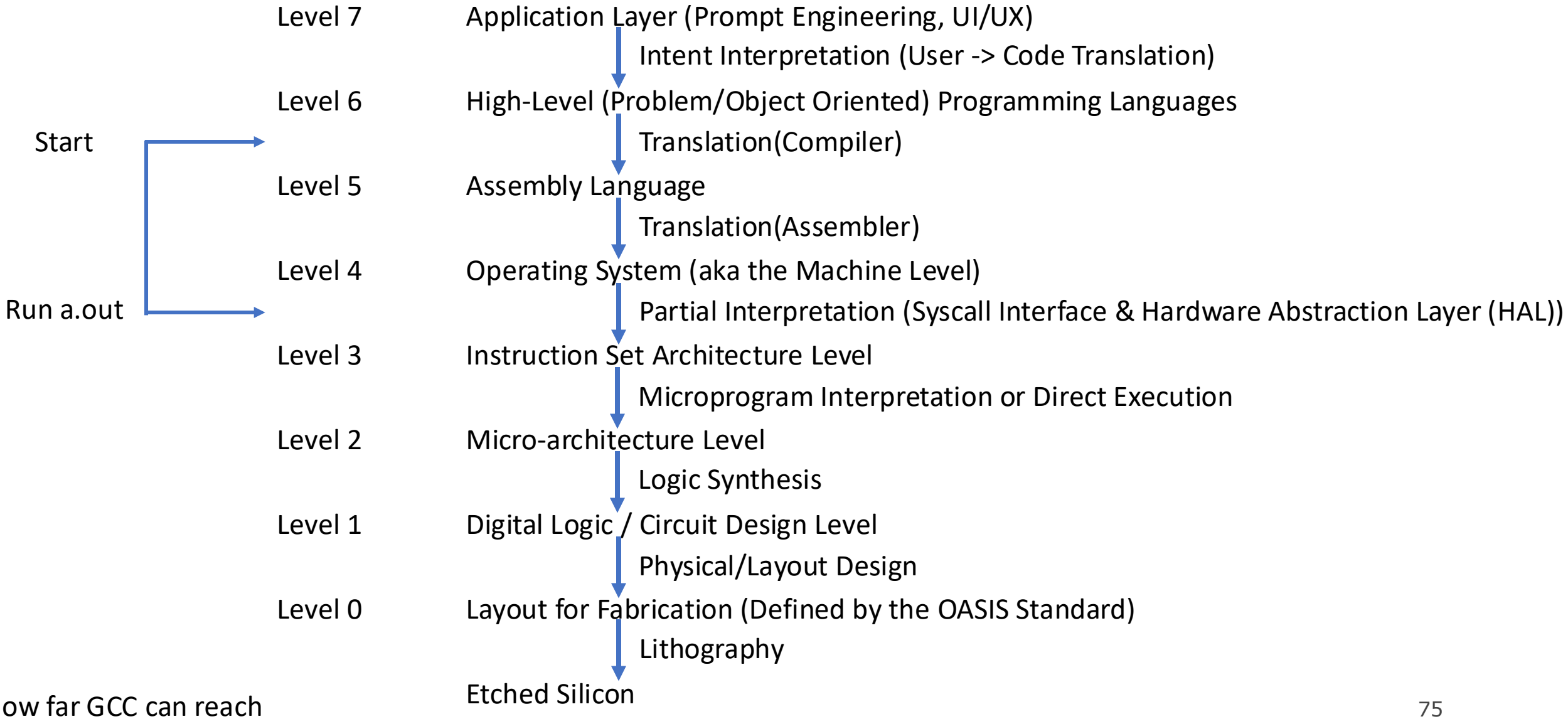


Programming Levels

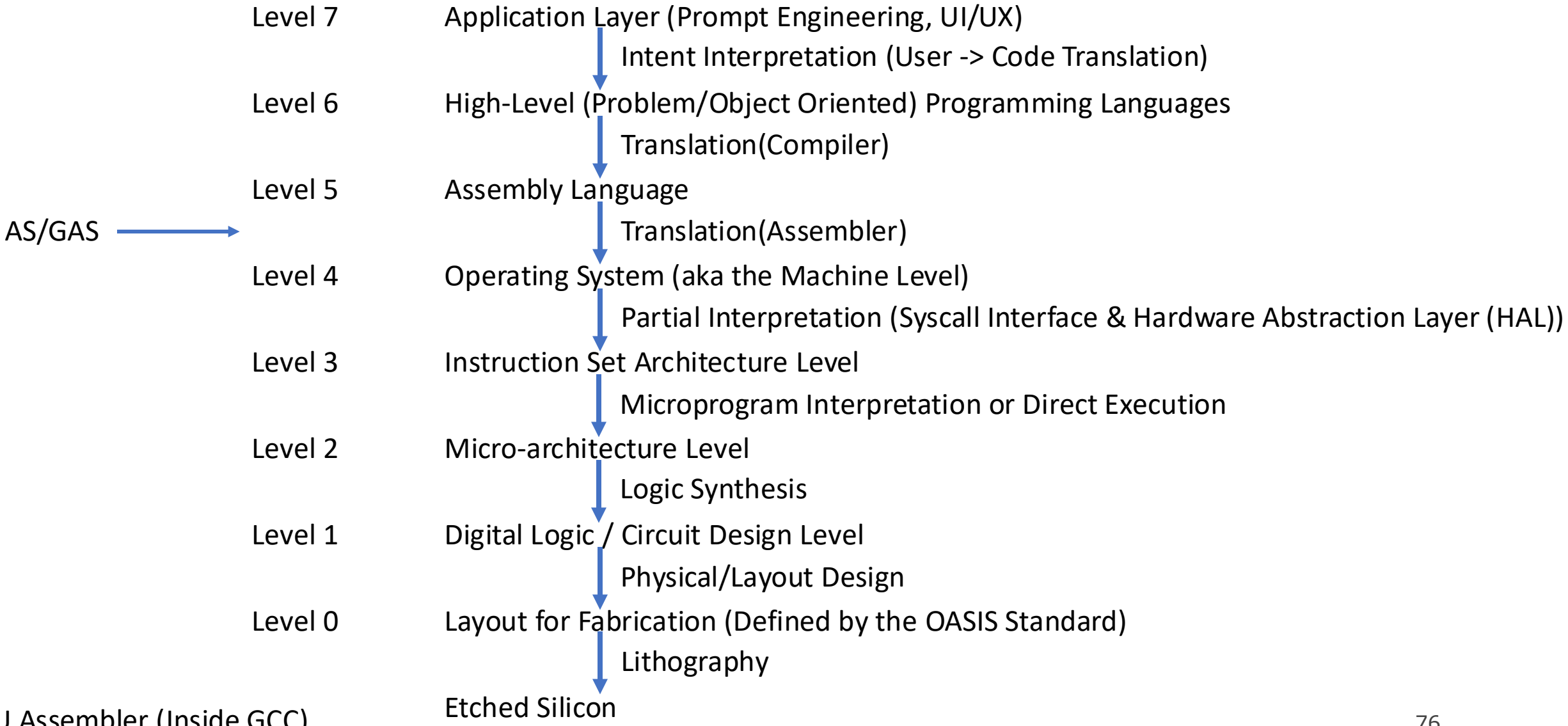


Where GCC Gets Its Name

Programming Levels

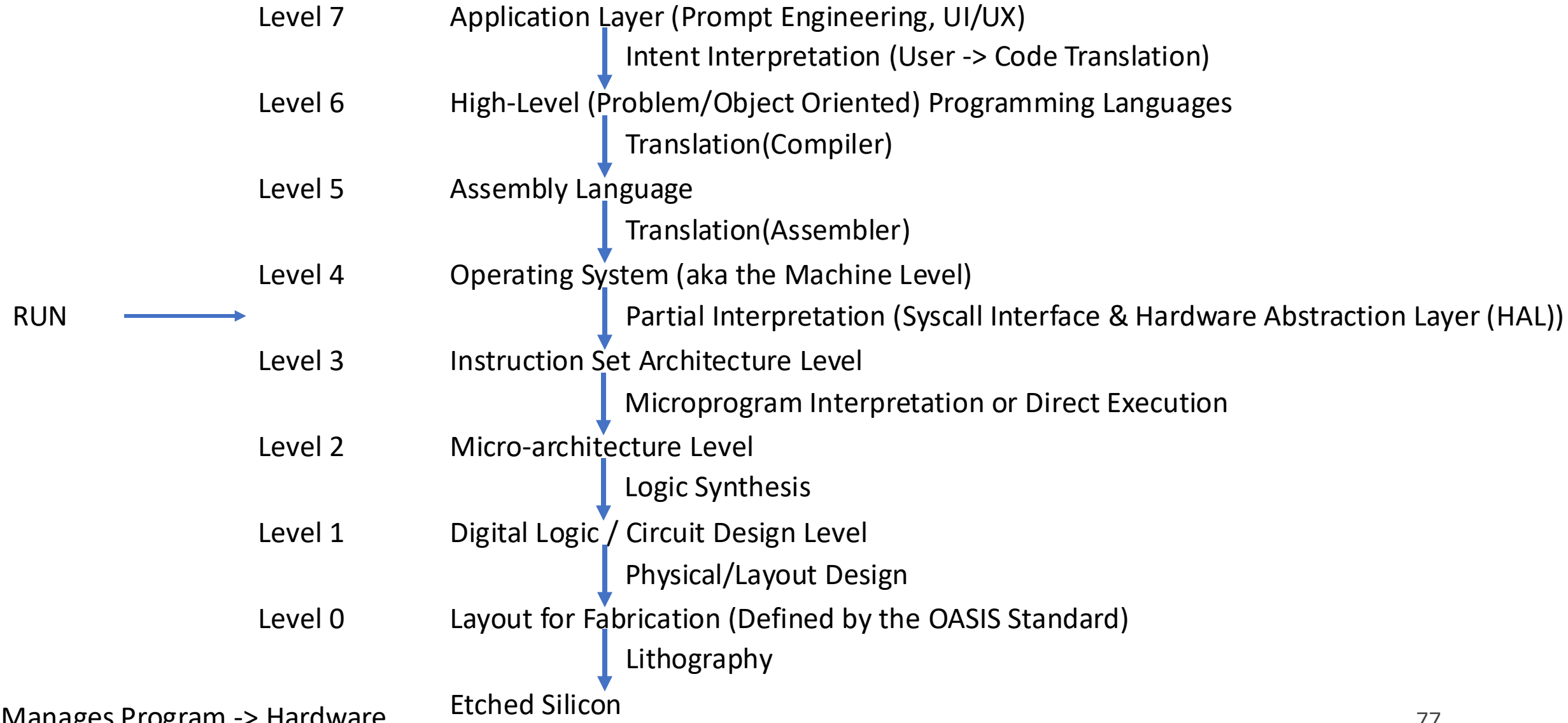


Programming Levels



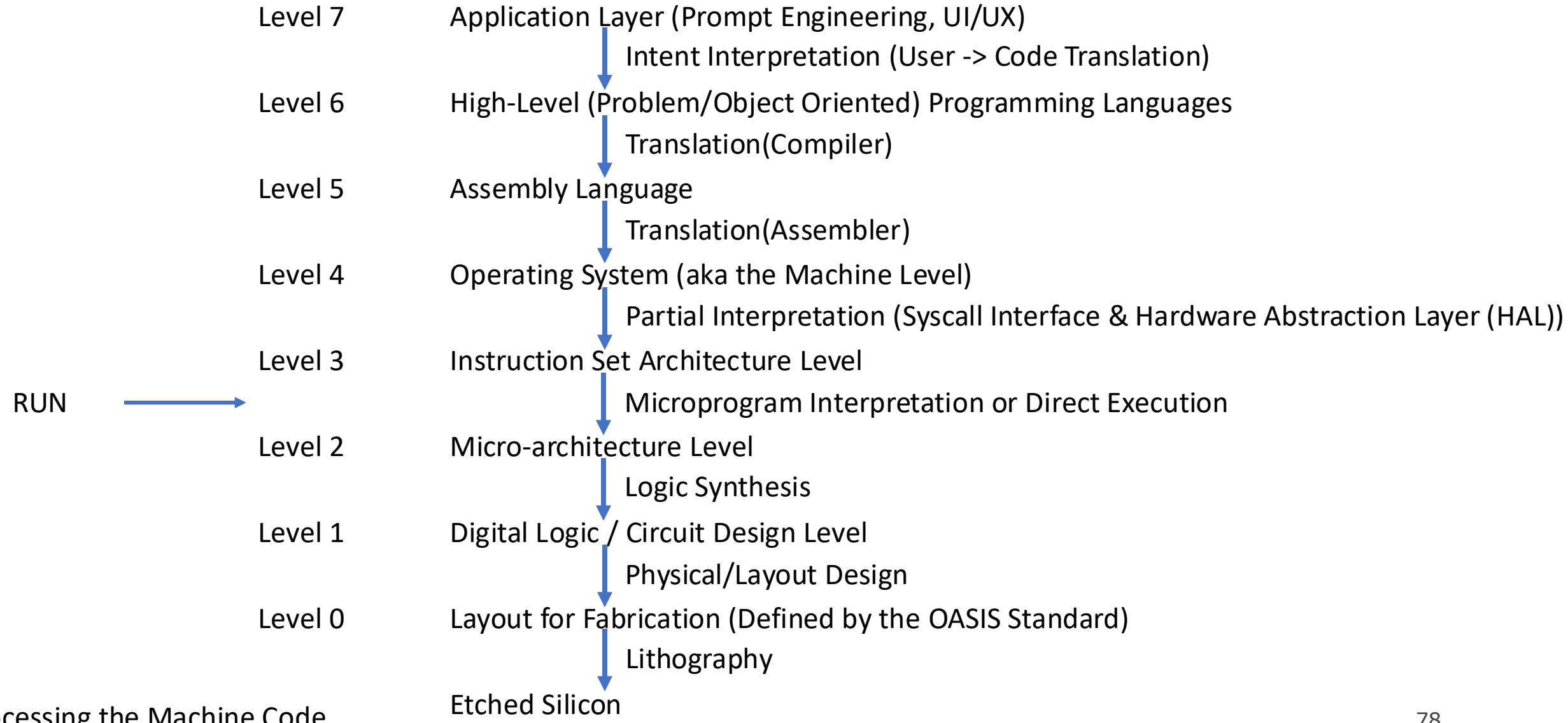
GNU Assembler (Inside GCC)

Programming Levels

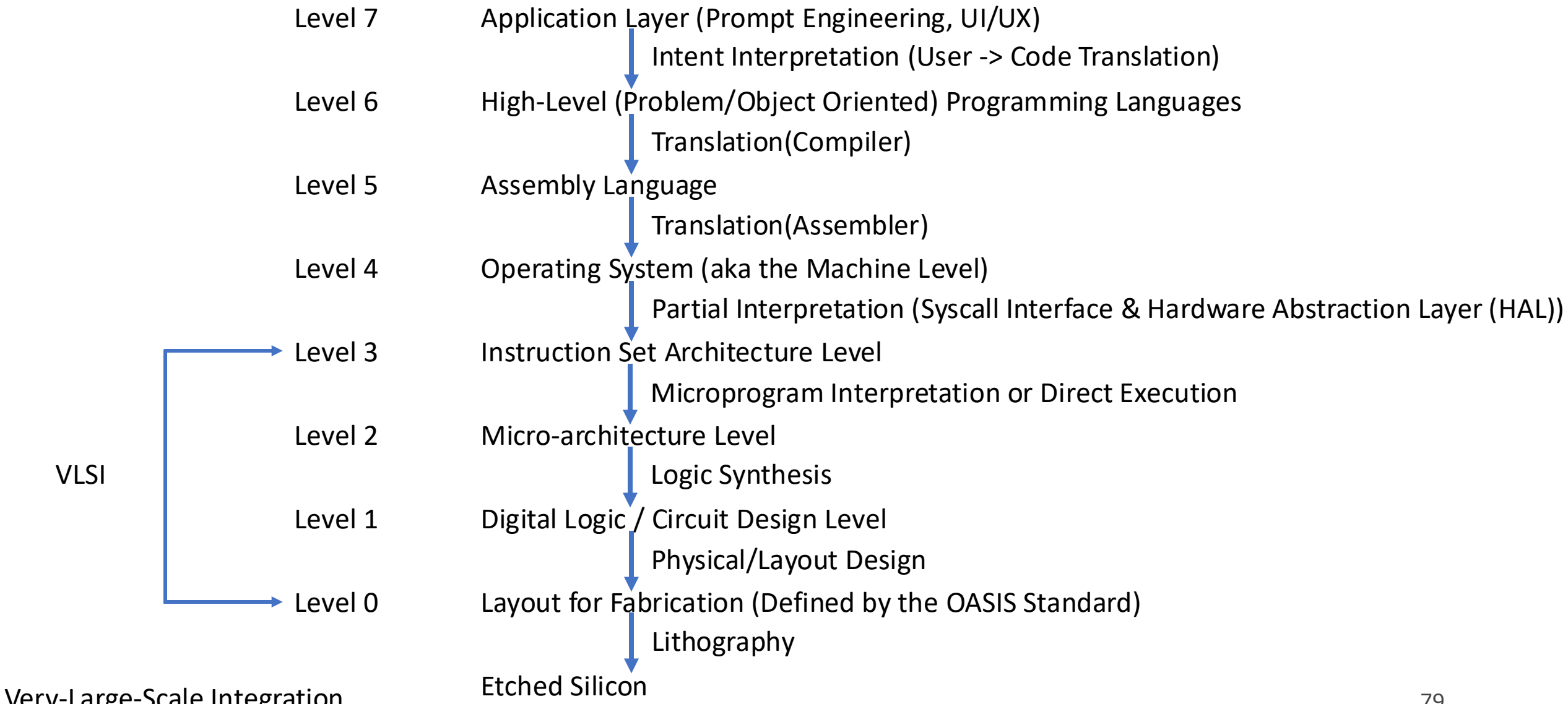


OS Manages Program -> Hardware

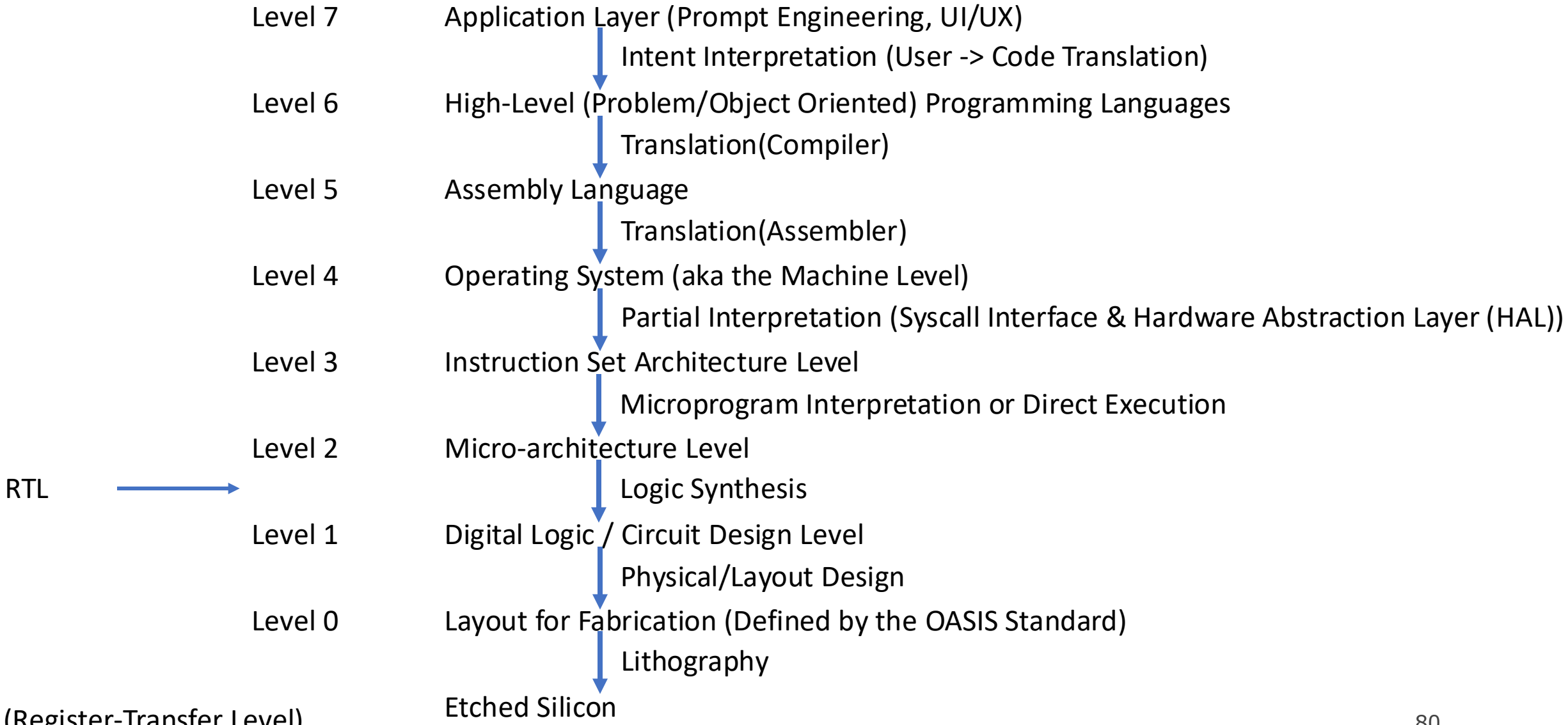
Programming Levels



Programming Levels



Programming Levels



RTL (Register-Transfer Level)

Programming Levels

Level 7 — Application Layer (Prompt Engineering, UI/UX)

↓ Intent Interpretation (User -> Code Translation)

Level 6 — High-Level (Problem/Object Oriented) Programming Languages

↓ Translation(Compiler)

Level 5 — Assembly Language

↓ Translation(Assembler)

Level 4 — Operating System (aka the Machine Level)

↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 3 — Instruction Set Architecture Level

↓ Microprogram Interpretation or Direct Execution

Level 2 — Micro-architecture Level

↓ Logic Synthesis

Level 1 — Digital Logic / Circuit Design Level

↓ Physical/Layout Design

Level 0 — Layout for Fabrication (Defined by the OASIS Standard)

↓ Lithography

Etched Silicon

Many Steps →

Floorplanning

Programming Levels

Level 7 — Application Layer (Prompt Engineering, UI/UX)

↓ Intent Interpretation (User -> Code Translation)

Level 6 — High-Level (Problem/Object Oriented) Programming Languages

↓ Translation(Compiler)

Level 5 — Assembly Language

↓ Translation(Assembler)

Level 4 — Operating System (aka the Machine Level)

↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 3 — Instruction Set Architecture Level

↓ Microprogram Interpretation or Direct Execution

Level 2 — Micro-architecture Level

↓ Logic Synthesis

Level 1 — Digital Logic / Circuit Design Level

↓ Physical/Layout Design

Level 0 — Layout for Fabrication (Defined by the OASIS Standard)

↓ Lithography

Etched Silicon

Many Steps →

Wire Routing
– Don't Cross the Wires

Programming Levels

Level 7	Application Layer (Prompt Engineering, UI/UX)
	↓ Intent Interpretation (User -> Code Translation)
Level 6	High-Level (Problem/Object Oriented) Programming Languages
	↓ Translation(Compiler)
Level 5	Assembly Language
	↓ Translation(Assembler)
Level 4	Operating System (aka the Machine Level)
	↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))
Level 3	Instruction Set Architecture Level
	↓ Microprogram Interpretation or Direct Execution
Level 2	Micro-architecture Level
	↓ Logic Synthesis
Level 1	Digital Logic / Circuit Design Level
	↓ Physical/Layout Design
Level 0	Layout for Fabrication (Defined by the OASIS Standard)
	↓ Lithography
	Etched Silicon

Many Steps →

Clock Tree Synthesis – Got to Time it Just Right

Programming Levels

Level 7 — Application Layer (Prompt Engineering, UI/UX)

↓ Intent Interpretation (User -> Code Translation)

Level 6 — High-Level (Problem/Object Oriented) Programming Languages

↓ Translation(Compiler)

Level 5 — Assembly Language

↓ Translation(Assembler)

Level 4 — Operating System (aka the Machine Level)

↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 3 — Instruction Set Architecture Level

↓ Microprogram Interpretation or Direct Execution

Level 2 — Micro-architecture Level

↓ Logic Synthesis

Level 1 — Digital Logic / Circuit Design Level

↓ Physical/Layout Design

Level 0 — Layout for Fabrication (Defined by the OASIS Standard)

↓ Lithography

Etched Silicon

Many Steps →

Heat & Capacitance

Programming Levels

Level 7 — Application Layer (Prompt Engineering, UI/UX)

↓ Intent Interpretation (User -> Code Translation)

Level 6 — High-Level (Problem/Object Oriented) Programming Languages

↓ Translation(Compiler)

Level 5 — Assembly Language

↓ Translation(Assembler)

Level 4 — Operating System (aka the Machine Level)

↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 3 — Instruction Set Architecture Level

↓ Microprogram Interpretation or Direct Execution

Level 2 — Micro-architecture Level

↓ Logic Synthesis

Level 1 — Digital Logic / Circuit Design Level

↓ Physical/Layout Design

Level 0 — Layout for Fabrication (Defined by the OASIS Standard)

↓ Lithography

Etched Silicon

ASML →

Checkout EUV Lithography

Programming Levels

Level 7 — Application Layer (Prompt Engineering, UI/UX)

↓ Intent Interpretation (User -> Code Translation)

Level 6 — High-Level (Problem/Object Oriented) Programming Languages

↓ Translation(Compiler)

Level 5 — Assembly Language

↓ Translation(Assembler)

Level 4 — Operating System (aka the Machine Level)

↓ Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 3 — Instruction Set Architecture Level

↓ Microprogram Interpretation or Direct Execution

Level 2 — Micro-architecture Level

↓ Logic Synthesis

Level 1 — Digital Logic / Circuit Design Level

↓ Physical/Layout Design

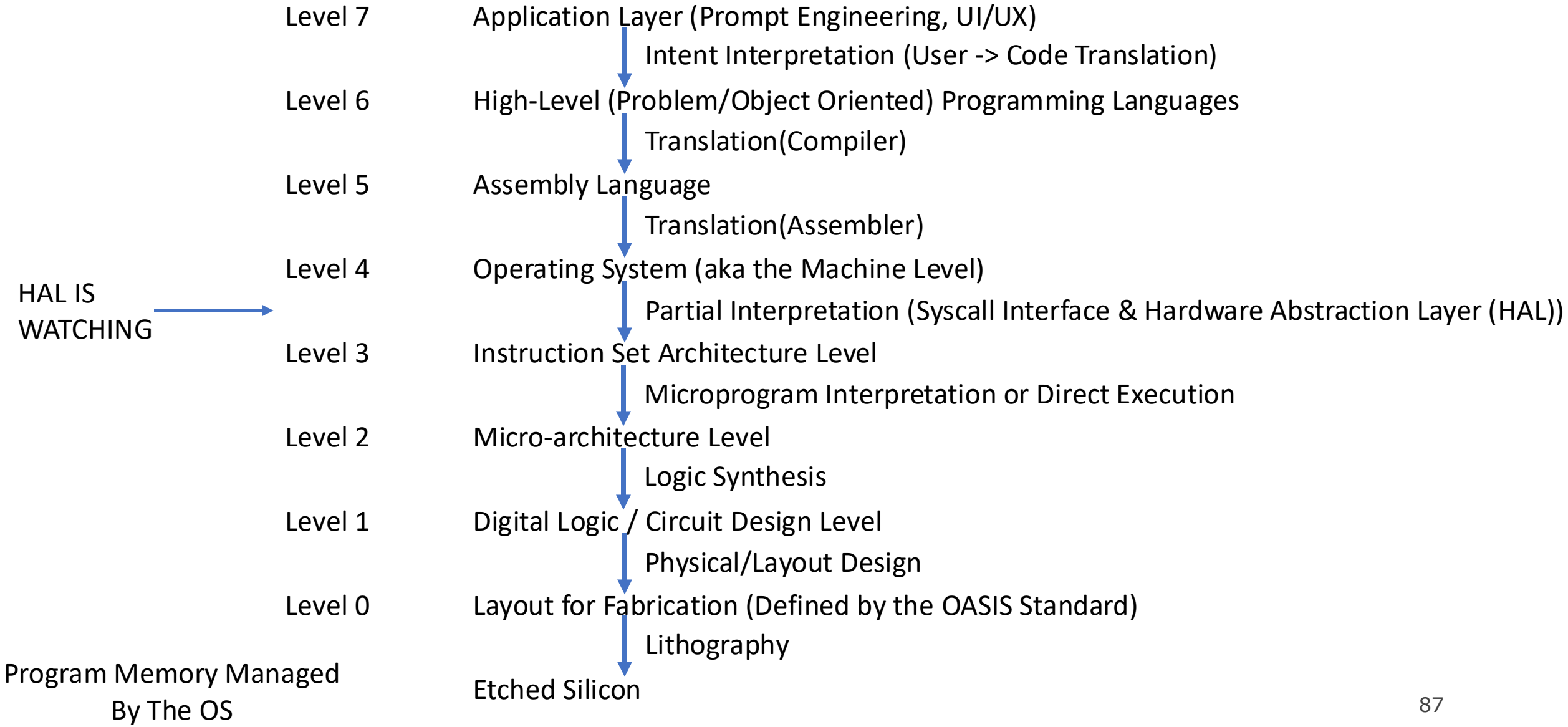
Level 0 — Layout for Fabrication (Defined by the OASIS Standard)

↓ Lithography

Etched Silicon

Which layer throws a segfault?

Programming Levels

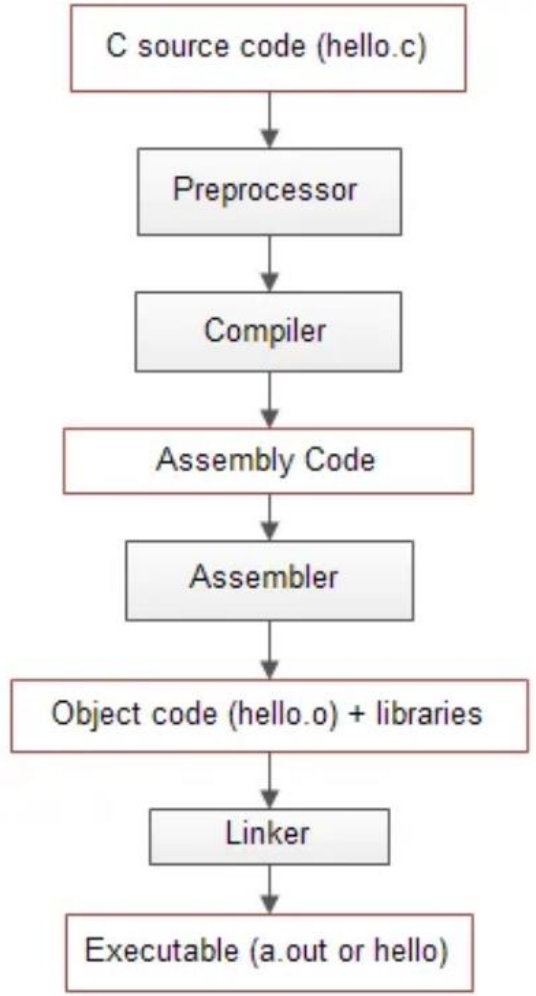


More on the Compiler

How Does GCC Work?

- One Unix Command – A lot of steps!

gcc hello.c -o hello

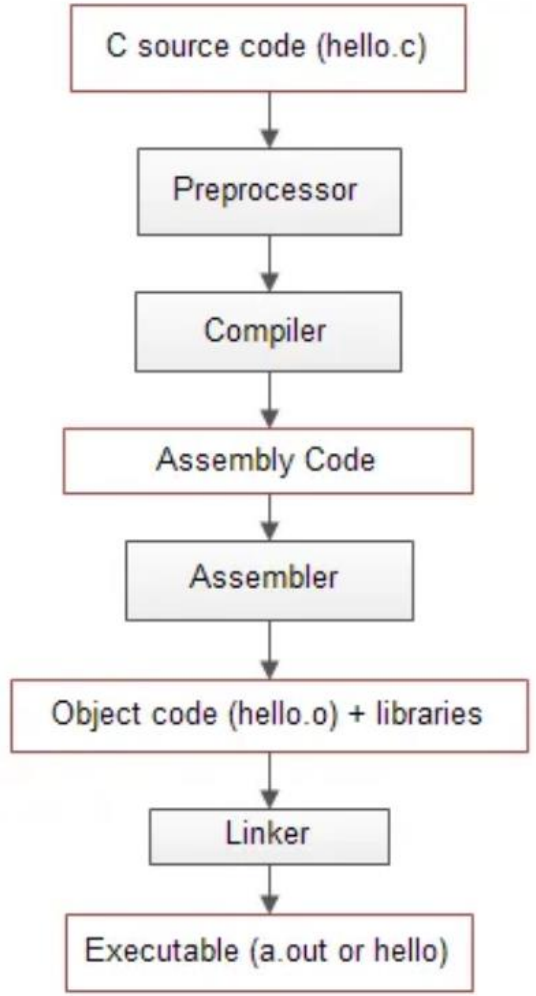


https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

How Does GCC Work?

- Preprocessing – Handle Programmer Conveniences
  - #Macros convert to normal C code
  - Lines split by \ are joined
  - Comments are removed
    - NOTE: Some comments are added, but our comments are removed
  - Bring in functions and variables from the headers
    - This is how the #include is resolved

gcc -E hello.c > pre_processed_hello



https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

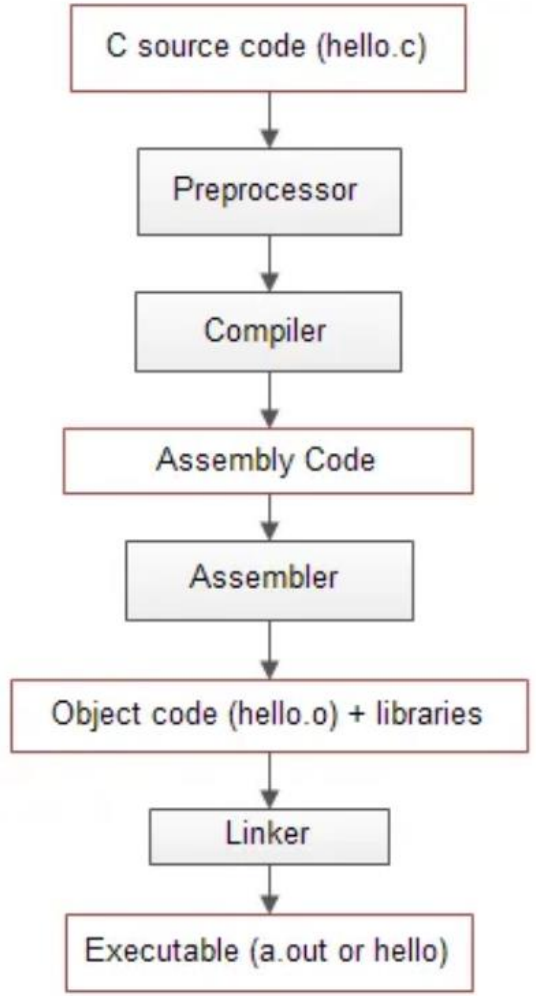
How Does GCC Work?

- Compilation – C to Assembly

gcc -S hello.c

- Will generate intermediate ‘human-readable’ assembly

- There are different styles/syntax for x86, we use AT&T
  - AT&T is also the gcc default



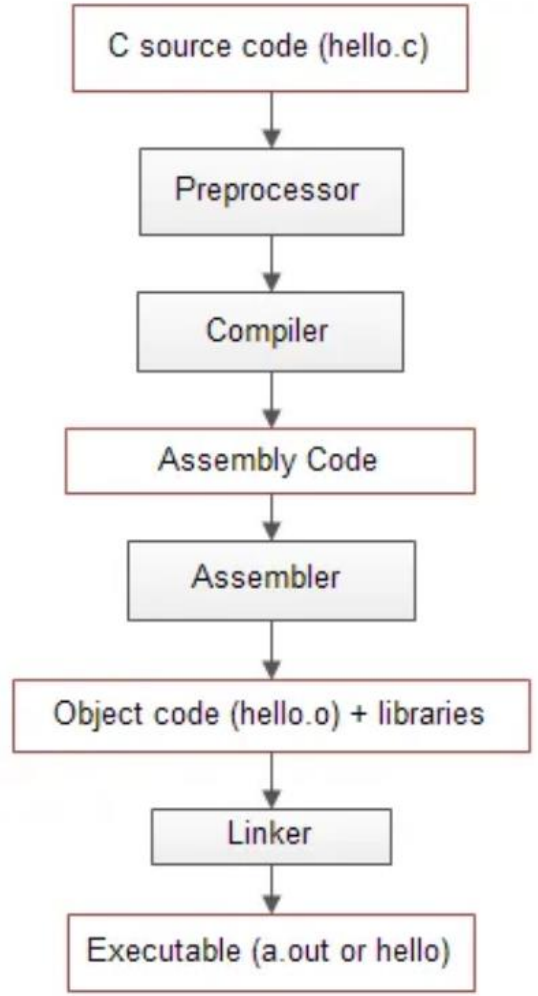
https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

How Does GCC Work?

- Object Generation – C to Object File

gcc -c hello.c

- “Just compile; Don't link"
- This outputs a non-human readable Object File
  - It is defined as a type of incomplete machine code
  - With extra metadata to power linking
- Using objdump –d hello.o , we can see the assembly



https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

How Does GCC Work?

- Linking – Bringing All the pieces together
  - Object Files & Libraries -> Fully Executable Machine Code

gcc hello.o -o hello

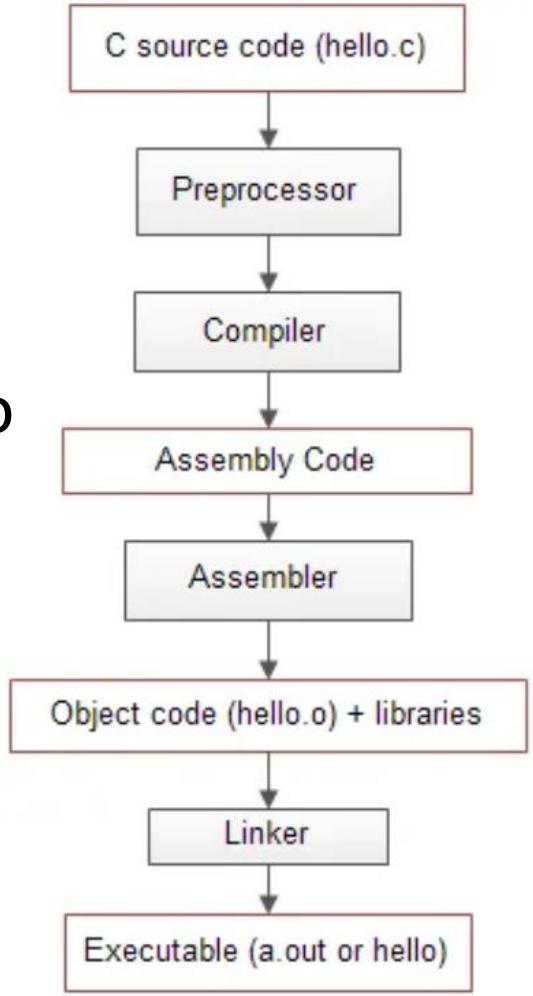
ld -o hello hello.o -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o

- NOTE: We can get our .o in more than one-way

gcc -c hello.c

OR

as hello.s



https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

What does the Assembler Do?

A Two Step Process

- Pass 1: Setup Memory Addresses
  - The program reads in the assembly program identifying and tracking:
    - Labels
    - Literals
    - Data Variables

- Pass 2: Generate the Machine Code (Byte/Binary Code)
  - Identify Opcode from the mnemonic assembly
  - Resolve labels/literals/variables using the tables from Step 1
  - Convert Data to Binary
  - Identifies External (Out of Program) References and places markers for the Linker
  - Setup Metadata for linking if this program has loadable parts

Final Output is not runnable, but has all the parts need if linking can complete

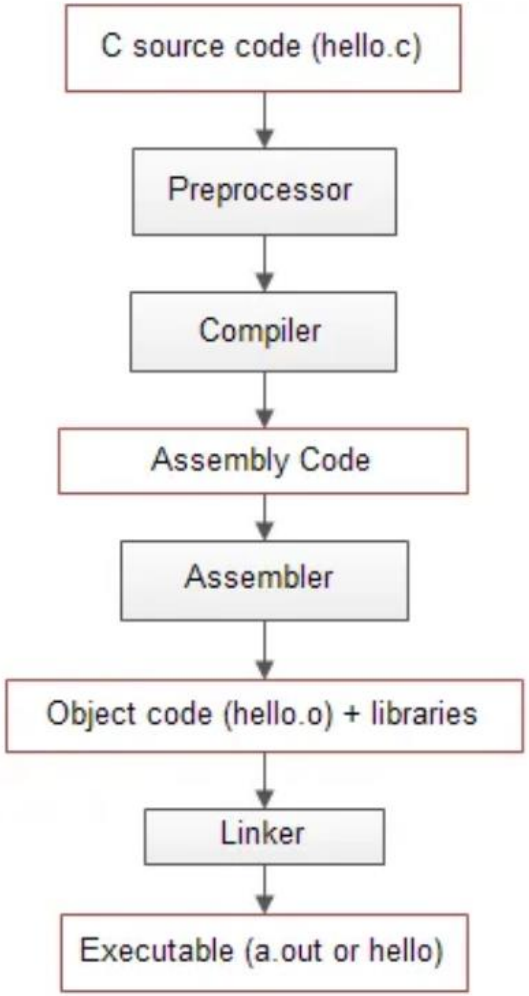
Why do we need a linker?

Many Links

- Every C file corresponds to a .o
- Libraries can also be made into linkable formats
- We don’t want to have to write all our code in 1 file and we want to use the STL
- The linker makes this all possible

How Does GCC Work?

- Multi-Step Process -> Multiple Failure Points
- Compilation can fail for many reasons at different points
- Mainly two areas that fail ‘Compilation’ or Linking
- If compilation succeeds, Intermediate Assembly will be good!

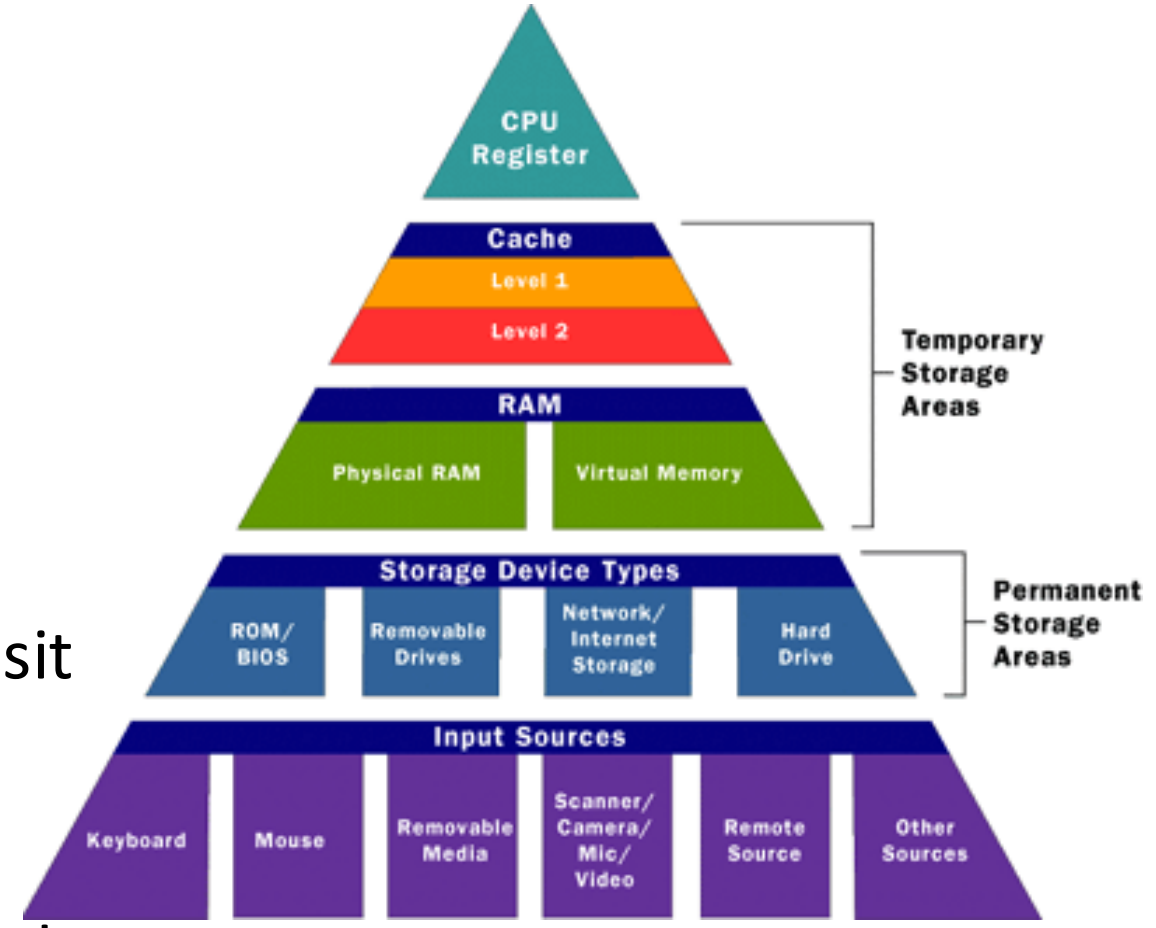


https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

Peeking at Memory

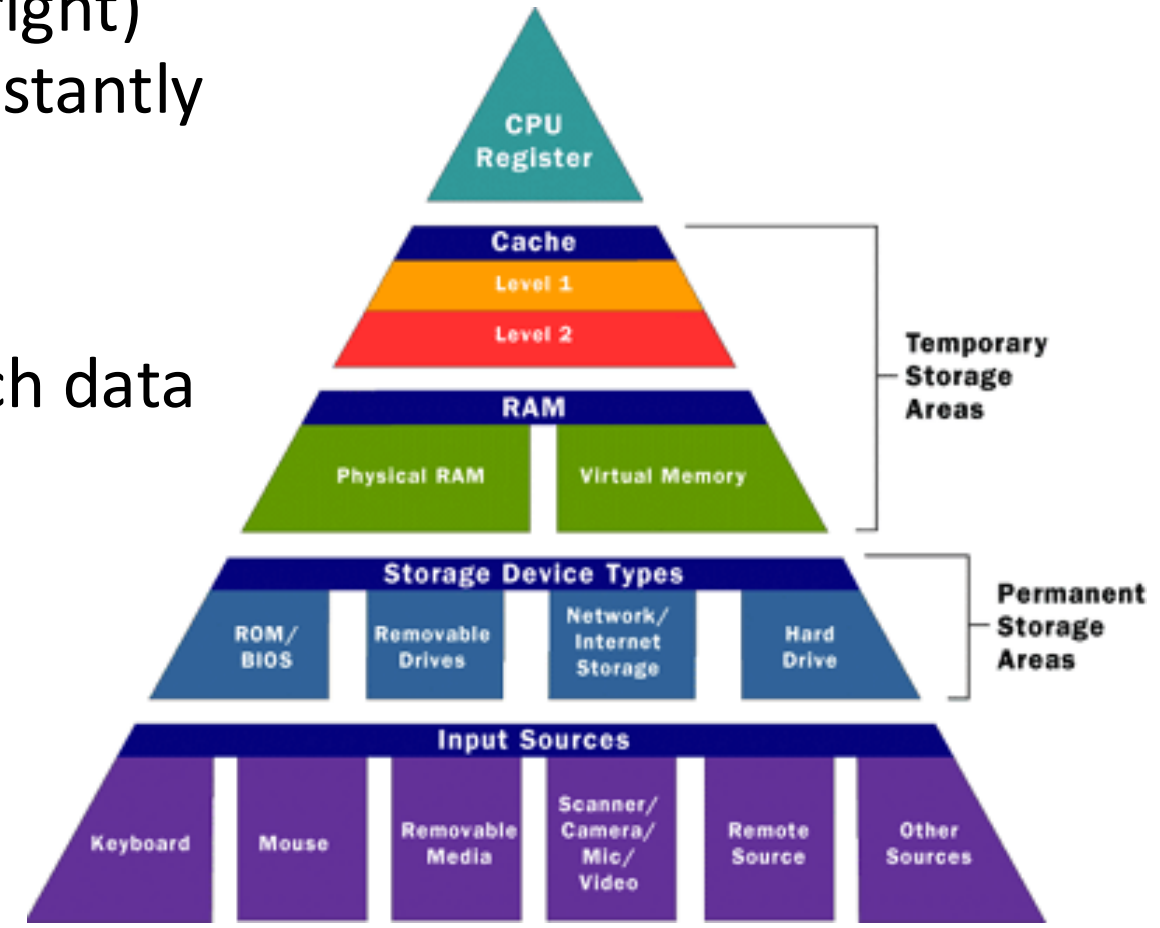
Speed vs Space

- CPU is the most important place
  - Closer to CPU, less travel time
  - But limited space, so bottleneck getting there
- Think of the CPU like downtown, generally expensive and highly desirable real estate
- The BUS (actual technical name) is our transit system around the computer
- Places close to the CPU are more limited and more valuable, since they can get to the CPU faster



Speed vs Space

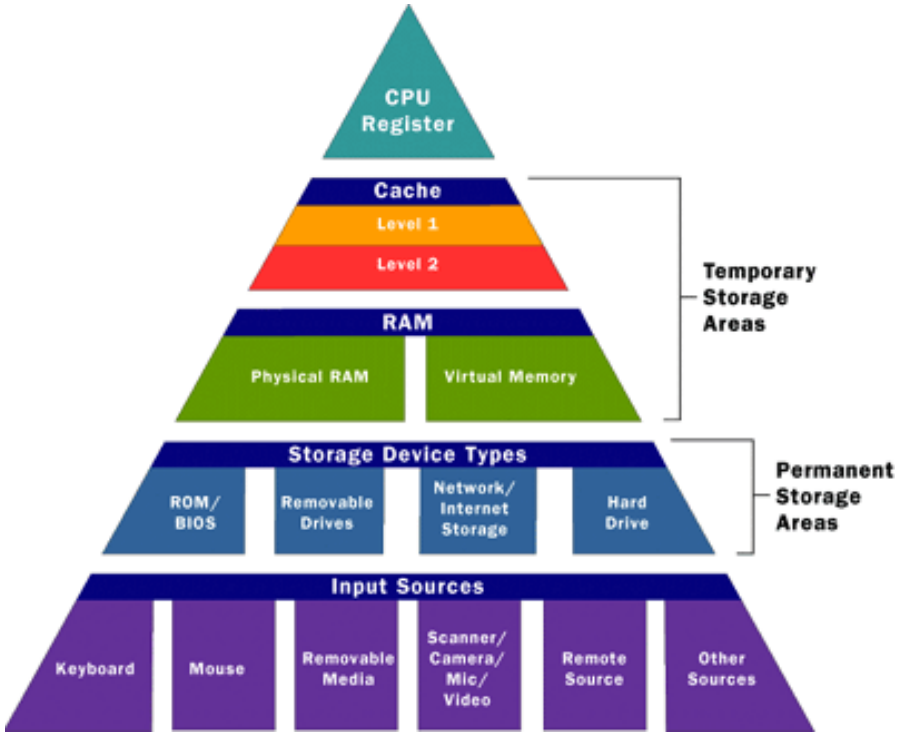
- All of Memory (Temporary Storage on the right) and the registers is rent only, so data is constantly moving around
- Many algorithms developed to decide which data gets to live where and for how long
- Proper access makes a huge difference on performance



Speed vs Space

- Approximate Access Times

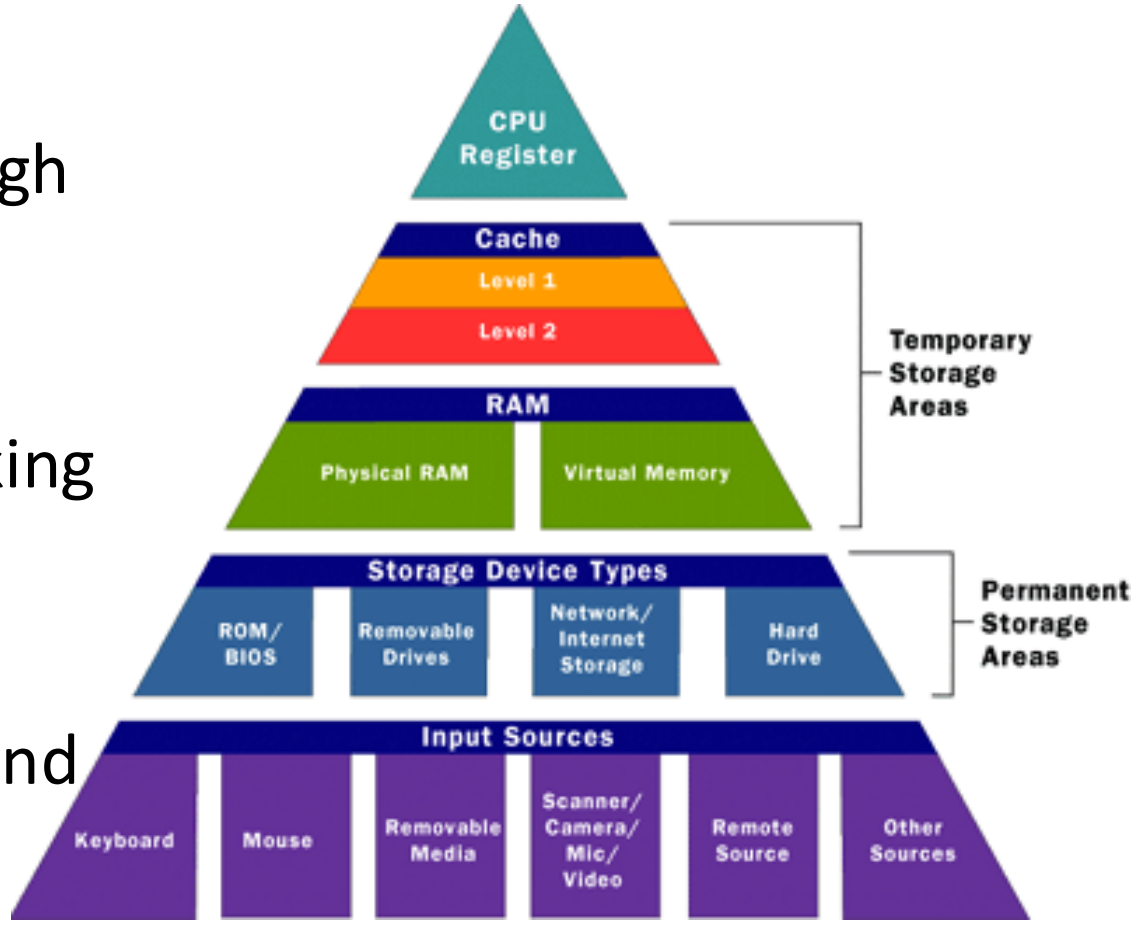
Resource	Latency Time
Register	0 Cycles (already here)
Level 1 Cache	~0.5 ns
Level 2 Cache	~7 ns (14x L1)
RAM	~100 ns (20x L2, 200x L1)
SSD	~100-150 us (~14Kx L2, 200Kx L1)
Hard (Spinning) Disk	~10 ms (~2.8Mx L2, 40Mx L1)
Network Packet CA -> Netherlands -> CA	~150 ms (~21Mx L2, 300Mx L1)
Average Human Response Time to Visual Stimulus	~200 ms (~28Mx L2, 400Mx L1)



For more on speed checkout:
https://www.cs.princeton.edu/courses/archive/spring20/cos217/lectures/20_Mem_Storage_Hierarchy.pdf
https://gist.github.com/jboner/2841832

Speed vs Space

- Pre-emptive requests and moving of data is critical
- Orders of Magnitude Improvements from high locality
- Every part of the pyramid is working on making this faster
- Better BUS, faster storage(both temporary and permanent), bigger RAM, better algorithms

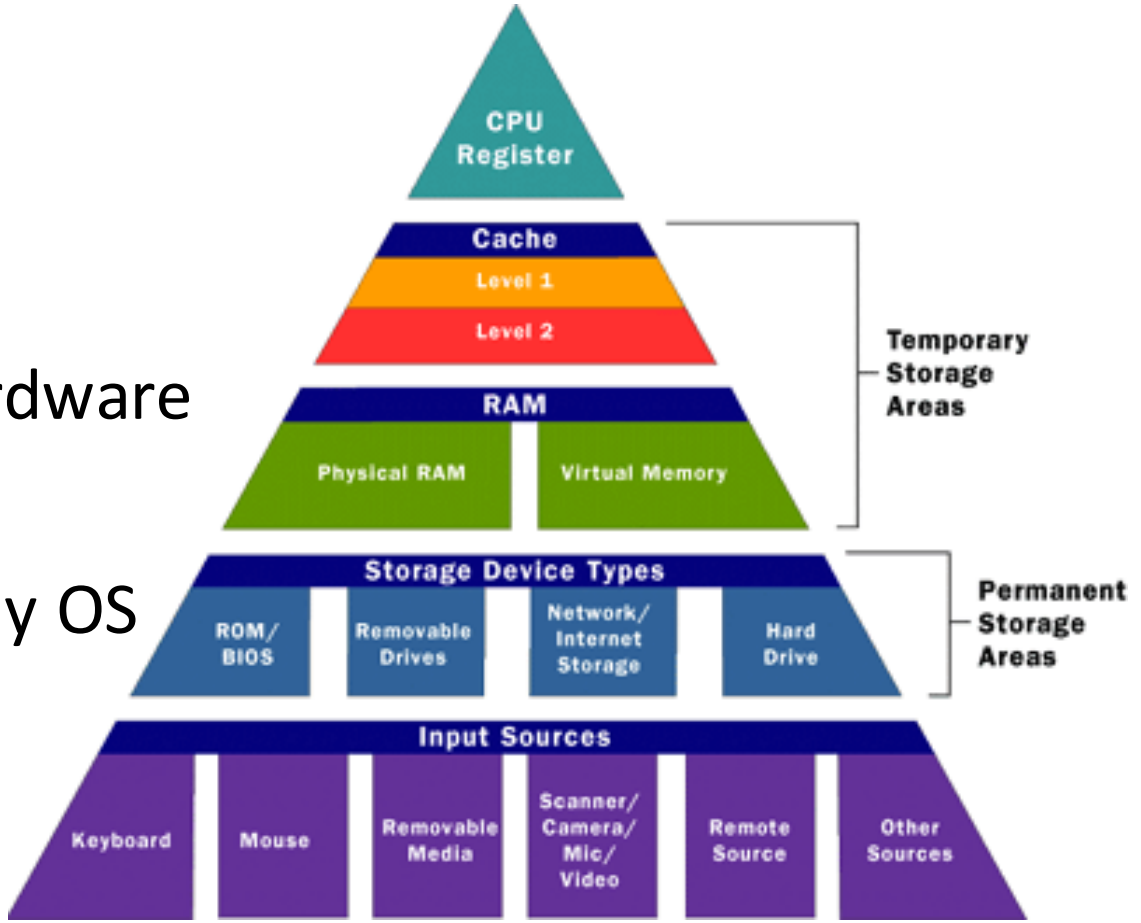


What is Locality?

- Temporal Locality
  - Has the data been used recently? Then we expect to be used again soon
- Spatial Locality
  - The data appears close together in the program/memory, so it will likely be needed at the same time.
- Hardware and OS designers consider algorithms to predict and leverage locality to optimize management of memory resources
- Cache in particular is a limited resource and must be used effectively to leverage benefits

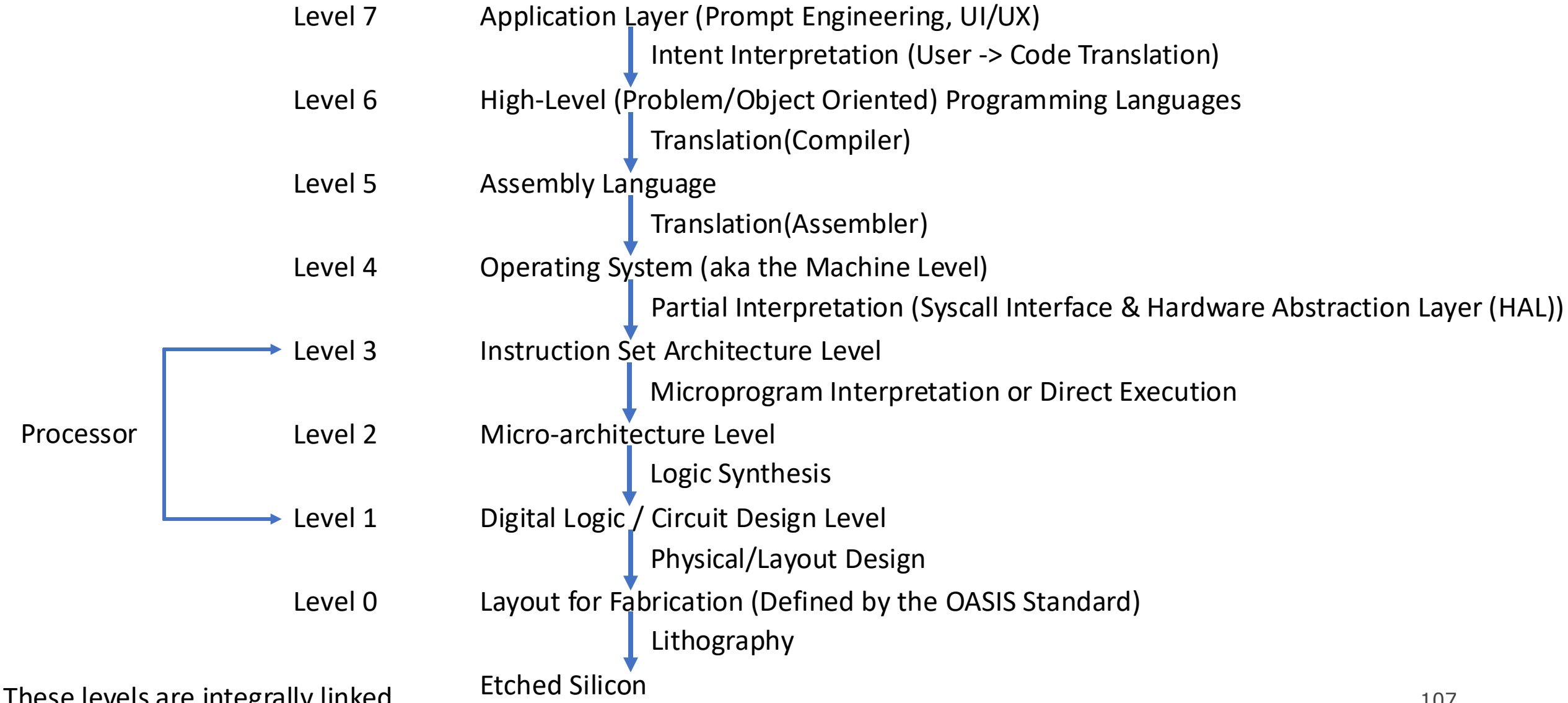
Who Gets to Manage the Memory?

- Registers – Managed by the Compiler/Assembler
- Cache – Managed by Hardware Designers
- Memory – Mainly the OS, influenced by hardware
- Disk – Managed by the user and occasionally OS



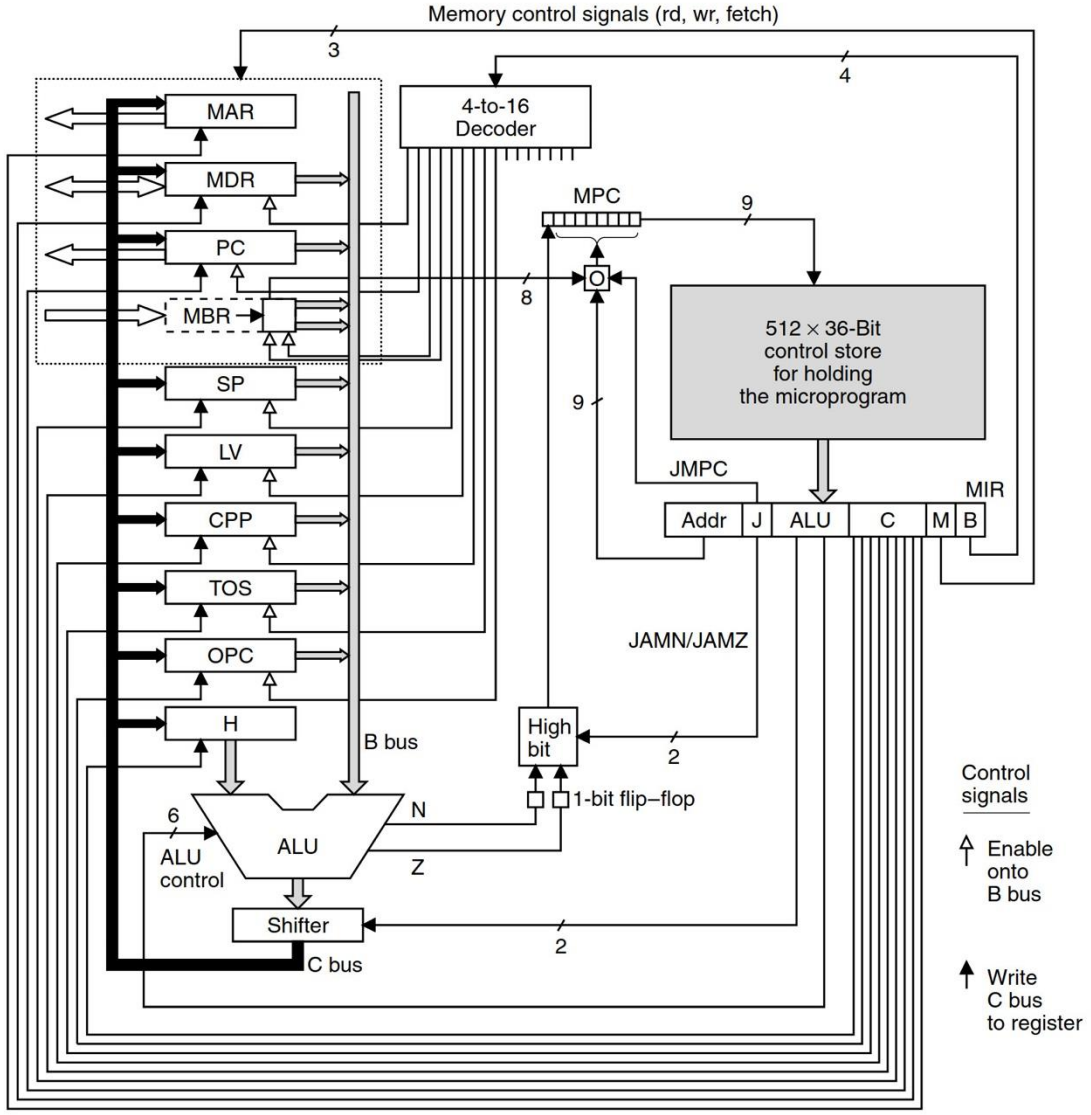
Architecture & The ISA

Programming Levels



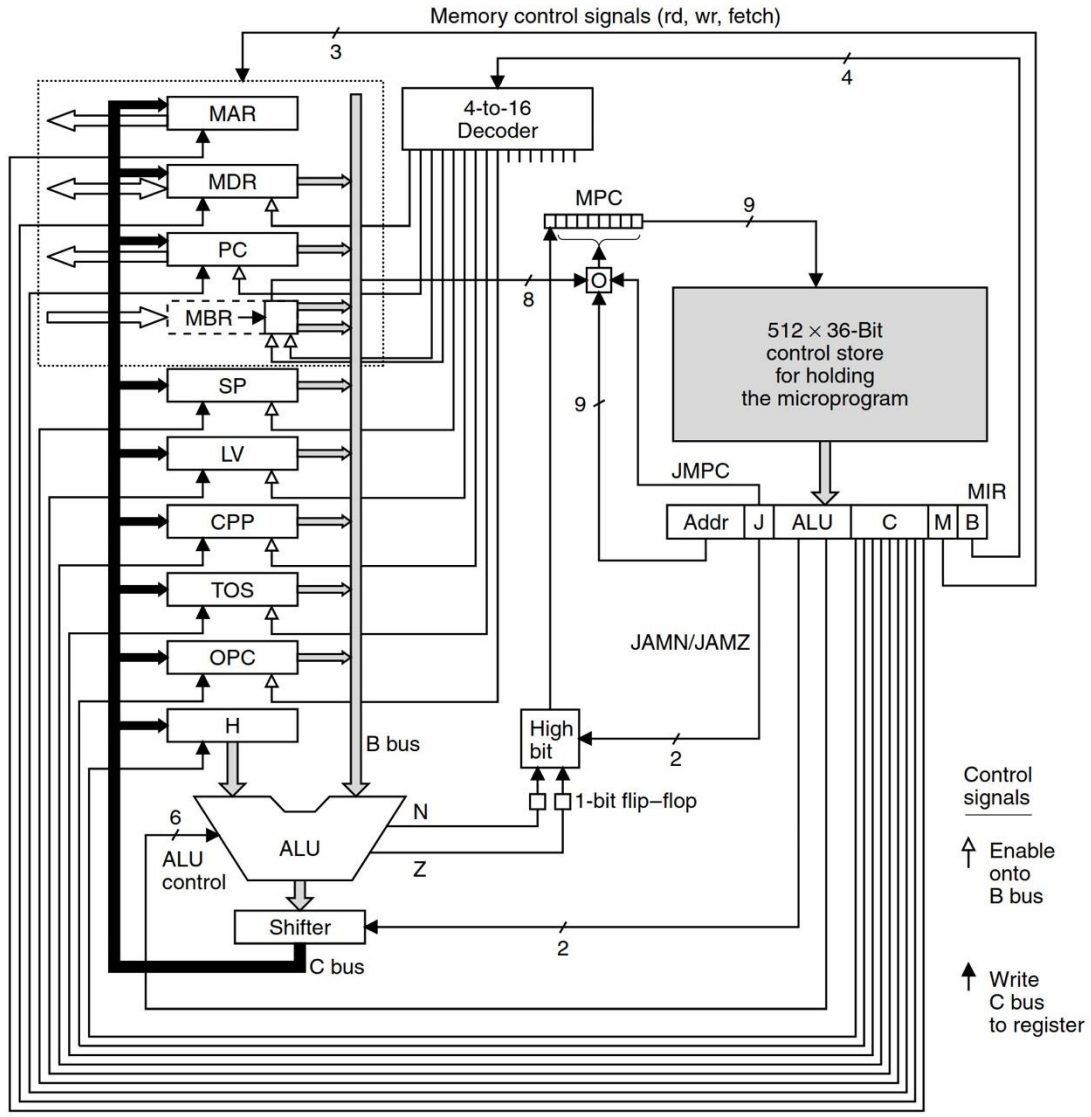
A ‘Simple’ Example

- MIC-1 Architecture (Tanenbaum - Structured Computer Organization 6th Edition)
- IJVM ISA – Subset of the Java Virtual Machine
- A ‘Vanilla’ processor design



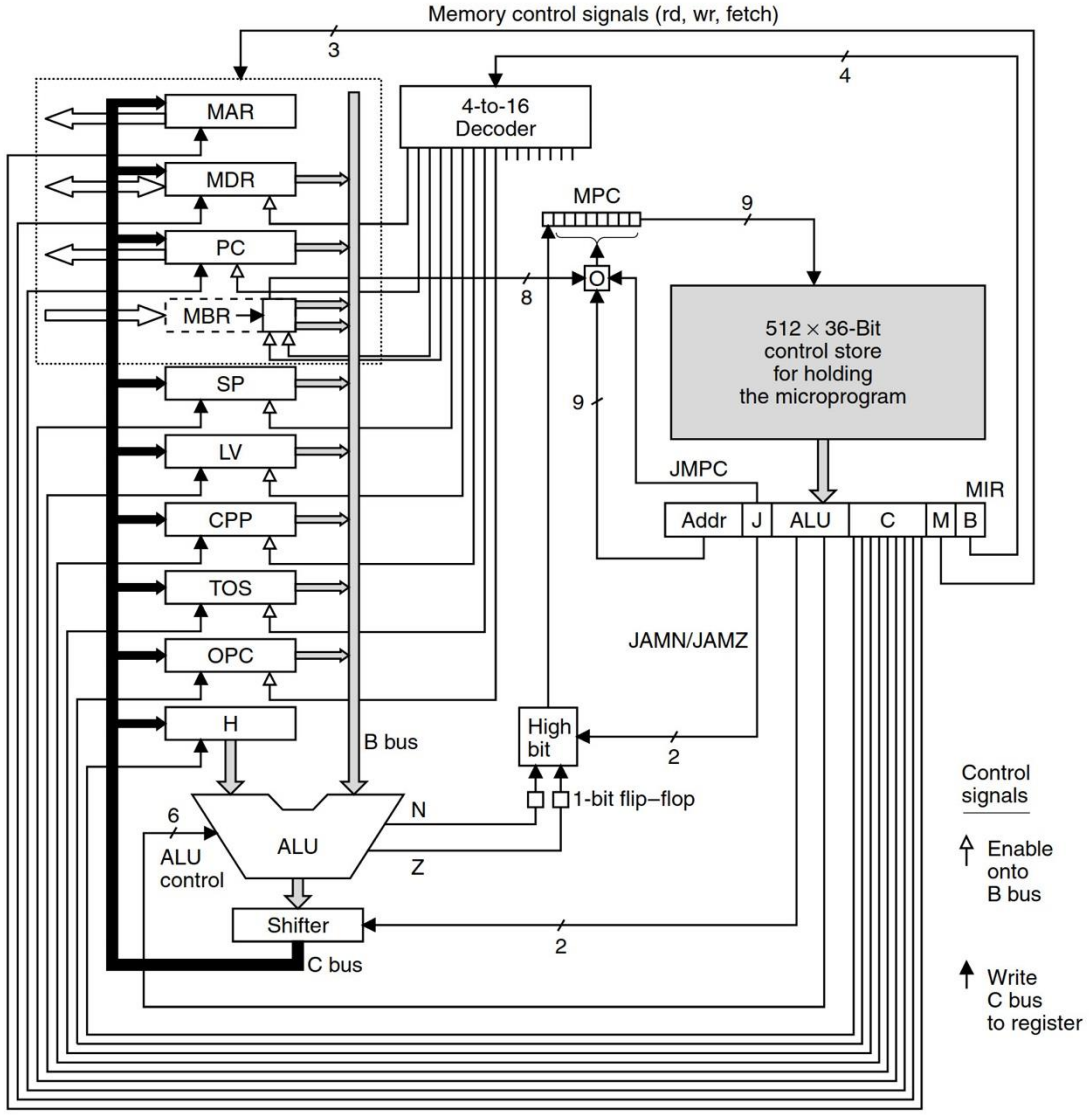
A ‘Simple’ Example

- Control Store is the most important part!
- Our ISA is defined by that unit
- 9 wires in -> 2**9 possible combinations, 2**9 (512) possible commands
- Each command drives 36 wires to control the chip
- Assembly/Machine Language is defined by the hardware



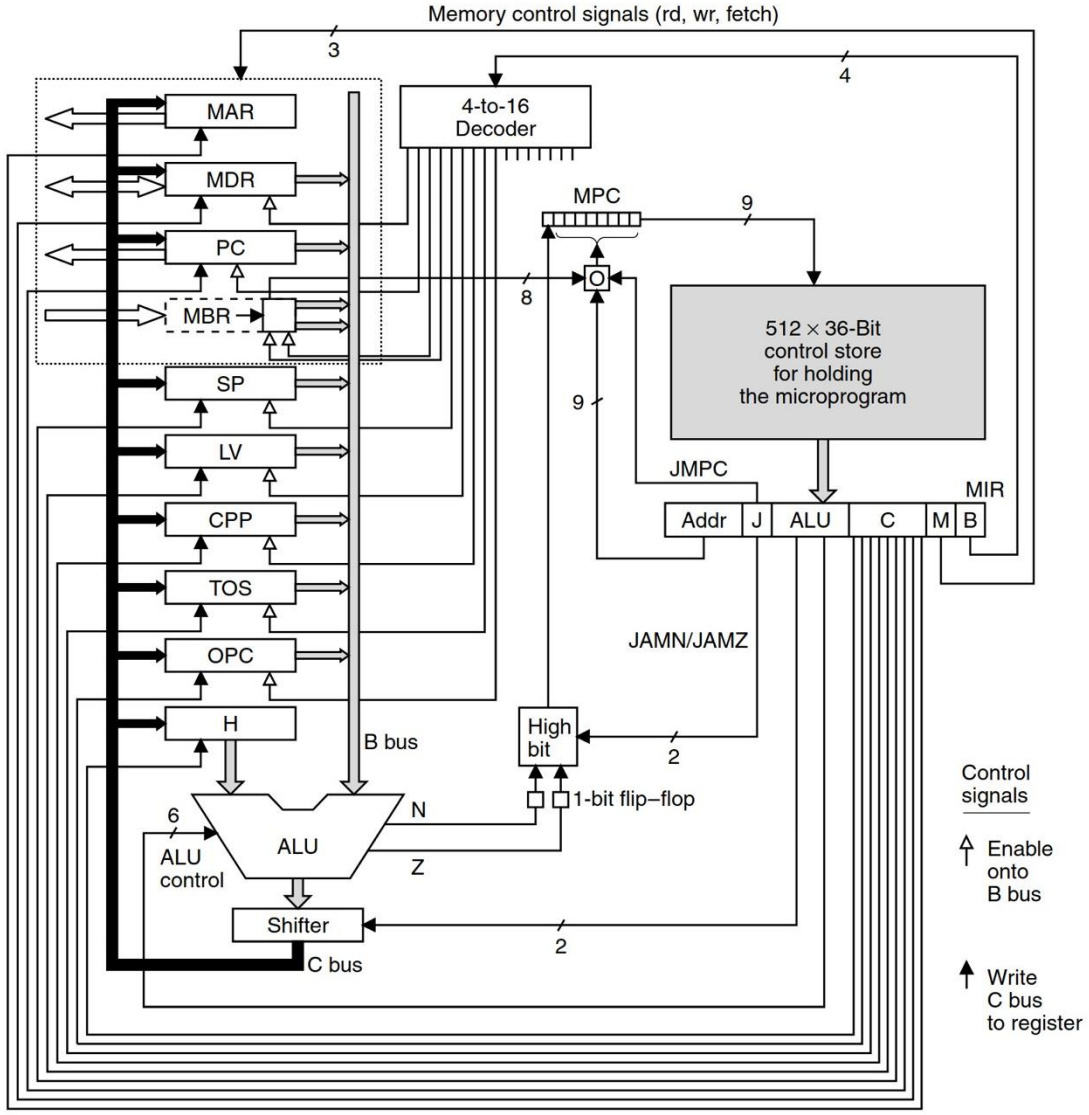
A ‘Simple’ Example

- ALU – Arithmetic & Logic Unit
  - Performs Math & Logic Operations
- MAR – H are the registers
- B + Decoder – Enables Register to load onto B Bus
- Z and N act similar to our condition codes, but in a much more limited/simple way
- C controls the C Bus, informing the destination register to receive its value



A ‘Simple’ Example

- Notice how the ALU is only able to take in the left operand from the H register
- All two operand ALU operations, would need to first load the left operand to H
- This would be an example of a hardware based constraint



Better Design Better Performance

- The MIC-2 Fixes this issue by adding another BUS improving the Datapath
- Design directly impacts the ISA that we can make available