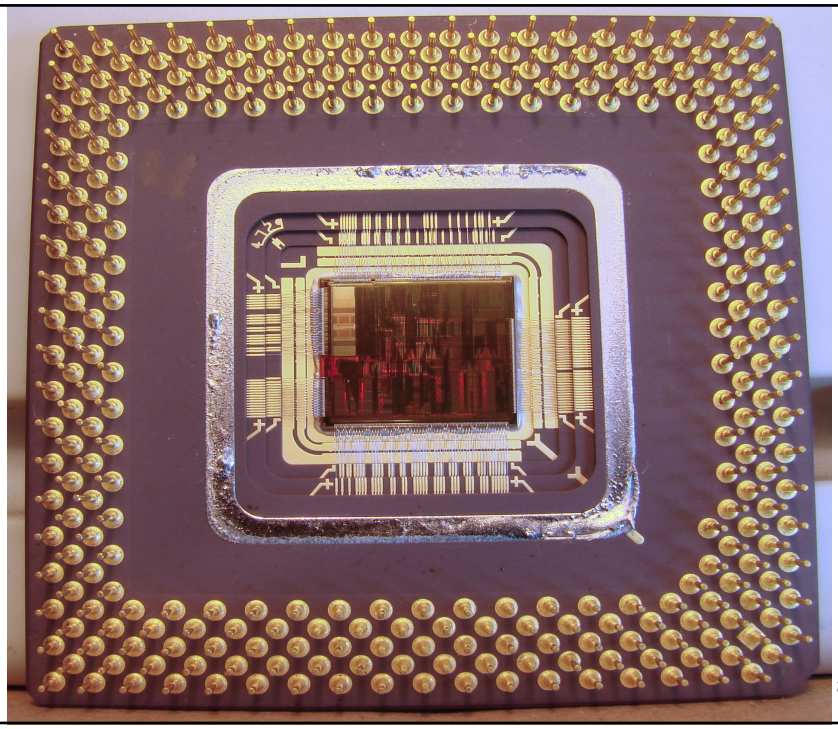
CS 444/544 Compilers
Ozyegin University
http://aktemur.github.io/cs544

Note by Baris Aktemur:
Our slides are adapted from Cooper and Torczon's slides that they prepared for COMP 412 at Rice.

0

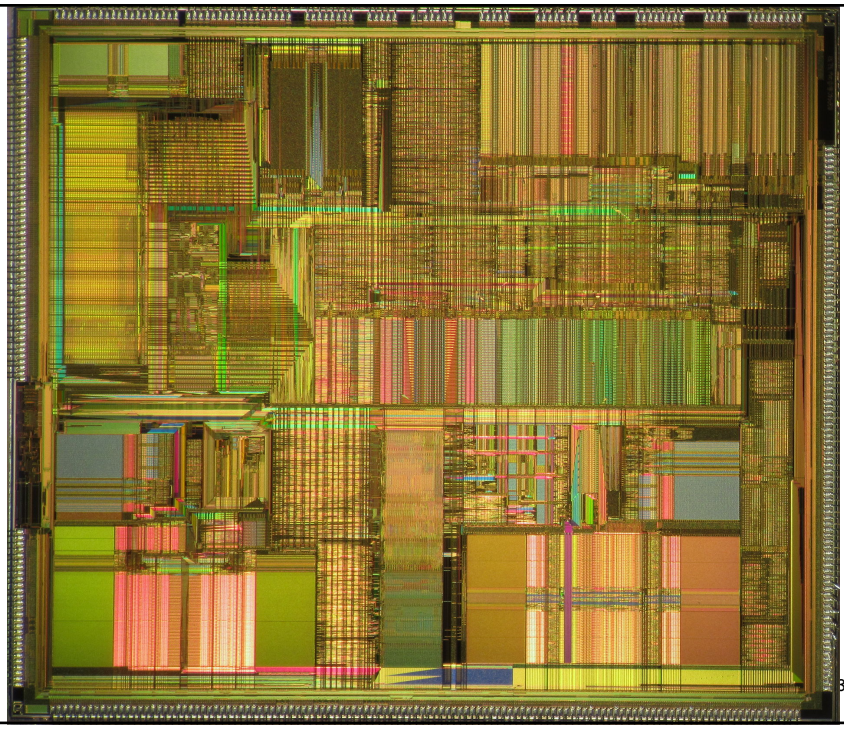
nextmedia.com.au

wikipedia



2

wikipedia



3

Instructing a CPU

- Wires go in, wires come out
- Low voltage, high voltage
- 0 and 1

$$\sum_{i=1}^{n} i$$

4

```
11001111 11111010 11101101 11111110 00000111 00000000
00000000 00000001 00000011 00000000 00000000 00000000
00000001 00000000 00000000 00000000 00000011 00000000
...
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 01010101 01001000
10001001 11100101 10000101 11111111 01111110 00010011
10001101 01001111 11111110 10001101 01000111 11111111
01001000 00001111 10101111 11000001 01001000 11010001
11101000 10001101 01000100 00111000 11111111 01011101
11000011 00110001 11000000 01011101 11000011 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00011111 00000000 00000000 00000000 00000000
00000000 00000000 00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00010100 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000001 01111010
01010010 00000000 00000001 01111000 00010000 00000001
00010000 00001100 00000111 00001000 10010000 00000001
00000000 00000000 00100100 00000000 00000000 00000000
00011100 00000000 00000000 00000000 10100000 11111111
11111111 11111111 11111111 11111111 11111111 11111111
00011111 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 01000001 00001110 00010000
10000110 00000010 01000011 00001101 00000110 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000001 00000000
00000000 00000110 00001110 00000000 00000000 00000000
00001110 00000011 00000000 00000000 01000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000 00001111 00000001
00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000110 00000000
00000000 00000000 00001111 00000011 00000000 00000000
01011000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 01011111 01100110 01101111
01101111 00000000 01011111 01100110 01101111 01101111
00101110 01100101 01101000 00000000 01000101 01001000
01011111 01100110 01110010 01100001 01101101 01100101
00110000 00000000 0.
```

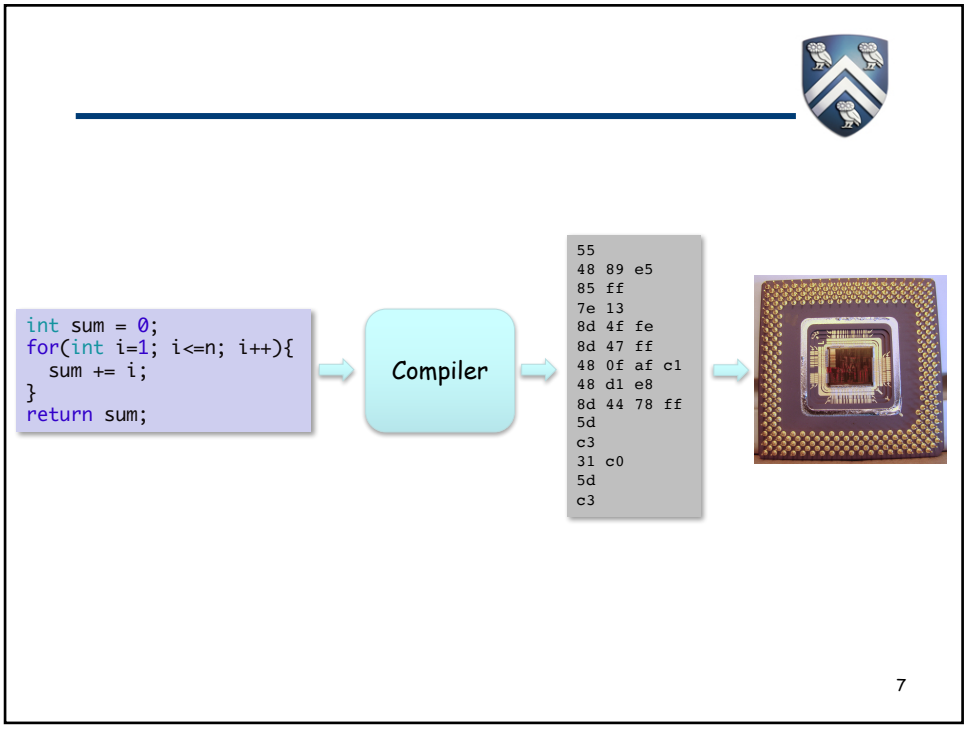
$$\sum_{i=1_5}^{n} i$$

```
55
48 89 e5
89 7d fc
c7 45 f8 00 00 00 00
c7 45 f4 00 00 00 00

8b 45 f4
3b 45 fc
0f 8d 1b 00 00 00
8b 45 f4
8b 4d f8
01 c1
89 4d f8
8b 45 f4
05 01 00 00 00
89 45 f4
e9 d9 ff ff ff

8b 45 f8
5d
c3
```

```
int sum = 0;
for(int i=1; i<=n; i++){
  sum += i;
}
return sum;
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    $0, -8(%rbp)
        movl    $0, -12(%rbp)
LBB0_1:
        movl    -12(%rbp), %eax
        , %eax
BB0_4
        ), %eax
        , %ecx
        ecx
        movl    %ecx, -8(%rbp)
        movl    -12(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rbp)
        jmp             LBB0_1
LBB0_4:
        movl    -8(%rbp), %eax
        popq    %rbp
        ret
```

$$\sum_{i=1}^{n} i$$



Compilers

- What is a compiler?
  - A program that translates an *executable* program in one language into an *executable* program in another language
  - The compiler should improve the program, *in some way*
- What is an interpreter?
  - A program that reads an *executable* program and produces the results of executing that program
- C,C++ are typically compiled. Scheme, Python are typically interpreted. Haskell, F#, Ocaml are compiled or interpreted, depending on the context.
- Java is compiled to bytecodes (code for the Java VM)
  - which are then interpreted
  - Or a hybrid strategy is used
    - → Just-in-time compilation

Common mis-statement:
X is an interpreted language
(or a compiled language)

Why Study Compilation?

- Compilers are important
  - Responsible for many aspects of system performance
  - Attaining performance has become more difficult over time
    - → In 1980, typical code got 85% or more of peak performance
    - → Today, that number is closer to 5 to 10% of peak
    - → Compiler has become a prime determiner of performance
- Compilers are interesting
  - Compilers include many applications of theory to practice
  - Writing a compiler exposes algorithmic & engineering issues
- Compilers are everywhere
  - Many practical applications have embedded languages
    - → Commands, macros, formatting tags …
  - Many applications have input formats that look like languages

Reducing the Price of Abstraction

Computer Science is the art of creating virtual objects and making them useful.

- We invent abstractions and uses for them
- We invent ways to make them efficient
- Programming is the way we realize these inventions

Well written compilers make abstraction affordable

- Cost of executing code should reflect the underlying work rather than the way the programmer chose to write it
- Change in expression should bring small performance change
- Cannot expect compiler to devise better algorithms
  - Don't expect bubblesort to become quicksort

Making Languages Usable

It was our belief that if FORTRAN, during its first months, were to translate any reasonable "scientific" source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.

— John Backus on the subject of the 1st FORTRAN compiler

Simple Examples

All data collected with gcc 4.1, -O3, running on a queiscent, multiuser Intel T9600 @ 2.8 GHz

Which is faster?

All three loops have distinct performance.

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    A[i][j] = 0;
```

0.51 sec on 10,000 x 10,000 array

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    A[j][i] = 0;
```

1.65 sec on 10,000 x 10,000 array

```
p = &A[0][0];
t = n * n;
for (i=0; i<t; i++)
  *p++ = 0;
```

0.11 sec on 10,000 x 10,000 array

A good compiler should know these tradeoffs, on each target, and generate the best code.

Few real compilers do.

Conventional wisdom suggests using

```
bzero((void*) &A[0][0],(size_t) n*n*sizeof(int))
```

0.52 sec on 10,000 x 10,000 array

Simple Examples

Example from Rⁿ Programming Environment, Rice, circa 1984

Abstraction has its price (& it is often higher than expected)

```
struct point {    /* Point on the plane of windows */
  int x; int y;
}

void Padd(struct point p, struct point q, struct point * r)
{
  r->x = p.x + q.x;
  r->y = p.y + q.y;
}

int main( int argc, char *argv[] )
{
  struct point p1, p2, p3;

  p1.x = 1; p1.y = 1;
  p2.x = 2; p2.y = 2;

  Padd(p1, p2, &p3);

  printf("Result is <%d,%d>.\n", p3.x, p3.y);
}
```

Simple Example (point add)

gcc 4.1, -S option

```
_main:  (some boilerplate code ellided for brevity's sake)
L5:
        popl    %ebx
        movl    $1, -16(%ebp)
        movl    $1, -12(%ebp)
        movl    $2, -24(%ebp)
        movl    $2, -20(%ebp)
        leal    -32(%ebp), %eax
        movl    %eax, 16(%esp)
        movl    -24(%ebp), %eax
        movl    -20(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 12(%esp)
        movl    -16(%ebp), %eax
        movl    -12(%ebp), %edx
        movl    %eax, (%esp)
        movl    %edx, 4(%esp)
        call    _PAdd
        movl    -28(%ebp), %eax
        movl    -32(%ebp), %edx
        movl    %eax, 8(%esp)
        movl    %edx, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $68, %esp
        popl    %ebx
        leave
        ret
```

Code for Intel Core 2 Duo

Assignments to p1 and p2

Setup for call to PAdd

Setup for call to printf

Address calculation for format string in printf call

Simple Example (point add)

gcc 4.1, -S option

```
_PAdd:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %edx
        movl    16(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, (%eax)
        movl    12(%ebp), %edx
        movl    20(%ebp), %eax
        addl    %eax, %edx
        movl    24(%ebp), %eax
        movl    %edx, 4(%eax)
        leave
        ret
```

Code for PAdd

Actual work

The code does a lot of work to execute two add instructions.

→ Factor of 10 in overhead

→ And a window system does a lot of point adds

Code optimization (careful compile-time reasoning & transformation) can make matters better.

Simple Example (point add)

gcc 4.1, -S –O3 option

```
_main:  (some boilerplate code ellided for brevity's sake)
L5:
        popl    %ebx
        subl    $20, %esp
        movl    $3, 8(%esp)
        movl    $3, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $20, %esp
        popl    %ebx
        leave
        ret
```

It inlined PAdd and folded the known constant values of p1 and p2.

With the right information, a good compiler can work wonders.
→ It kept the body of PAdd because it could not tell if it was dead

What if it could not discern the values of p1 and p2?

Simple Example (point add)

gcc 4.1, -S –O3 option

```
_main:  (some boilerplate code ellided for brevity's sake)
L5:
        popl    %ebx
        subl    $20, %esp
        movl    _one-"L00000000001$pb"(%ebx), %eax
        addl    _two-"L00000000001$pb"(%ebx), %eax
        movl    %eax, 8(%esp)
        movl    %eax, 4(%esp)
        leal    LC0-"L00000000001$pb"(%ebx), %eax
        movl    %eax, (%esp)
        call    L_printf$stub
        addl    $20, %esp
        popl    %ebx
        leave
        ret
```

I put 1 and 2 in global variables named "one" and "two".

The optimizer inlined PAdd

The optimizer recognized that
p1.x = p1.y and p2.x = p2.y
so
p1.x + p2.x = p1.y + p2.y.

If I make PAdd static, it deletes the code for PAdd

This code shows the more general version. It inlined PAdd and subjected the arguments to local optimization. It still had to perform the adds, but it recognized that the second one was redundant.
→ Gcc did a good job on this example.

Intrinsic Merit

- Compiler construction poses challenging and interesting problems:
  - Compilers must process large inputs, perform complex algorithms, but also run quickly
  - Compilers have primary responsibility for run-time performance
  - Compilers are responsible for making it acceptable to use the full power of the programming language
  - Computer architects perpetually create new challenges for the compiler by building more complex machines
    - → Compilers must hide that complexity from the programmer
- A successful compiler requires mastery of the many complex interactions between its constituent parts

Intrinsic Interest

- Compiler construction involves ideas from many different parts of computer science

Artificial intelligence	Greedy algorithms Heuristic search techniques
Algorithms	Graph algorithms, union-find Dynamic programming
Theory	DFAs & PDAs, pattern matching Fixed-point algorithms
Systems	Allocation & naming, Synchronization, locality
Architecture	Pipeline & hierarchy management Instruction set use

Why Does This Matter Today?

In the last years, most processors have gone multicore

- The era of clock-speed improvements is drawing to an end
  - Faster clock speeds mean higher power (n^2 *effect)*
  - Smaller wires mean higher resistance for on-chip wires
- For the near term, performance improvement will come from placing multiple copies of the processor (*core*) on a single die
  - Classic programs, written in old languages, are not well suited to capitalize on this kind of multiprocessor parallelism
    - → Parallel languages, some kinds of OO systems, functional languages
  - Parallel programs require sophisticated compilers
- Think of the Intel/AMD bet on multicore as a full-employment act for well-trained compiler writers