CS107, Lecture 14

Alignment, Optimization, & Basic Architecture

Based on slides created by Nick Troccoli, Chris Gregg, and Raymond Klefstad

Attendance

https://forms.gle/mUWfemVpi1R81VyW6

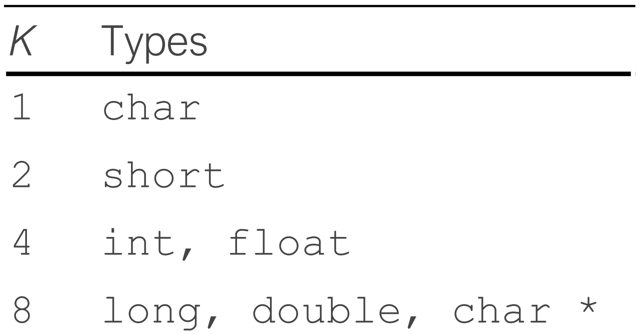
Registers Vs Addresses

- So far, we’ve often seen local variables stored directly in registers, rather than on the stack.

- There are **three** common reasons that local data must be in memory:
  - We’ve run out of registers
  - The ‘&’ operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

Data Alignment

- Computer systems often put restrictions on the allowable addresses for primitive data types, requiring that the address for some objects must be a multiple of some value *K* (normally 2, 4, or 8).
- These ***alignment restrictions*** simplify the design of the hardware.
- For example, suppose that a processor always fetches 8 bytes from the memory system, and an address must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address as a multiple of 8, then we can read or write the values with a single memory access.
- For x86-64, Intel recommends the following alignments for best performance:



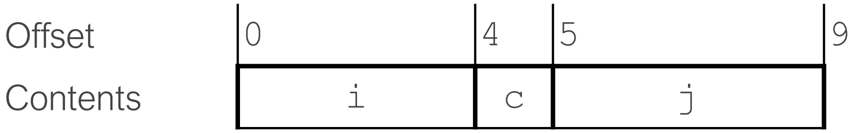
K	Types
1	`char`
2	`short`
4	`int, float`
8	`long, double, char *`

Data Alignment

- The compiler enforces alignment by making sure that every data type is organized in such a way that every field within the struct satisfies the alignment restrictions.
- For example, let's look at the following struct:

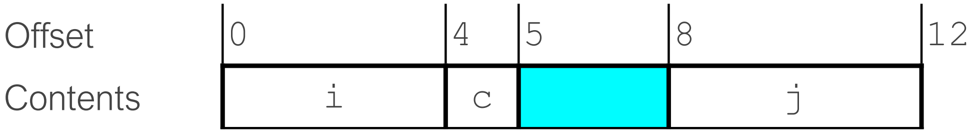
```
struct S1 {
    int i;
    char c;
    int j;
};
```

- If the compiler used a minimal allocation:



Offset	0	4	5	9
Contents	i	c	j	

- This would make it impossible to align fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap between fields `c` and `j`:



Offset	0	4	5	8	12
Contents	i	c		j	

- So, don't be surprised if your structs have a `sizeof()` that is larger than you expect!

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

Optimizations you'll see

`nop`

- **`nop/nopl`** are "no-op" instructions – they do nothing!
- Intent: Make functions align on address boundaries that are nice multiples of 8.
- "Sometimes, doing nothing is how to be most productive" – Philosopher Nick

`mov %ebx,%ebx`

- Zeros out the top 32 register bits (because a `mov` on an e-register zeros out rest of 64 bits).

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.

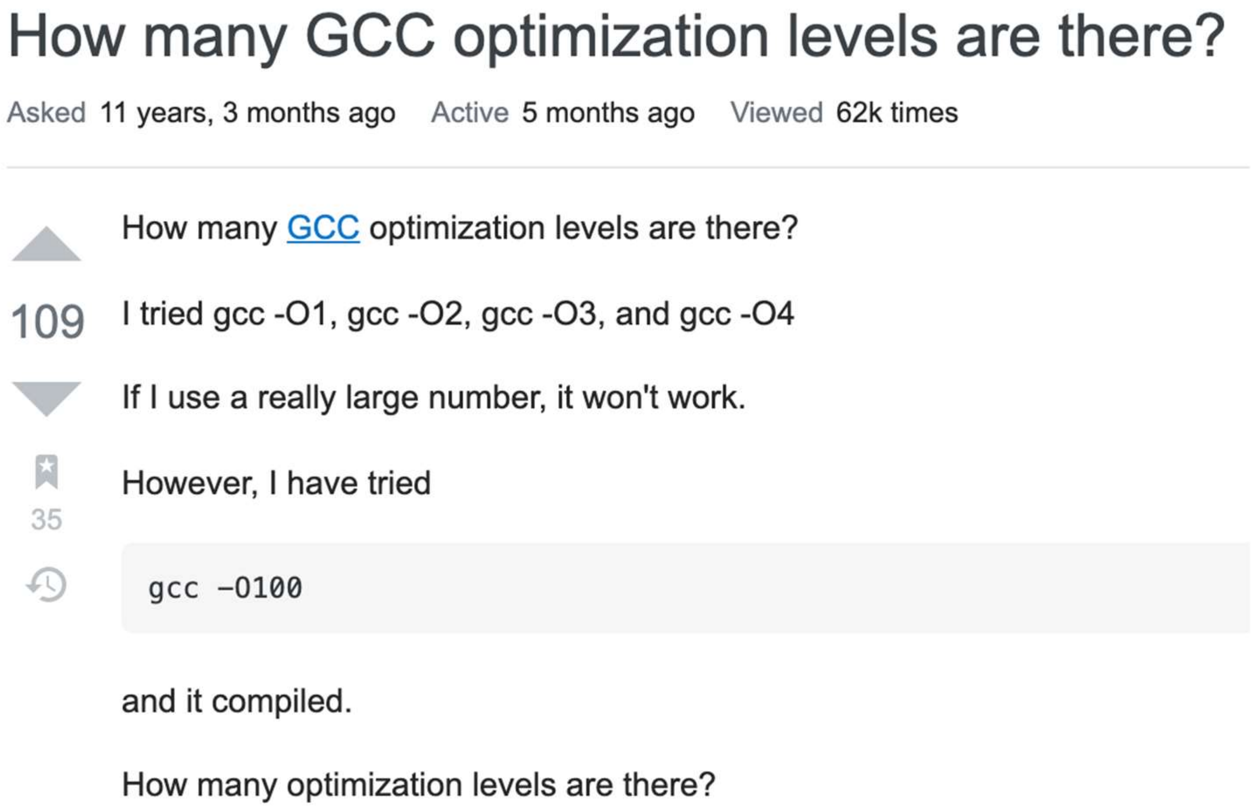
GCC Optimization

- 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

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
- Loop Unrolling

Constant Folding

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

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

Constant Folding

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

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

```
int seconds = 86400 * n_days;
```

Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = 1;
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

Constant Folding

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * sizeof(arr);
    int c = 1;
    return a * param + (a + 0x15 / c + strlen("Hello") * b - 0x37) / 4;
}
```

```
int fold(int param) {
    char arr[5];
    int a = 0x107;
    int b = a * 5;
    int c = 1;
    return a * param + (a + 0x15 / c + 5 * b - 0x37) / 4;
}
```

Constant Folding

```
int fold(int param) {
    int a = 0x107;
    int b = a * 5;
    int c = 1;
    return a * param + (a + 0x15 / c + 5 * b - 0x37) / 4;
}

int fold(int param) {
    int b = 0x107 * 5;
    int c = 1;
    return 0x107*param+(0x107+0x15/c+5*b-0x37) / 4;
}
```

Constant Folding

```
int fold(int param) {
    int b = 0x107 * 5;
    int c = 1;
    return 0x107*param+(0x107+0x15/c+5*b-0x37) / 4;
}

int fold(int param) {
    return 0x107*param+(0x11c/1+5* 0x107 * 5 -0x37) / 4;
}
```

Constant Folding

```
int fold(int param) {
    int b = 0x107 * 5;
    int c = 1;
    return 0x107*param+(0x107+0x15/c+5*b-0x37) / 4;
}

int fold(int param) {
    return 0x107*param+(0x107 + 0x15/1+5* 0x107 * 5 -0x37) / 4;
}

int fold(int param) {
    return 0x107 * param + 1701;
}
```

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)

```
00000000000011b0 <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
- 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
```

```
00000000000011b0 <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?

1) The compiler may not always be able to optimize every instance.
2) Helps reduce redundancy!
3) Makes code more readable!

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- **Dead Code**
- Strength Reduction
- Code Motion
- 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)

```
0000000000011a9 <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
- 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 << 5;`

- `int b = a * 7;`

  Becomes:

- `int b = a + (a << 2) + (a << 1); or // (a << 3) - a`

- `int c = b / 2;`

  Becomes

- `int c = b >> 1 // Division by odd numbers is more complex`

GCC Optimizations

- Constant Folding
- Common Sub-expression Elimination
- Dead Code
- Strength Reduction
- **Code Motion**
- 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

GCC Optimizations

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

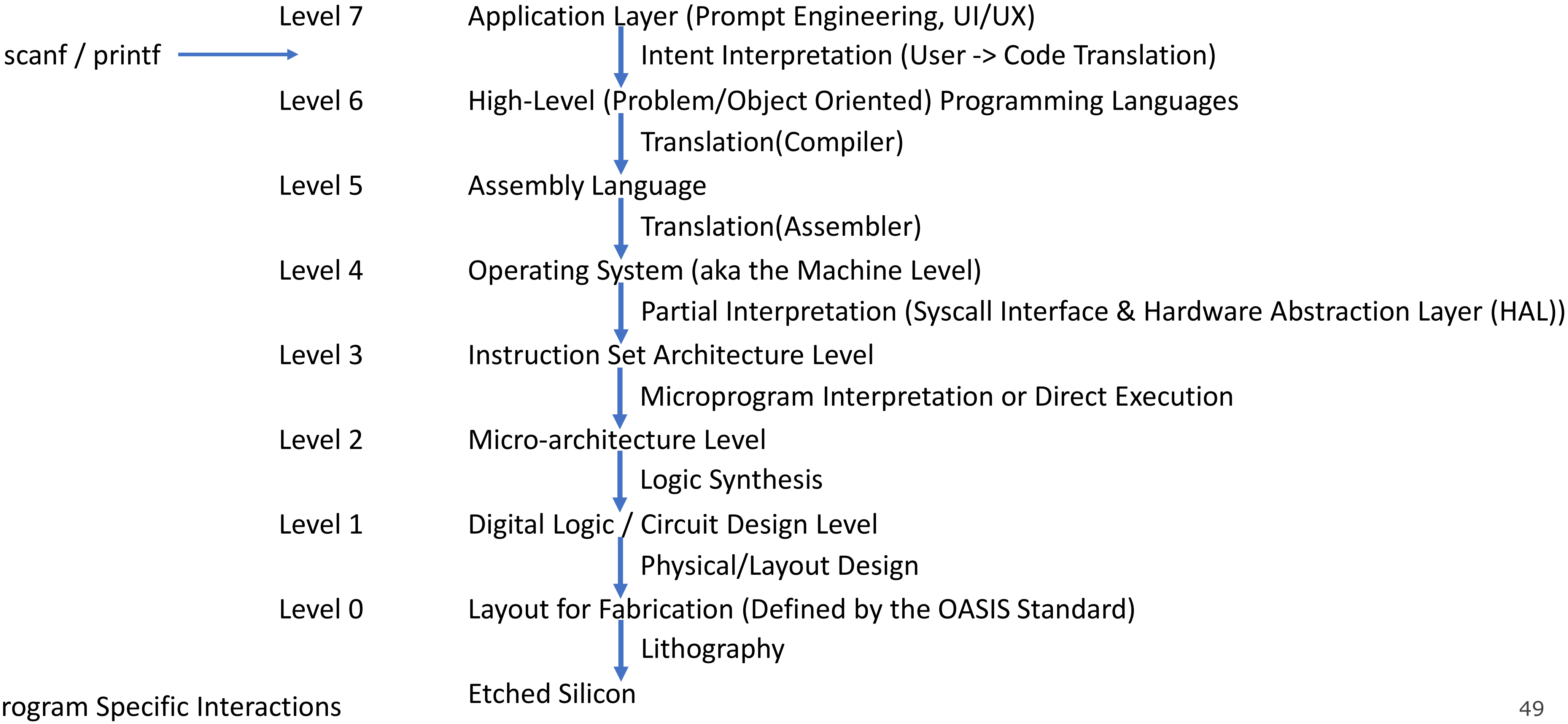
Loop Unrolling

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
```

Into the Architecture!

Programming Levels



scanf / printf

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

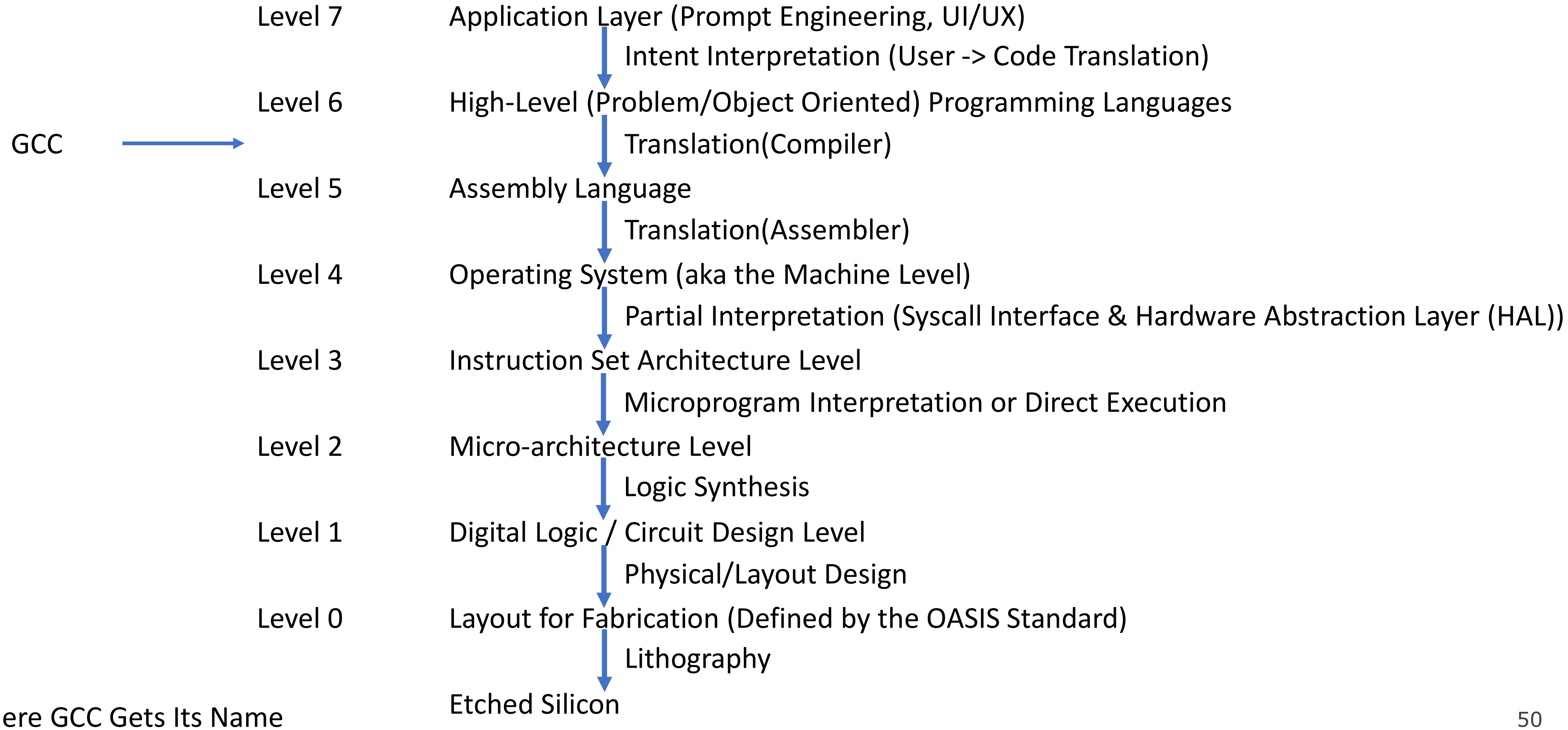
Physical/Layout Design

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

Lithography

Etched Silicon

Programming Levels



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

Intent Interpretation (User -> Code Translation)

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

GCC

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

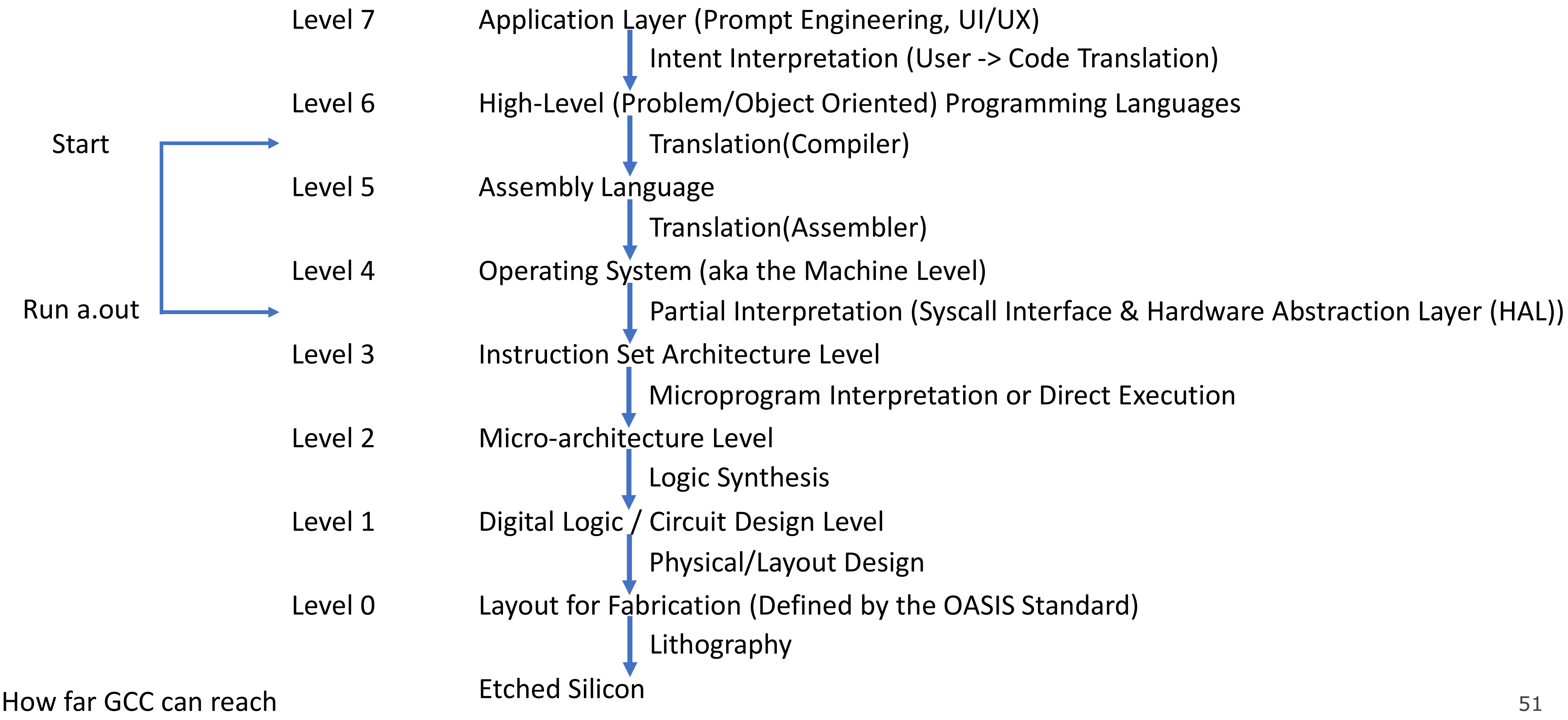
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

Where GCC Gets Its Name

Programming Levels



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

Intent Interpretation (User -> Code Translation)

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

Start

Translation(Compiler)

Level 5 Assembly Language

Translation(Assembler)

Level 4 Operating System (aka the Machine Level)

Run a.out

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

How far GCC can reach

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

AS/GAS → 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

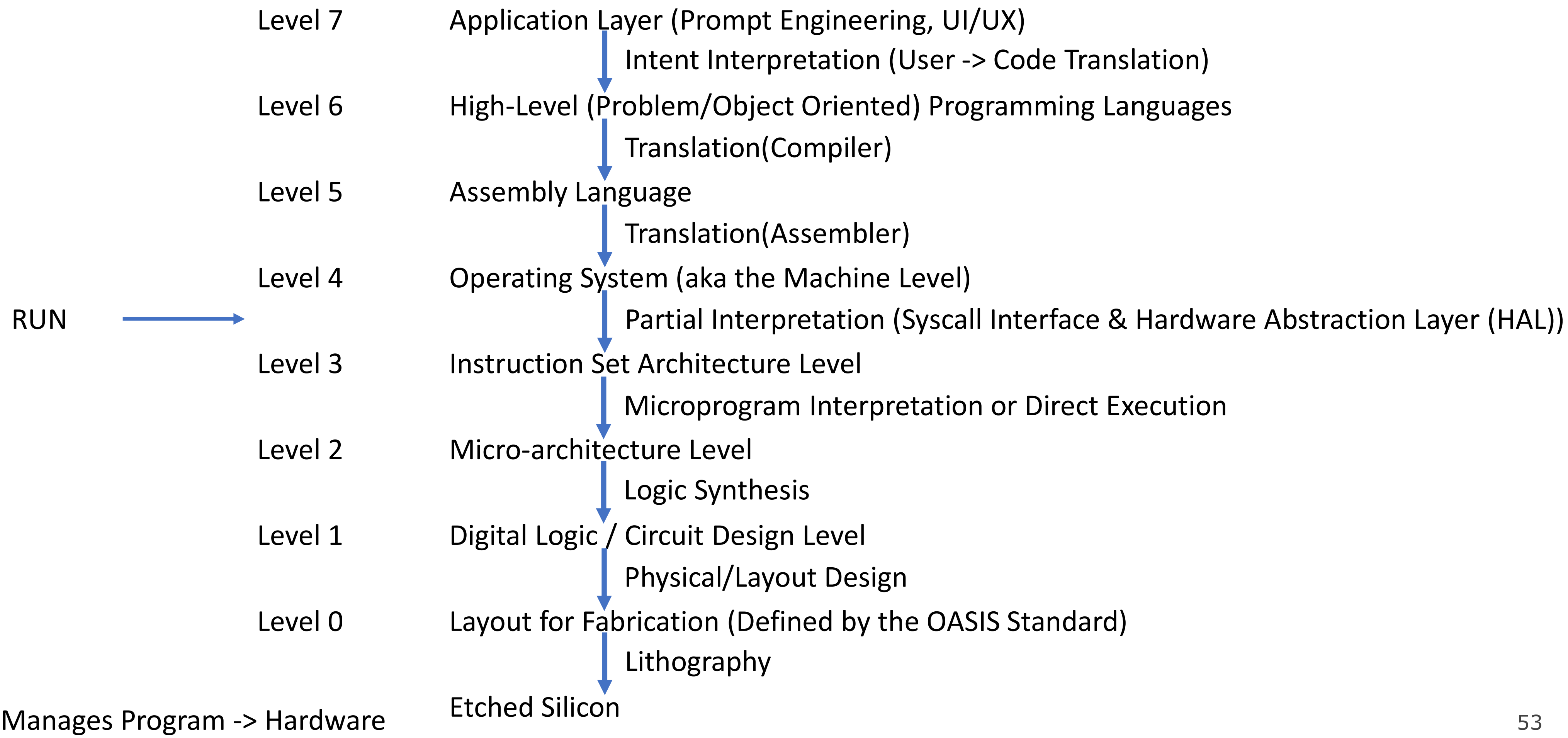
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

GNU Assembler (Inside GCC)

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)

RUN

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

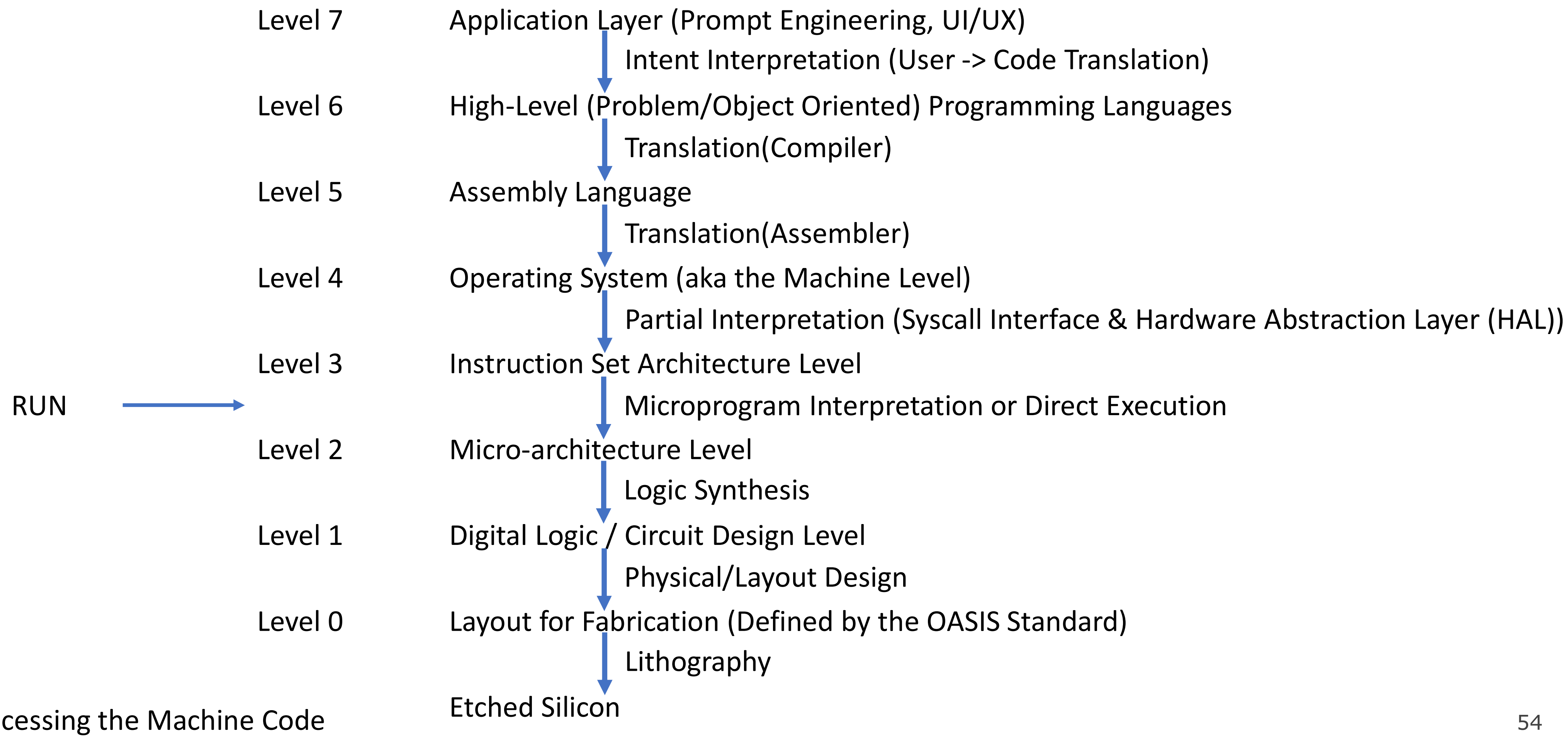
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

OS Manages Program -> Hardware

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

RUN

Microprogram Interpretation or Direct Execution

Level 2 Micro-architecture Level

Logic Synthesis

Level 1 Digital Logic / Circuit Design Level

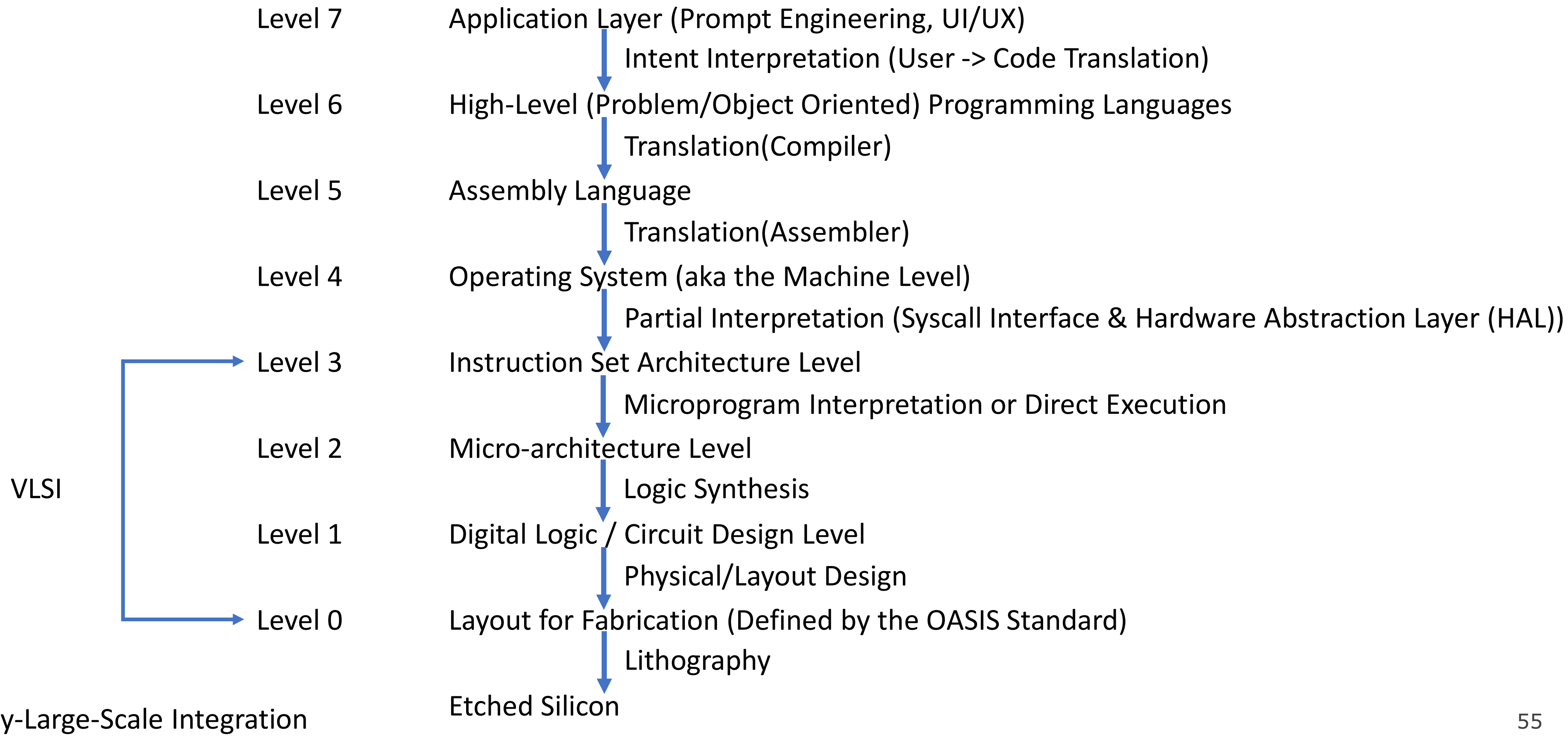
Physical/Layout Design

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

Lithography

Etched Silicon

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)

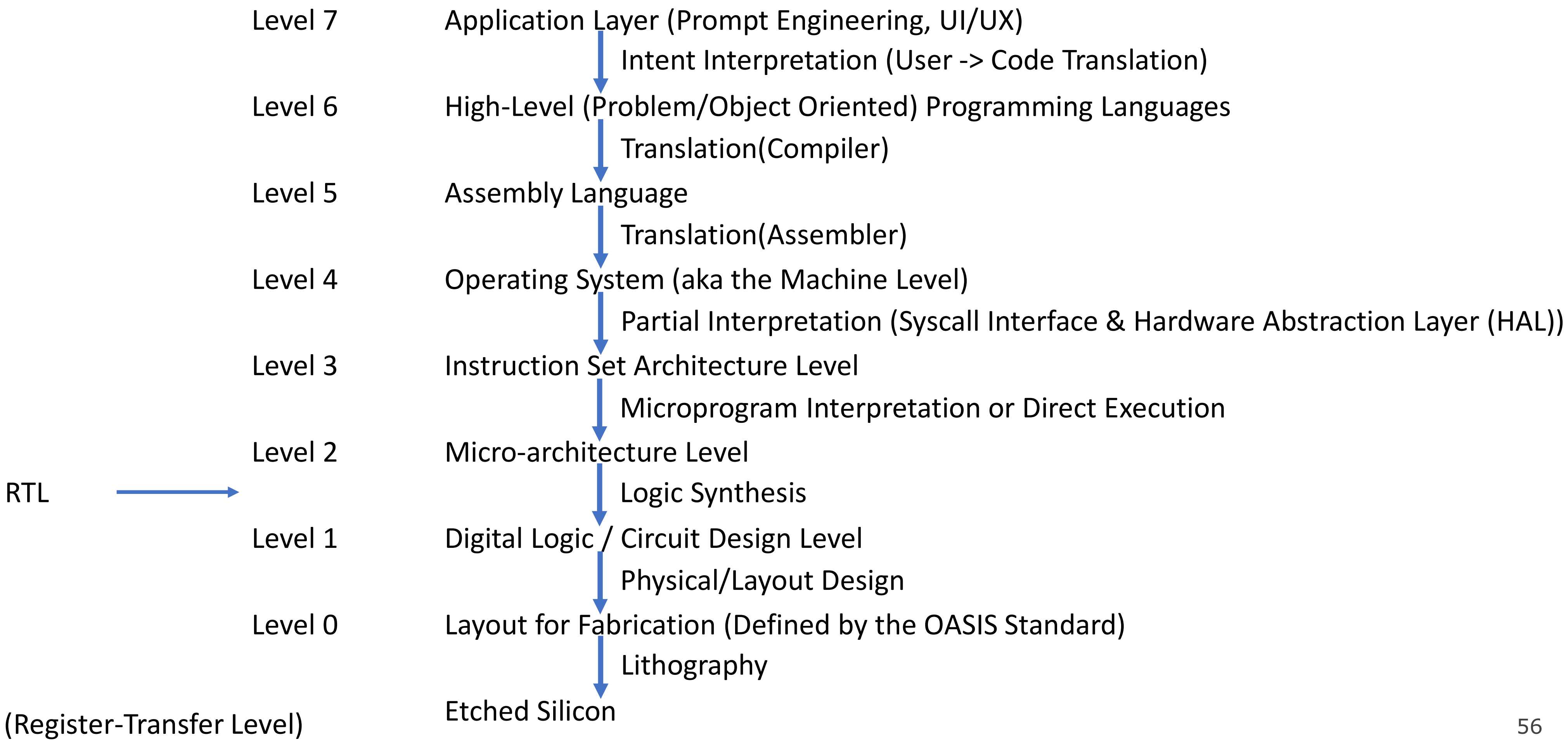
Lithography

Etched Silicon

VLSI

Very-Large-Scale Integration

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

RTL

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

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

Many Steps → Physical/Layout Design

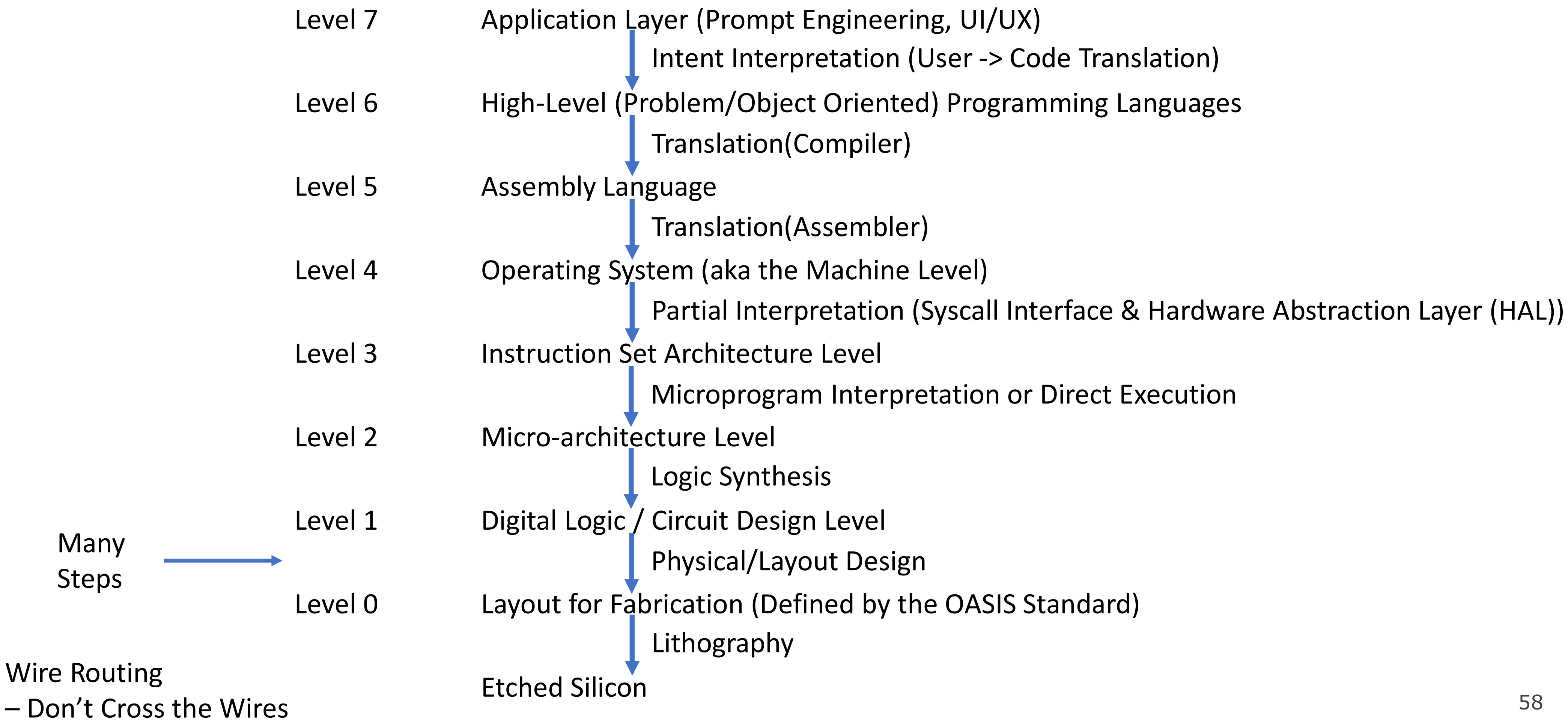
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

Floorplanning

Programming Levels

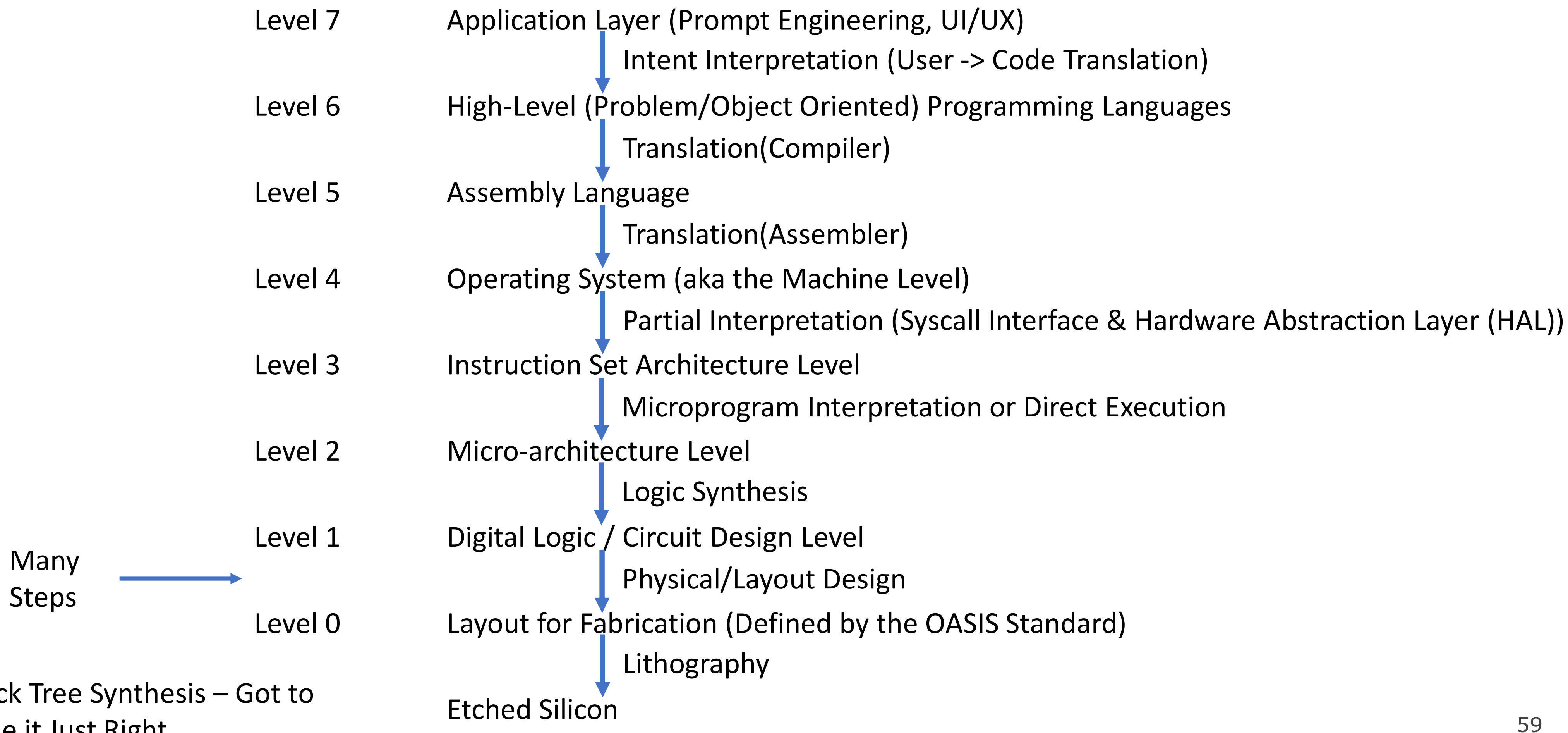


Level 7	Application Layer (Prompt Engineering, UI/UX)
	Intent Interpretation (User -> Code Translation)
Level 6	High-Level (Problem/Object Oriented) Programming Languages
	Translation(Compiler)
Level 5	Assembly Language
	Translation(Assembler)
Level 4	Operating System (aka the Machine Level)
	Partial 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

Many Steps → Physical/Layout Design

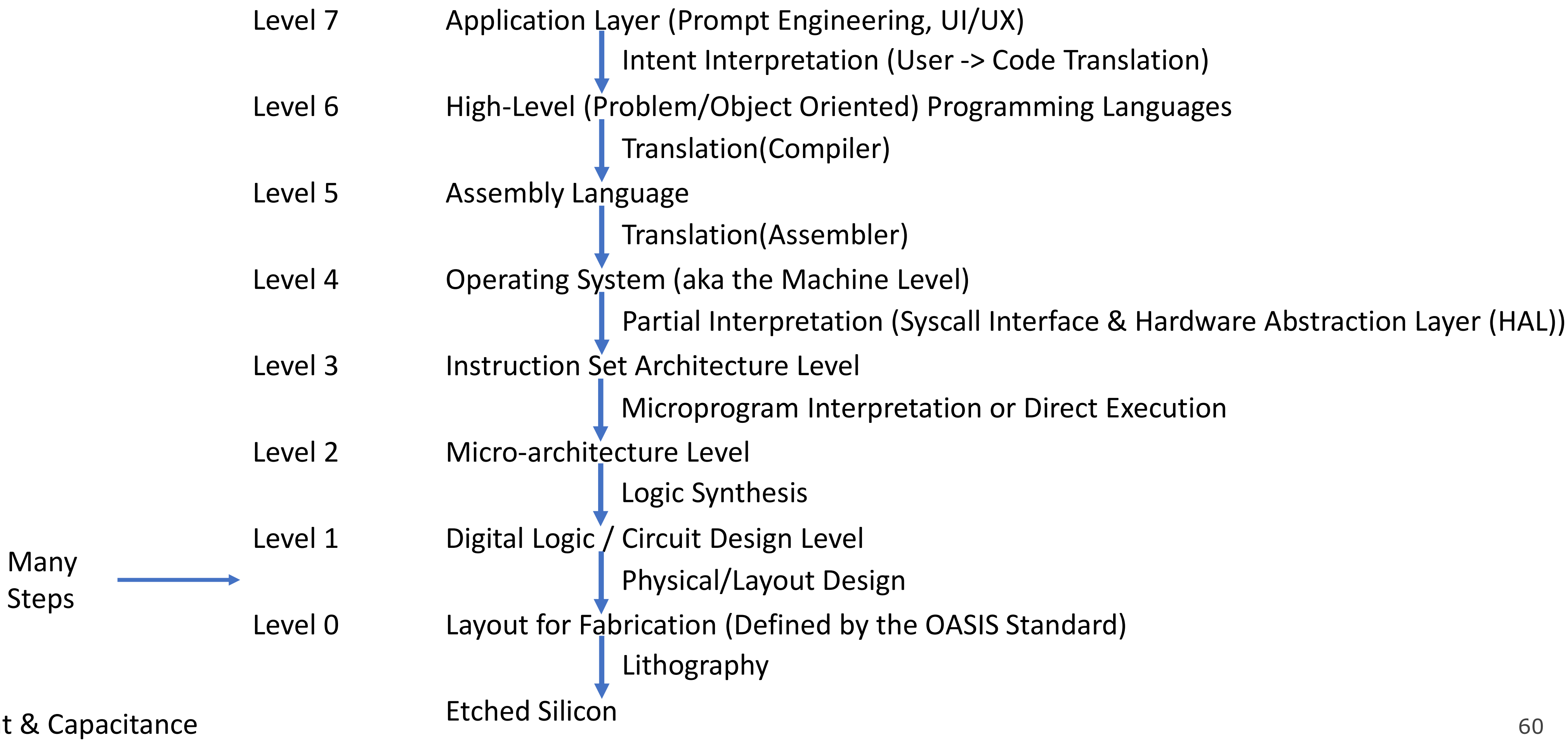
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

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

Many Steps

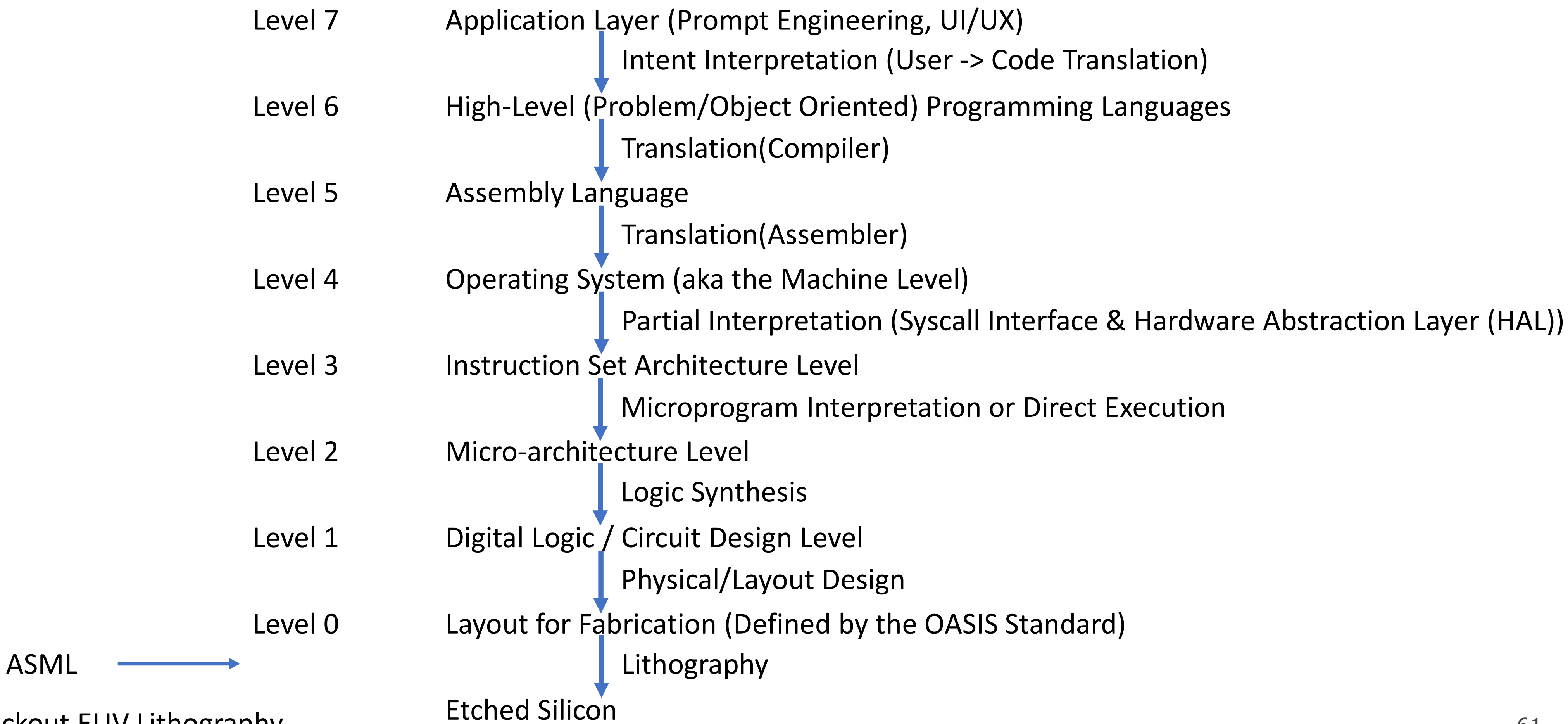
Physical/Layout Design

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

Lithography

Etched Silicon

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

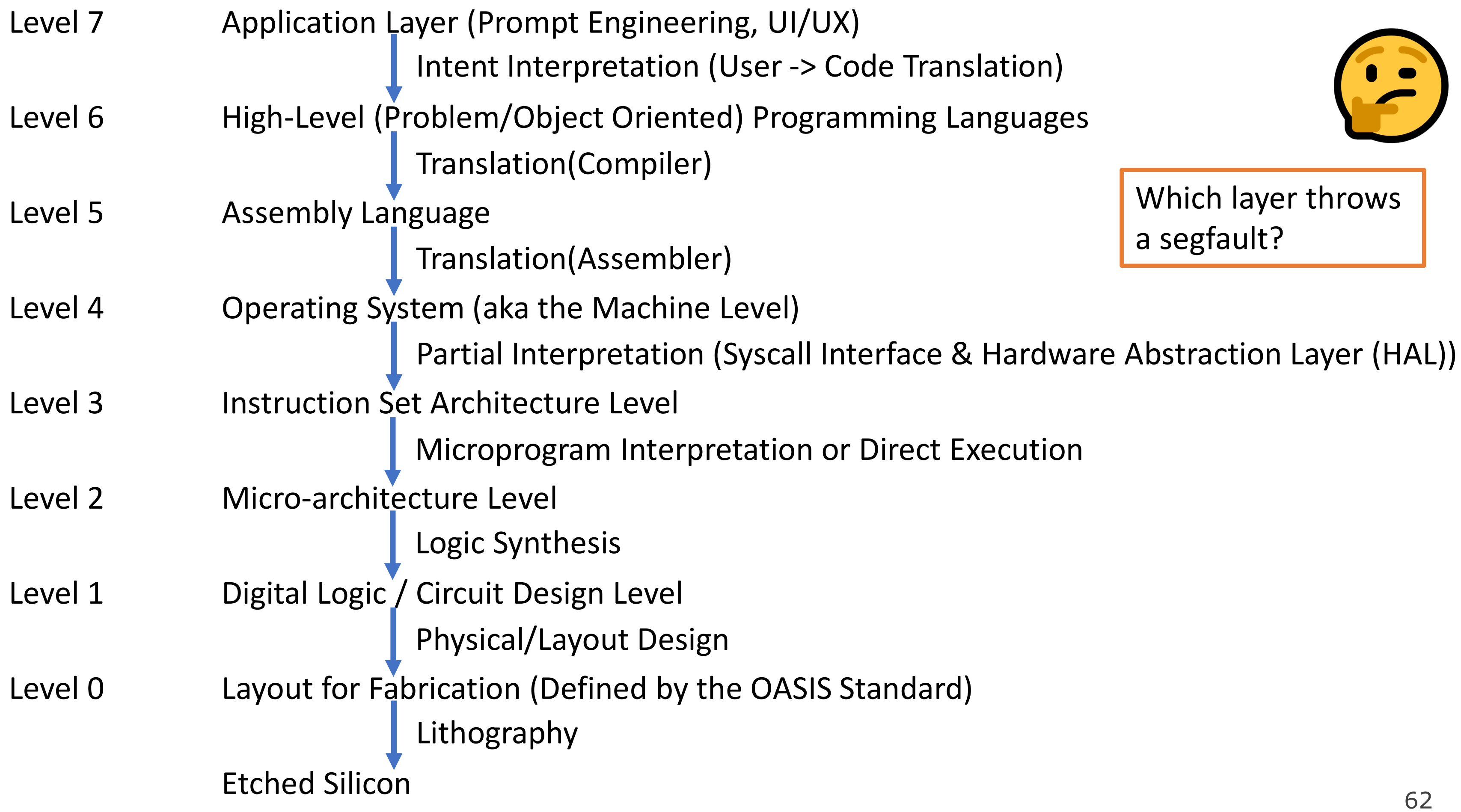
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

ASML → Lithography

Etched Silicon

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

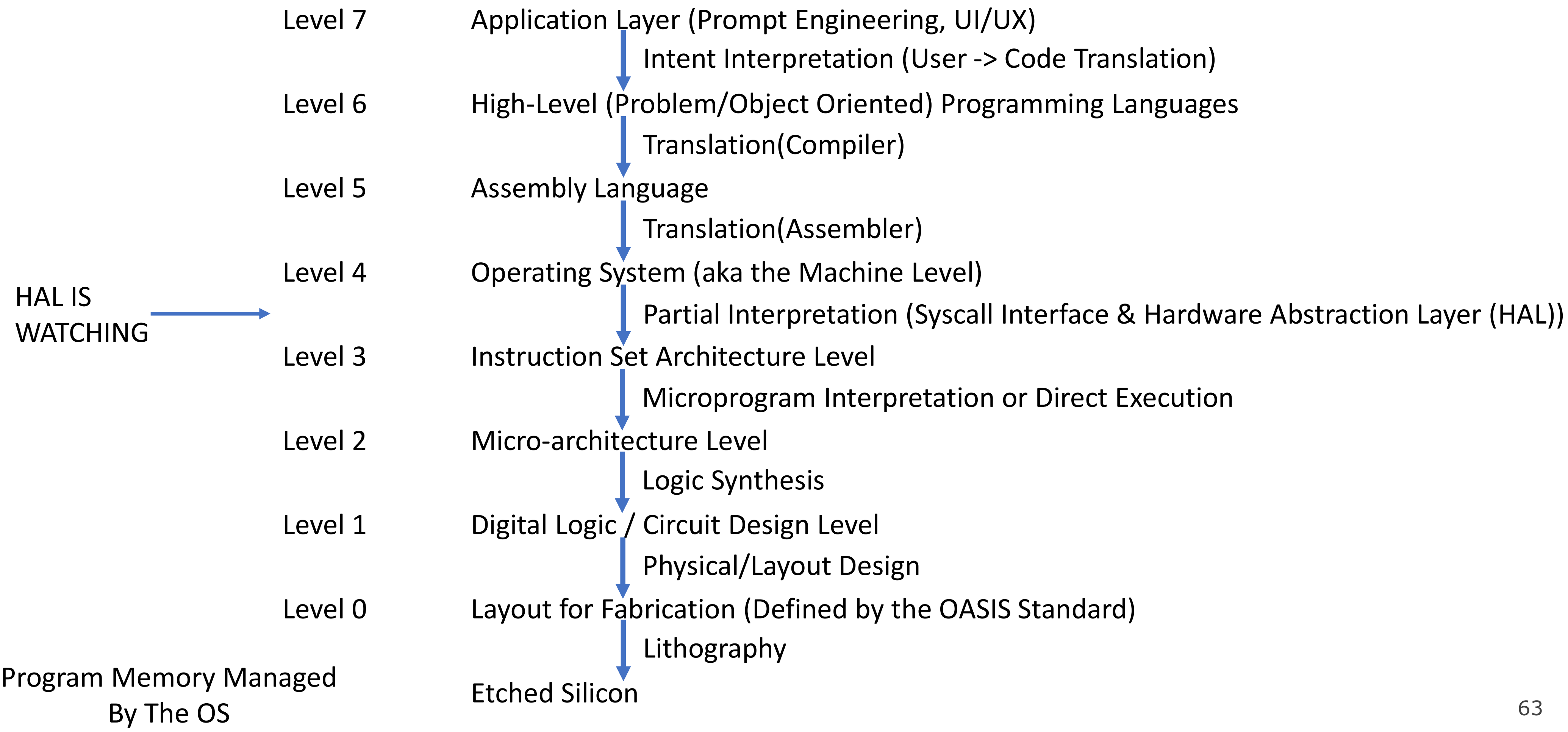
Level 0 Layout for Fabrication (Defined by the OASIS Standard)

Lithography

Etched Silicon

Which layer throws a segfault?

Programming Levels



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)

HAL IS WATCHING

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

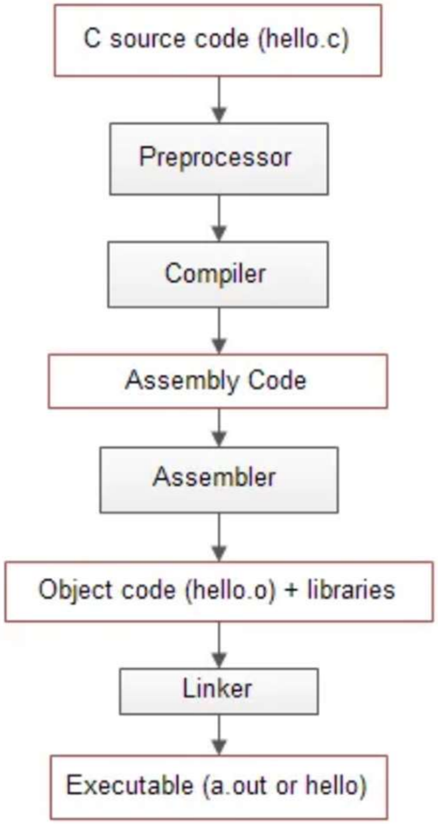
Program Memory Managed By The OS

More on the Compiler

How Does GCC Work?

- One Unix Command – A lot of steps!

gcc hello.c -o hello

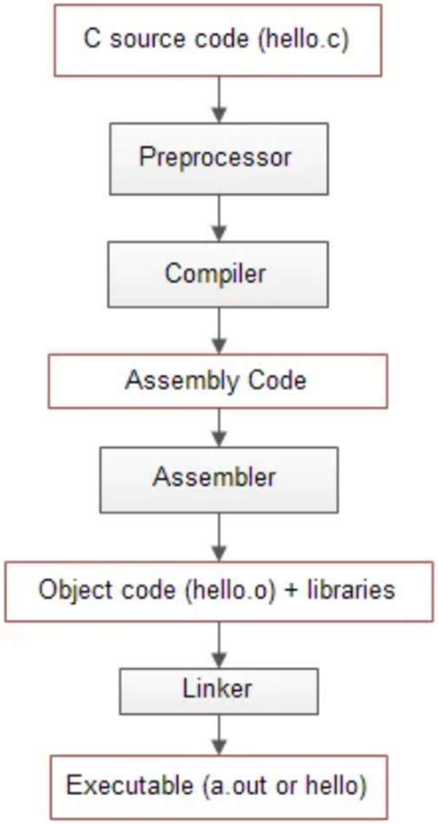


https://medium.com/@tuvo1106/the-gcc-compilation-process-8accb463e227

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

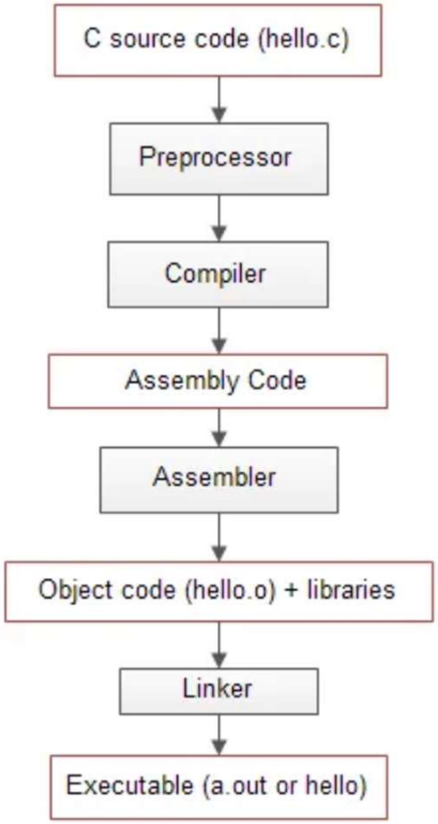
How Does GCC Work?

- 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?

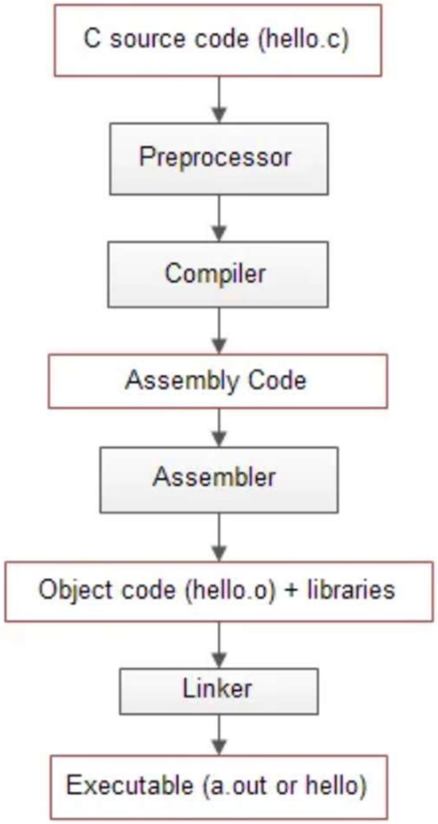
- Object Generation – C to Object File

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

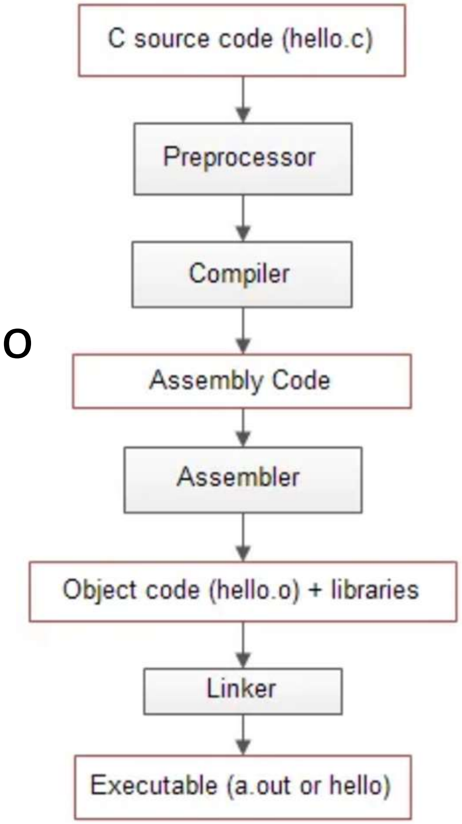
ld -o hello hello.o -lc -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o /usr/lib/x86_64-linux-gnu/crtn.o

- 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

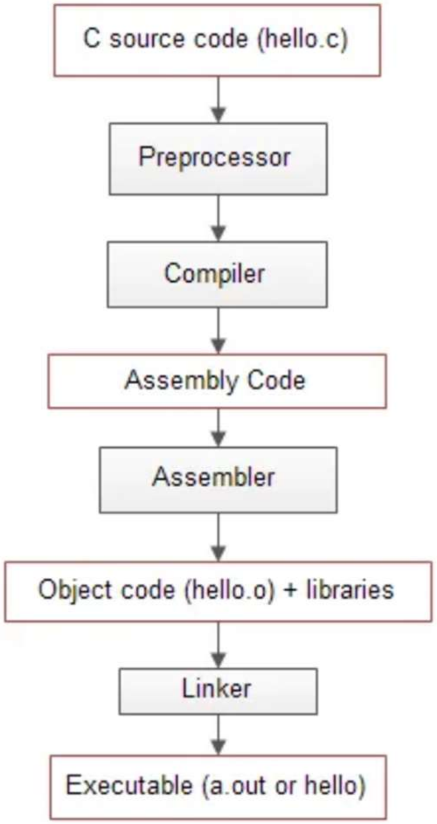
Why do we need a linker?

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

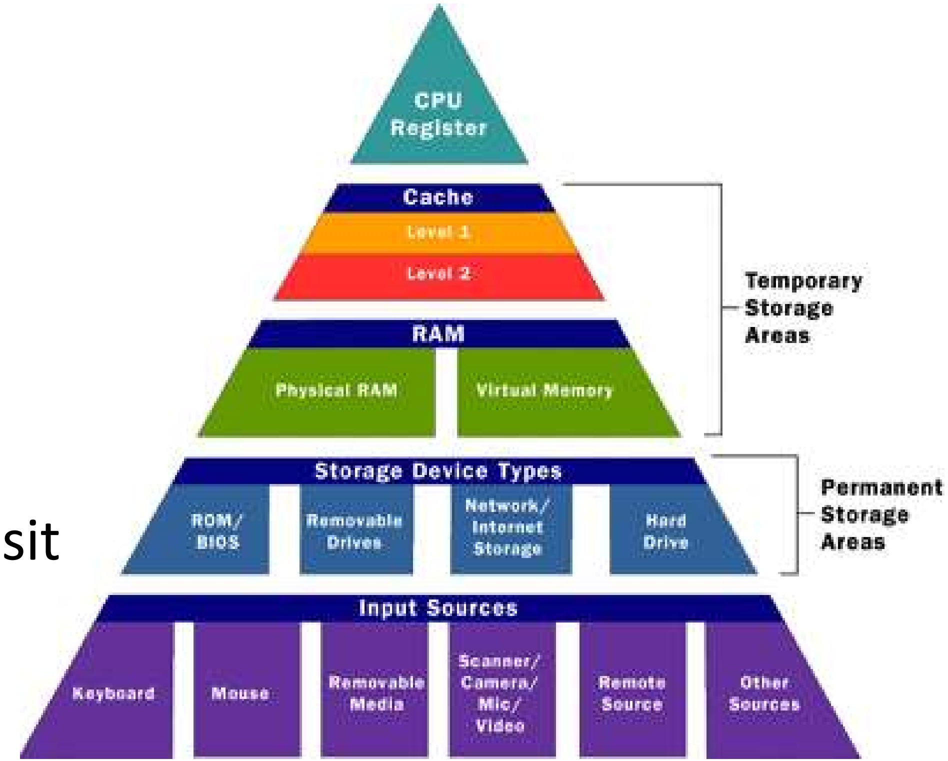
Peeking at Memory

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

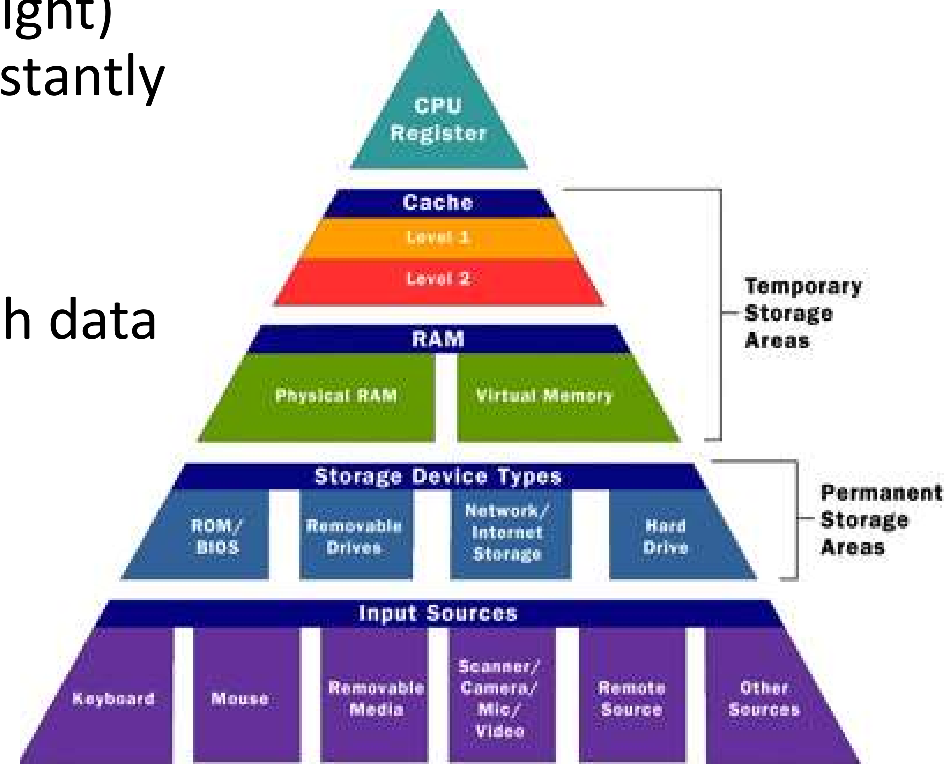
- The BUS (actual technical name) is our transit system around the computer



- Places close to the CPU are more limited and more valuable, since they can get to the CPU faster

Speed vs Space

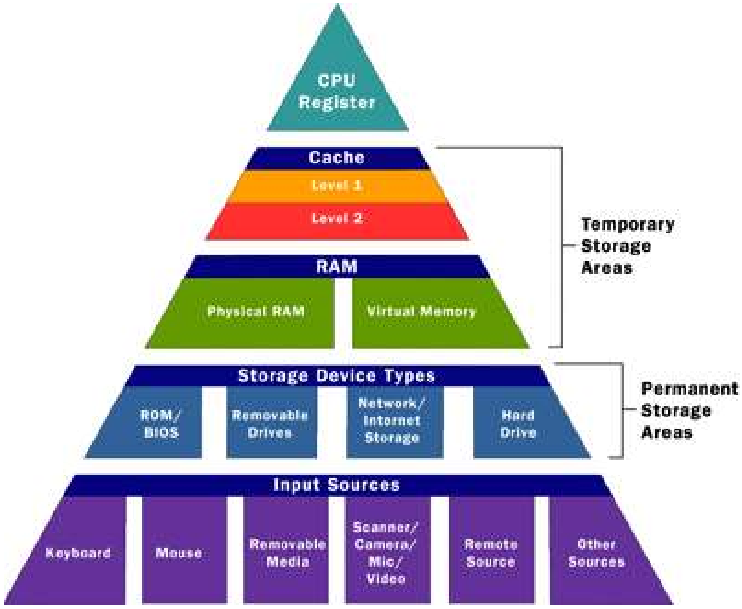
- All of Memory (Temporary Storage on the right) and the registers is rent only, so data is constantly moving around
- Many algorithms developed to decide which data gets to live where and for how long
- Proper access makes a huge difference on performance



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

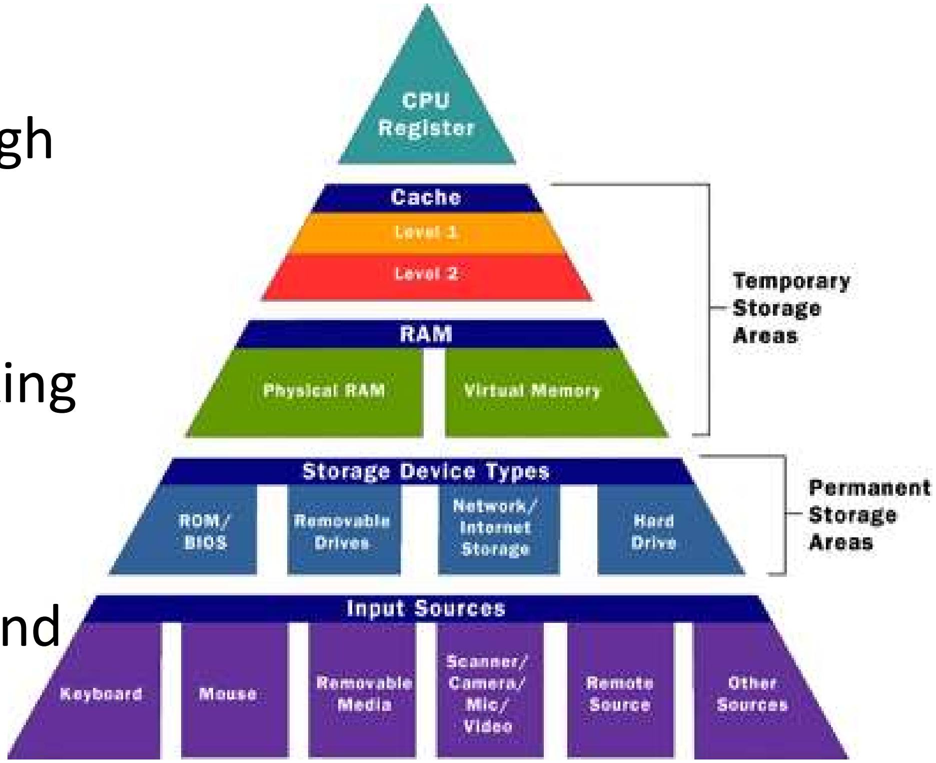
Speed vs Space

- 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

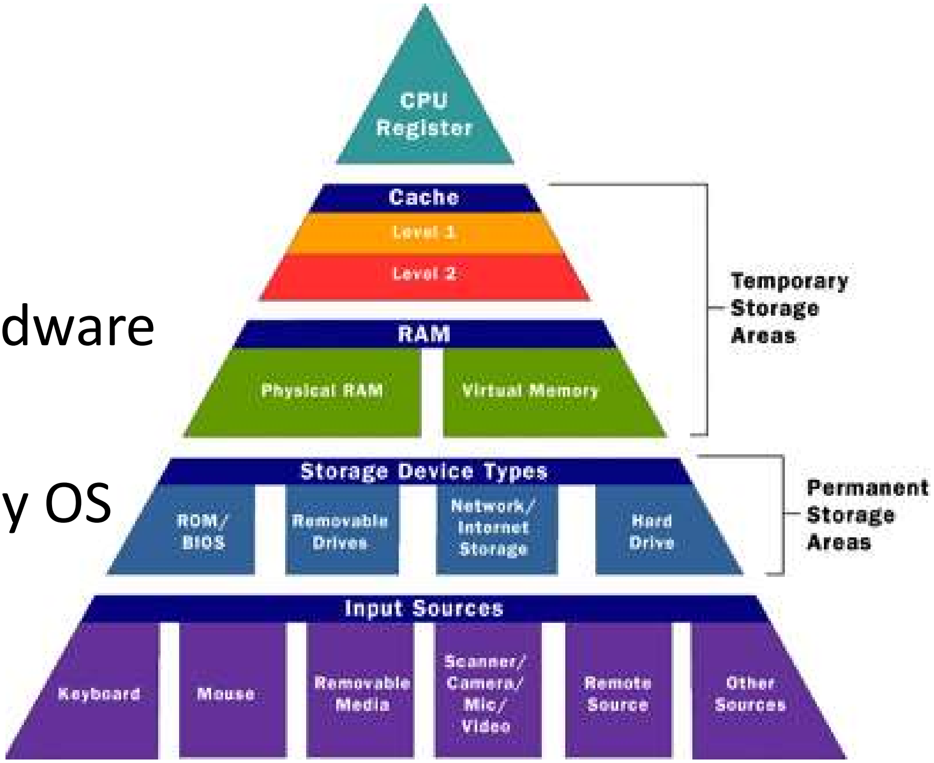


What is Locality?

- 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

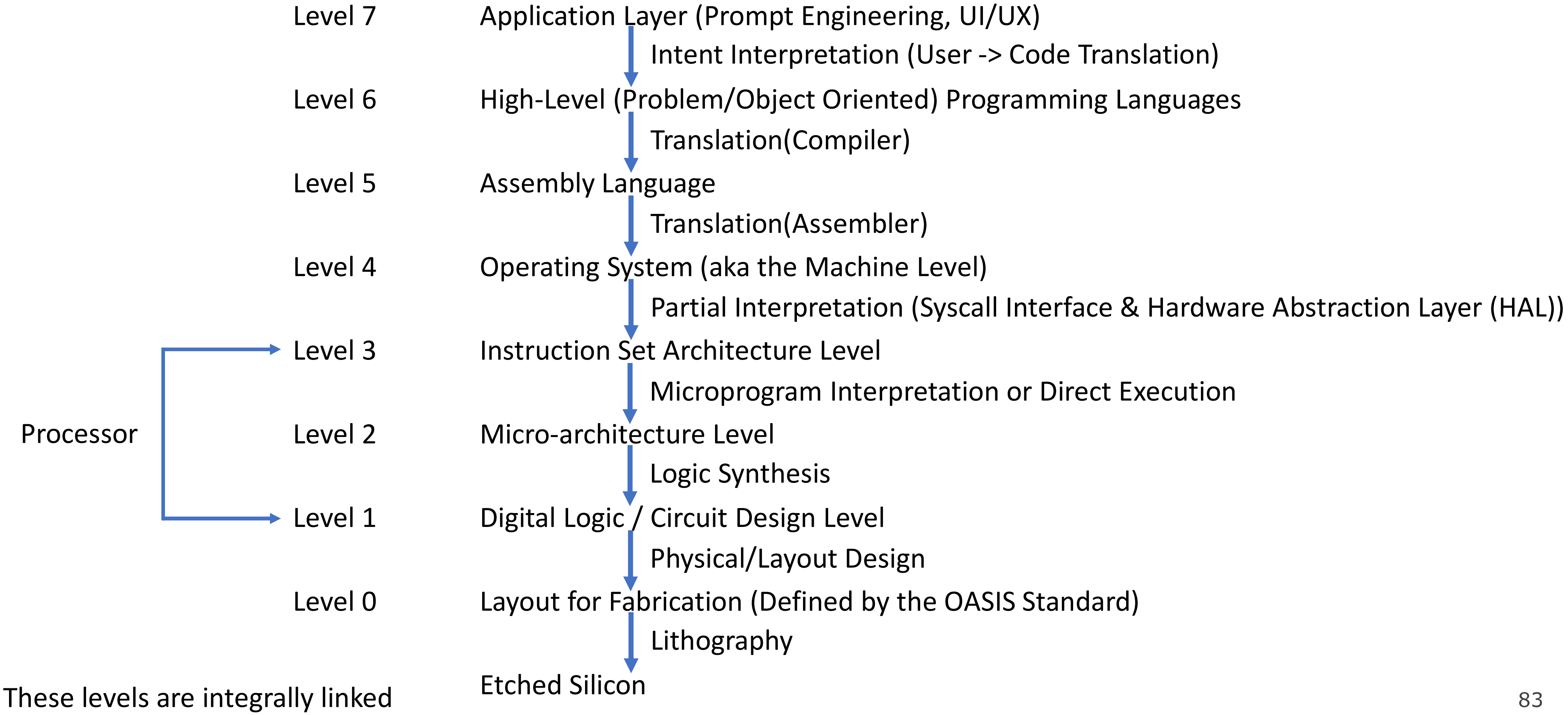
Who Gets to Manage the Memory?

- 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



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

Processor

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)

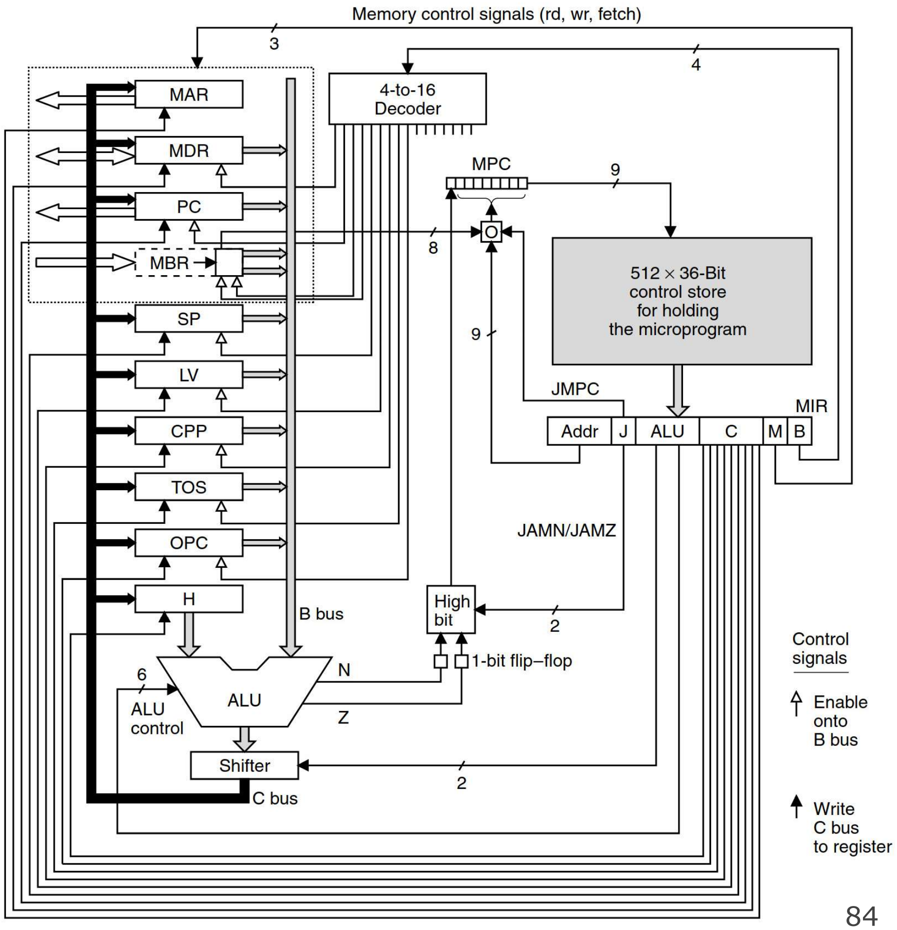
Lithography

Etched Silicon

These levels are integrally linked

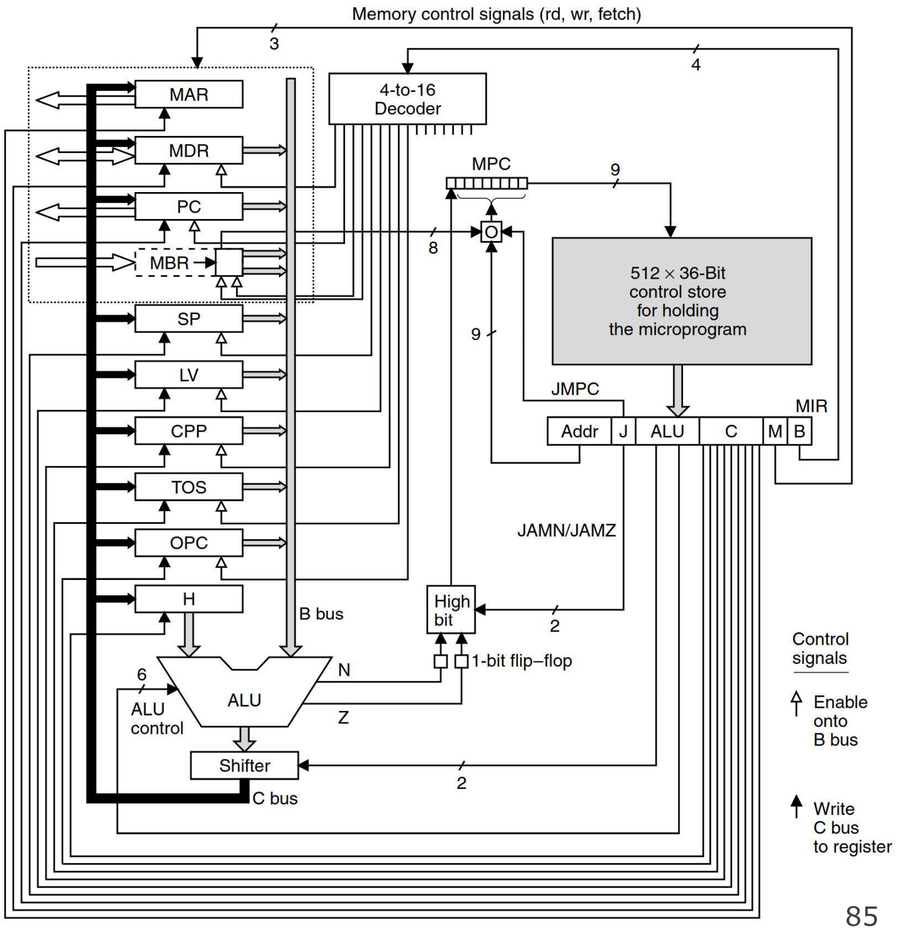
A 'Simple' Example

- 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

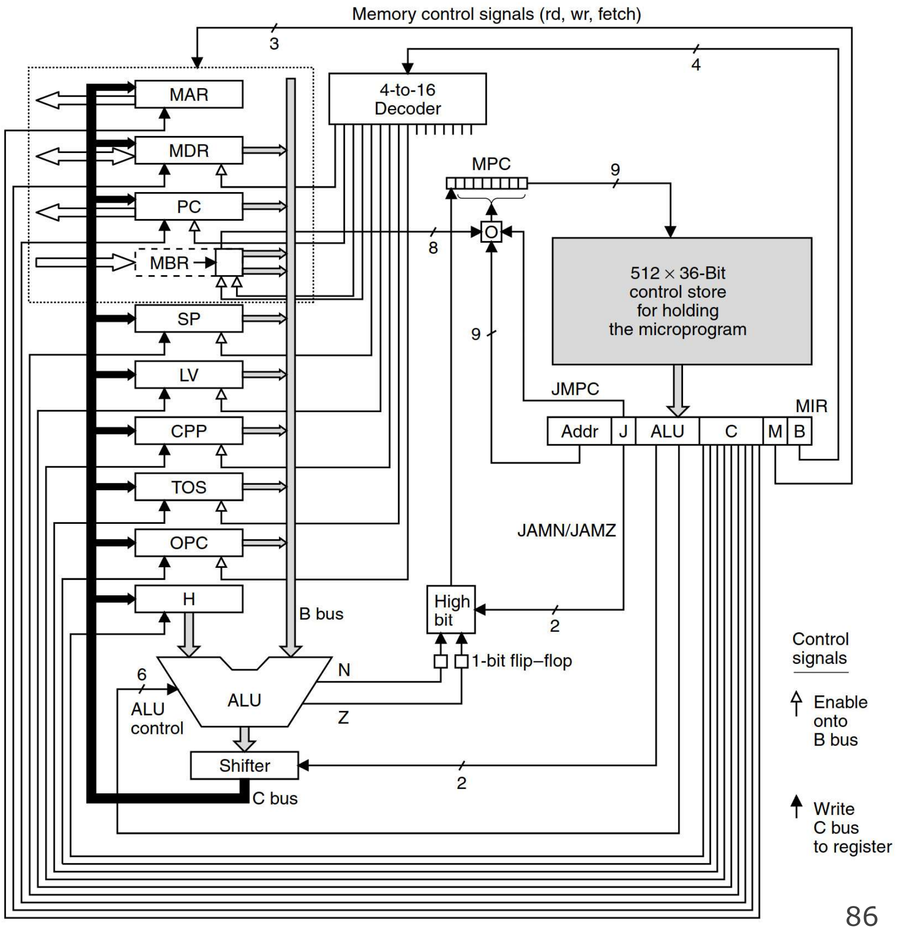
- ALU – Arithmetic & Logic Unit
    - Performs Math & Logic Operations

- 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

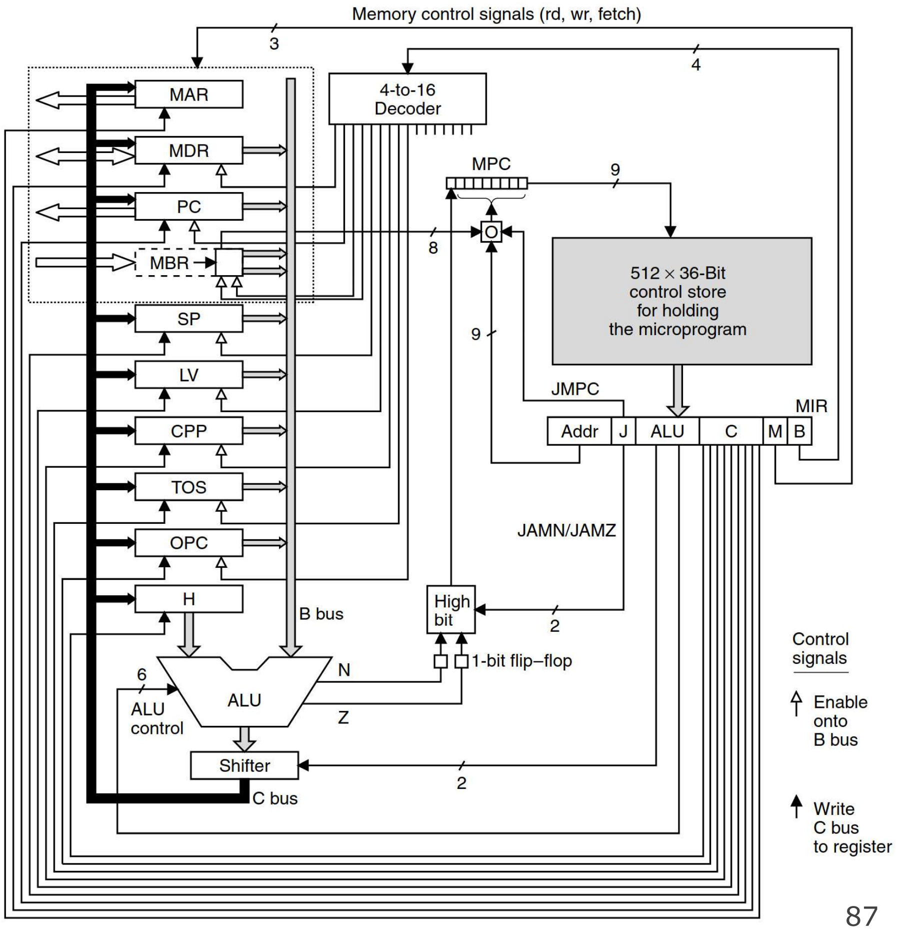


A ‘Simple’ Example

- 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

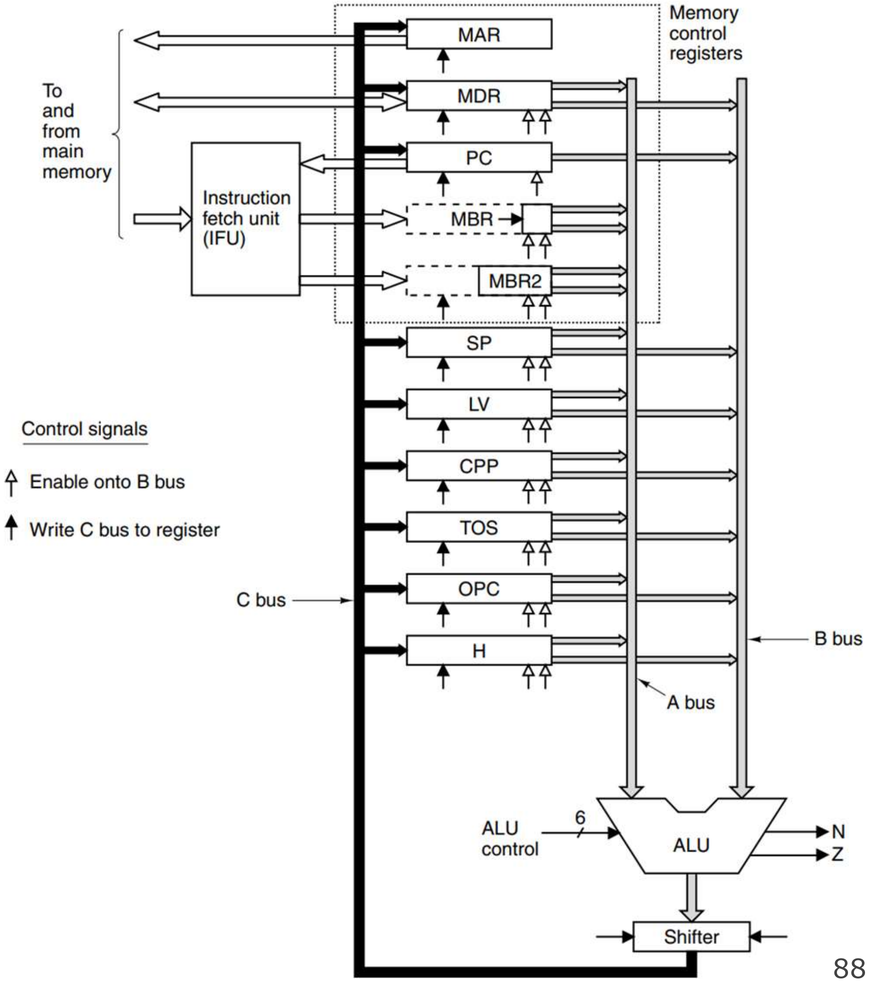
- This would be an example of a hardware based constraint



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



Some Extra Reading

Key GDB Tips For Assembly

- Examine 4 giant words (8 bytes) on the stack:

```
(gdb) x/4g $rsp
0x7ffffffe870: 0x0000000000000005      0x0000000000400559
0x7ffffffe880: 0x0000000000000000      0x0000000000400575
```

- display/undisplay (prints out things every time you step/next)

```
(gdb) display/4w $rsp
1: x/4xw $rsp
0x7ffffffe8a8:
0xf7a2d830      0x00007fff      0x00000000      0x00000000
```

Key GDB Tips For Assembly

- `stepi`/`finish`: step into current function call/return to caller:

```
(gdb) finish
```

- Set register values during the run

```
(gdb) p $rdi = $rdi + 1
```

(Might be useful to write down the original value of $rdi somewhere)

- Tui things
  - `refresh`
  - `focus cmd` – use up/down arrows on gdb command line (vs `focus asm`, `focus regs`)
  - `layout regs`, `layout asm`

gdb tips ⭐⭐⭐

`layout split` (ctrl-x a: exit, ctrl-l: resize)	View C, assembly, and gdb
`info reg`	Print all registers
`p $eax`	Print register value
`p $eflags`	Print all condition codes currently set
`b *0x400546`	Set breakpoint at assembly instruction
`b *0x400550 if $eax > 98`	Set **conditional breakpoint**
`ni`	Next assembly instruction
`si`	Step into assembly instruction (will step into function calls)

gdb tips ⭐⭐⭐

`p/x $rdi`	Print register value in hex
`p/t $rsi`	Print register value in binary
`x $rdi`	Examine the byte stored at this address
`x/4bx $rdi`	Examine 4 bytes starting at this address
`x/4wx $rdi`	Examine 4 ints starting at this address

References and Advanced

- References:
  - Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  - CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  - gdbtui: https://beej.us/guide/bggdb/
  - More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  - Compiler explorer: https://gcc.godbolt.org
- Advanced Reading:
  - Stack frame layout on x86-64: https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64
  - x86-64 Intel Software Developer manual: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
  - history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  - x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64