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:

<table>
<thead>
<tr>
<th>$K$</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>int, float</td>
</tr>
<tr>
<td>8</td>
<td>long, double, char *</td>
</tr>
</tbody>
</table>
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:

```c
struct S1 {
    int i;
    char c;
    int j;
};
```
- If the compiler used a minimal allocation:
  - 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  9
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 Common For Loop Output

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

GCC For Loop Output

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
...

// n = 100

...
GCC For Loop Output

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

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

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

// n = 100
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
```c
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 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++)
• 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!)
• **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.
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:
How many GCC 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** pre-calculates constants at compile-time where possible.

```c
int seconds = 60 * 60 * 24 * n_days;
```
**Constant Folding** pre-calculates constants at compile-time where possible.

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

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

```c
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

```c
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;
}
```
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                shr $0x2, %eax
11da:  01 d0                   add %edx, %eax
11dc:  8b 45 ec                mov %eax,-0x14(%rbp)
11df:  48 89 e5                mov %rsp,%rbp
11e0:  48 0f 6e c0              cvttsd2si %xmm0,%eax
11e4:  89 45 e4                mov %eax,-0x14(%rbp)
11e7:  0f af 45 cc               imul -0x14(%rbp),%eax
11eb:  48 63 d8                movslq %eax,%rbx
11ec:  48 8d 3d ed 0d 00 00    leaq 0x14(%rip), %rdi
11f0:  e8 20 fe ff ff          callq 1040 <strlen@plt>
11f4:  48 0f af c2               imul %rdx,%rax
11f8:  48 01 e0                   add %r12d,%eax
11fa:  48 83 c4 30             add $0x30,%rsp
11fd:  5b                      pop %rbx
11fe:  41 5c                   pop %r12
1200:  48 89 e5                mov %rsp,%rbp
1204:  48 0f af c2               imul %rdx,%rax
1208:  48 01 e0                   add %r12d,%eax
120a:  48 83 c4 30             add $0x30,%rsp
120c:  5b                      pop %rbx
120d:  41 5c                   pop %r12
120f:  5d                      pop %rbp
1210:  c3                      retq

# 2010 <_IO_stdin_used+0x10>
# 2008 <_IO_stdin_used+0x8>
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** prevents the recalculation of the same thing many times by doing it once and saving the result.

```c
int a = (param2 + 0x107);
int b = param1 * (param2 + 0x107) + a;
return a * (param2 + 0x107) + b * (param2 + 0x107);
```
Common Sub-Expression Elimination prevents the recalculation of the same thing many times by doing it once and saving the result.

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

```
000000000000011b0 <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
```
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:

```c
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)

Dead Code: Before (-O0)

0000000000000011a9 <dead_code>:

```
11a9: 55
11aa: 48 89 e5
11ad: 48 83 ec 20
11b1: 89 7d ec
11b4: 89 75 e8
11b7: 8b 45 ec
11ba: 3b 45 e8
11bd: 7d 19
11bf: 8b 45 ec
11c2: 3b 45 e8
11c5: 7e 11
11c7: 48 8d 3d 0e 00 00
11ce: b8 00 00 00 00
11d3: e8 eb ff ff
11d8: c7 45 fc 00 00 00 00
11df: eb 04
11e1: 83 45 fc 01
11e5: 81 7d fc e7 03 00 00
11ec: 7e f3
11ee: 8b 45 ec
11f1: 3b 45 e8
11f4: 75 06
11f6: 83 45 ec 01
11fa: eb 04
11fc: 83 45 ec 01
1200: 83 7d ec 00
1204: 75 07
1206: b8 00 00 00 00
120b: eb 03
120d: 8b 45 ec
1210: c9
1211: c3
```

push  %rbp
mov  %rsp,%rbp
sub  $0x20,%rsp
mov  %edi,-0x14(%rbp)
mov  %esi,-0x18(%rbp)
mov  -0x14(%rbp),%eax
cmp  -0x18(%rbp),%eax
jge  11d8 <dead_code+0x2f>
mov  -0x14(%rbp),%eax
cmp  -0x18(%rbp),%eax
jle  11d8 <dead_code+0x2f>
lea  0xe36(%rip),%rdi # 2004 <_IO_stdin_used+0x4>
```
```
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).

```c
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.

```java
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.

```c
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.
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?
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: 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.

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

Program Specific Interactions  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

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  
  - 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

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
  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

GNU Assembler (Inside GCC)
  Etched Silicon
Programming Levels

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

OS Manages Program -> Hardware
# Programming Levels

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

**Processing the Machine Code**

**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
  - Logic Synthesis

Level 1: Digital Logic / Circuit Design Level
  - Physical/Layout Design

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

RTL (Register-Transfer Level)
  - 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

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

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Layout for Fabrication (Defined by the OASIS Standard)</td>
</tr>
<tr>
<td></td>
<td>Lithography</td>
</tr>
<tr>
<td></td>
<td>Etched Silicon</td>
</tr>
<tr>
<td>Level 1</td>
<td>Digital Logic / Circuit Design Level</td>
</tr>
<tr>
<td></td>
<td>Physical/Layout Design</td>
</tr>
<tr>
<td>Many Steps</td>
<td></td>
</tr>
<tr>
<td>Clock Tree Synthesis – Got to Time it Just Right</td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>Micro-architecture Level</td>
</tr>
<tr>
<td></td>
<td>Logic Synthesis</td>
</tr>
<tr>
<td>Level 3</td>
<td>Instruction Set Architecture Level</td>
</tr>
<tr>
<td></td>
<td>Microprogram Interpretation or Direct Execution</td>
</tr>
<tr>
<td>Level 4</td>
<td>Operating System (aka the Machine Level)</td>
</tr>
<tr>
<td></td>
<td>Partial Interpretation (Syscall Interface &amp; Hardware Abstraction Layer (HAL))</td>
</tr>
<tr>
<td>Level 5</td>
<td>Assembly Language</td>
</tr>
<tr>
<td></td>
<td>Translation(Assembler)</td>
</tr>
<tr>
<td>Level 6</td>
<td>High-Level (Problem/Object Oriented) Programming Languages</td>
</tr>
<tr>
<td></td>
<td>Translation(Compiler)</td>
</tr>
<tr>
<td>Level 7</td>
<td>Application Layer (Prompt Engineering, UI/UX)</td>
</tr>
<tr>
<td></td>
<td>Intent Interpretation (User -&gt; Code Translation)</td>
</tr>
</tbody>
</table>

---

59
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
ASML
Checkout EUV Lithography
Etched Silicon
Programming Levels

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

Level 1  Digital Logic / Circuit Design Level
          Physical/Layout Design

Level 2  Micro-architecture Level
          Logic Synthesis

Level 3  Instruction Set Architecture Level
          Microprogram Interpretation or Direct Execution

Level 4  Operating System (aka the Machine Level)
          Partial Interpretation (Syscall Interface & Hardware Abstraction Layer (HAL))

Level 5  Assembly Language
          Translation(Assembler)

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

Level 7  Application Layer (Prompt Engineering, UI/UX)
          Intent Interpretation (User -> Code Translation)

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)
         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

Program Memory Managed By The OS

Etched Silicon

HAL IS WATCHING
More on the Compiler
How Does GCC Work?

- One Unix Command – A lot of steps!

```bash
gcc hello.c -o hello
```
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
```
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
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
How Does GCC Work?

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

```bash
gcc hello.o -o hello
```

```bash
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

```bash
gcc -c hello.c
```

OR

```bash
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!
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
• 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

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

For more on speed checkout:
- [https://gist.github.com/jboner/2841832](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

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

These levels are integrally linked

Etched Silicon
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
  0x7fffffffffe870: 0x0000000000000005 0x0000000000400559
  0x7fffffffffe880: 0x0000000000000000 0x0000000000400575

• display/undisplay (prints out things every time you step/next)
  (gdb) display/4w $rsp
  1: x/4xw $rsp
  0x7fffffffffe8a8:
  0xf7a2d830 0x00007fff 0x00000000 0x00000000
• **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
layout split  (ctrl-x a: exit, ctrl-l: resize)
info reg

p $eax
p $eflags

b *0x400546
b *0x400550 if $eax > 98

ni
si

View C, assembly, and gdb
Print all registers
Print register value
Print all condition codes currently set
Set breakpoint at assembly instruction
Set conditional breakpoint
Next assembly instruction
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
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64