CSC 252: Computer Organization Spring 2021: Lecture 15

Instructor: Yuhao Zhu

Department of Computer Science

University of Rochester

Announcements

- Programming assignment 3 is **due today**
  - Details: https://www.cs.rochester.edu/courses/252/spring2021/labs/assignment3.html
  - Due on **March 23**, 11:59 PM
- Lab 2 grades are out.

7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23 **Today Due**	24	25 **Mid-term**	26	27

Announcements

- Mid-term exam: this Thursday, 12:30 PM — 1:45 PM; online.
- Past exam & Problem set: https://www.cs.rochester.edu/courses/252/spring2021/handouts.html
- Exam will be electronic using Gradescope, but we will send you an PDF version so that you can work offline in case
  - 1) you don't have stable Internet access at the exam time or
  - 2) the time doesn't work for you
  - Write down the answers on a scratch paper, take pictures, and send us the pictures
  - **Send me an email** explaining why you need to work offline before the exam data/time.
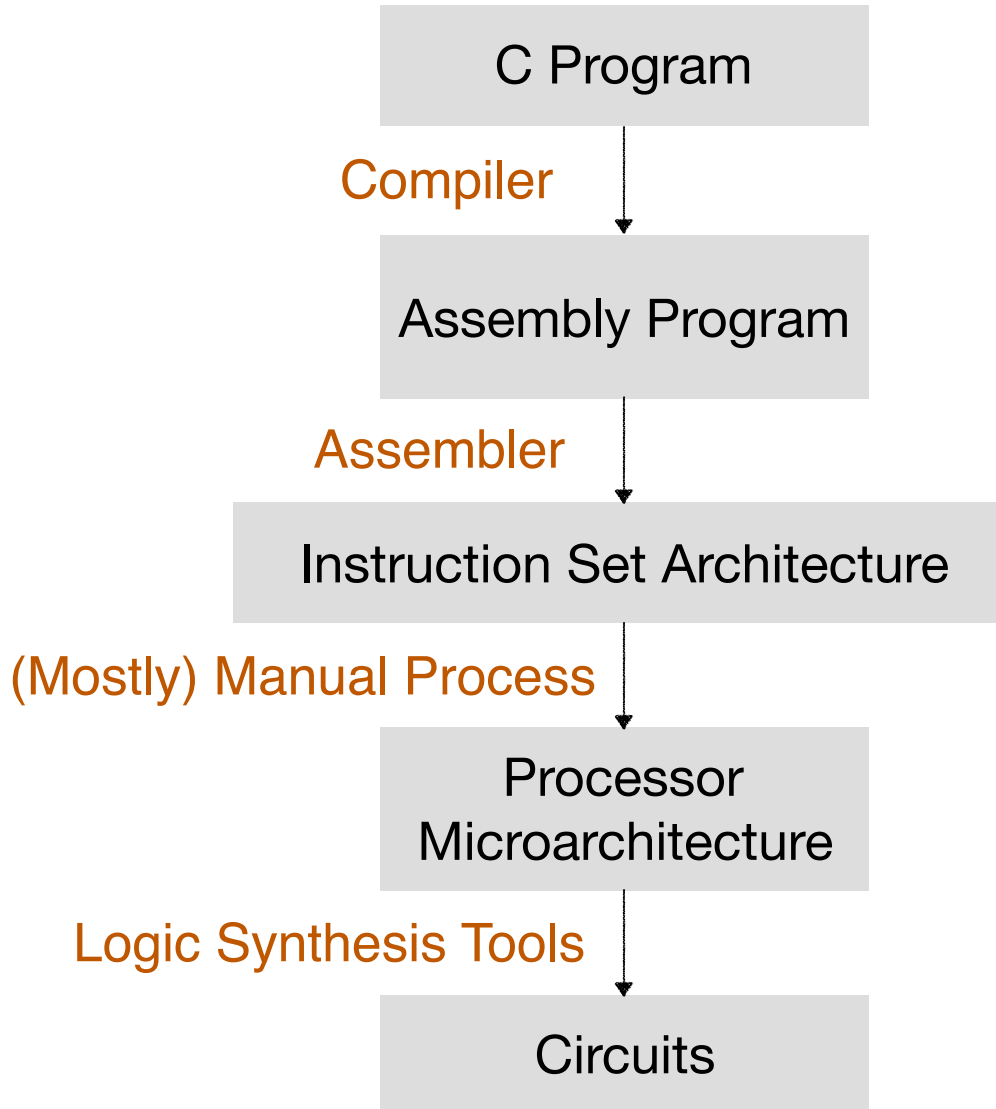
Announcements

- Exams are designed to test your ability to apply what you have learned and not your memory (though a good memory could help).

- Open book test: any sort of paper-based product, e.g., book, **notes**, magazine, old tests.

- **Nothing electronic (including laptop, cell phone, calculator, etc) other than the computer you use to take the exam**.

- You *can* use your electronic copy of the textbook, but you must not search in the textbook.

- **Nothing biological**, including your roommate, husband, wife, your hamster, another professor, etc.

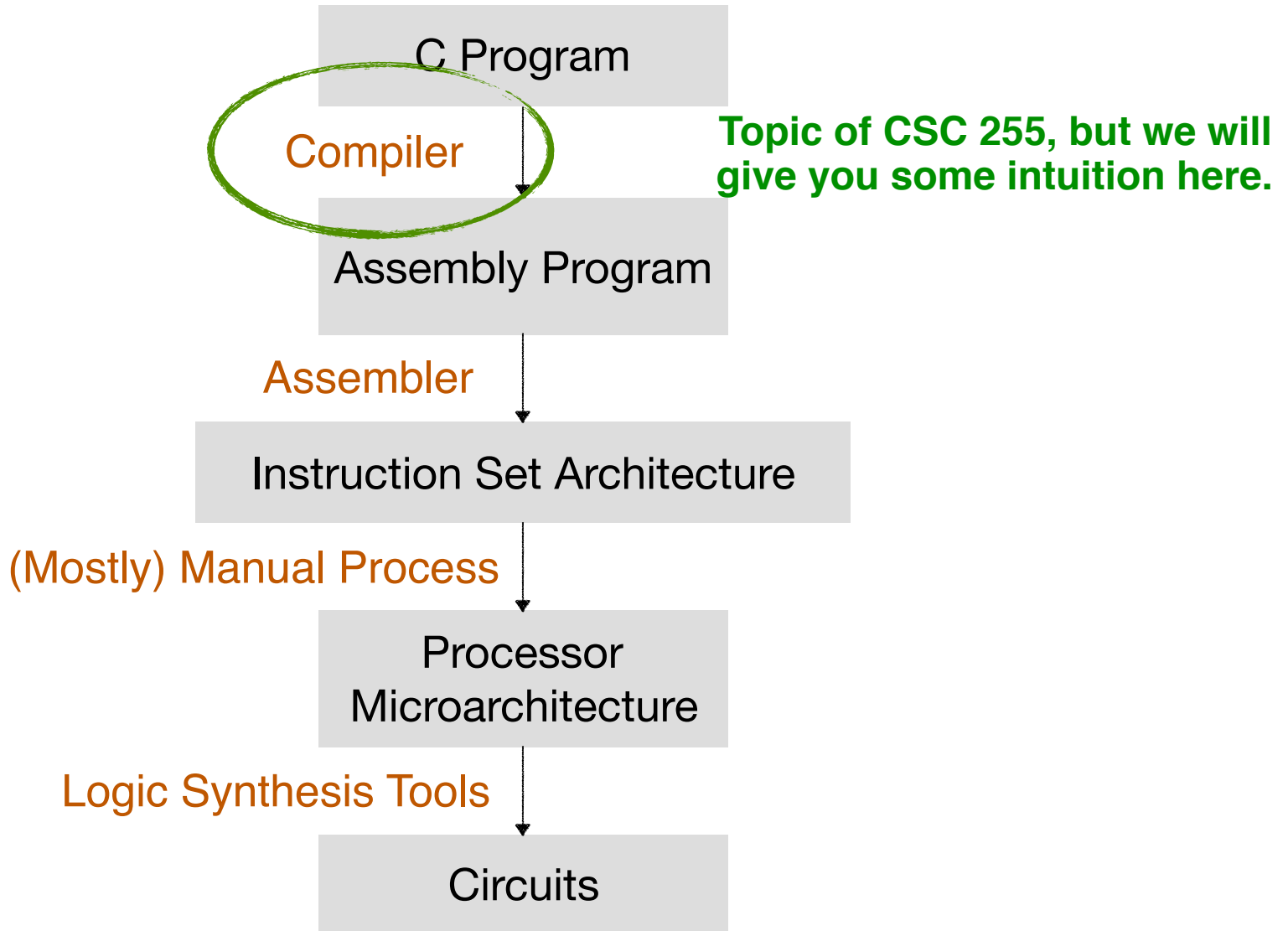- "**I don't know**" gets15% partial credit. Must erase everything else.

Announcements

- Mock exam now. Won't be graded.

- Go to https://www.gradescope.com/courses/256242/assignments/1116540.

- Log in using your NetID.

So far in 252…

C Program

Compiler

Assembly Program

Assembler

Instruction Set Architecture

(Mostly) Manual Process

Processor
Microarchitecture

Logic Synthesis Tools

Circuits

So far in 252…

C Program

Compiler

Topic of CSC 255, but we will give you some intuition here.

Assembly Program

Assembler

Instruction Set Architecture

(Mostly) Manual Process

Processor Microarchitecture

Logic Synthesis Tools

Circuits

Code Optimization Overview

- Three entities can optimize the program: programmer, compiler, and hardware

- The best thing to speed up a program is to pick a good algorithm. Compilers/hardware can't do that in general.
  - Quicksort: $O(n \log n) = K * n * \log(n)$
  - Bubblesort: $O(n^2) = K * n^2$

- Algorithm choice decides overall complexity (big O), compiler/hardware decides the constant factor in the big O notation

- Compiler and hardware implementations decide the K.

- Programmers can write code that makes it easier to compiler and hardware to improve performance.

Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
    long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

⟶

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
    long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
long j;
int ni = n*i;
for (j = 0; j < n; j++)
    a[ni+j] = b[j];
```

```
set_row:
        testq      %rcx, %rcx               # Test n
        jle        .L1                      # If 0, goto done
        imulq      %rcx, %rdx               # ni = n*i
        leaq       (%rdi,%rdx,8), %rdx # rowp = A + ni*8
        movl       $0, %eax                 # j = 0
.L3:                                        # loop:
        movsd      (%rsi,%rax,8), %xmm0     # t = b[j]
        movsd      %xmm0, (%rdx,%rax,8)     # M[A+ni*8 + j*8] = t
        addq       $1, %rax                 # j++
        cmpq       %rcx, %rax               # j:n
        jne        .L3                      # if !=, goto loop
.L1:                                        # done:
        rep ; ret
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
    - `16*x`   `-->`   `x << 4`
    - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles. Division takes even more cycles. Shift can generally be done in 1 cycle.
- Use the `lea` instruction

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  - `16*x        -->      x << 4`
  - Depends on cost of multiply or divide instruction
  - On Intel Nehalem, integer multiply requires 3 CPU cycles. Division takes even more cycles. Shift can generally be done in 1 cycle.
- Use the `lea` instruction

```
long m12(long x)
{
  return x*12;
}
```

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Common Subexpression Elimination

- Reuse portions of expressions
- GCC will do this with –O1

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

```
leaq   1(%rsi), %rax   # i+1
leaq   -1(%rsi), %r8   # i-1
imulq  %rcx, %rsi      # i*n
imulq  %rcx, %rax      # (i+1)*n
imulq  %rcx, %r8       # (i-1)*n
addq   %rdx, %rsi      # i*n+j
addq   %rdx, %rax      # (i+1)*n+j
addq   %rdx, %r8       # (i-1)*n+j
```

```
imulq   %rcx, %rsi  # i*n
addq    %rdx, %rsi  # i*n+j
movq    %rsi, %rax  # i*n+j
subq    %rcx, %rax  # i*n+j-n
leaq    (%rsi,%rcx), %rcx # i*n+j+n
```

Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- Exploit Hardware Microarchitecture

Optimization Blocker #1: Procedure Calls

- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Calling Strlen

```c
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

- Strlen performance
  - Has to scan the entire length of a string, looking for null character.
  - O(N) complexity
- Overall performance
  - N calls to strlen
  - Overall O(N^2) performance

Improving Performance

- Move call to `strlen` outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}


size_t total_lencount = 0;
size_t strlen(const char *s)
{

    size_t length = 0;
    while (*s != '\0') {
        s++; length++;
    }
    total_lencount += length;
    return length;
}
```

Why couldn't compiler move `strlen` out of loop?

- Procedure may have side effects, e.g., alters global state each time called
- Function may not return same value for given arguments

Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.

Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.
- Remedies:
  - Use of inline functions
  - Do your own code motion

Optimization Blocker: Procedure Calls

- Most compilers treat procedure call as a black box
  - Assume the worst case, weak optimizations near them
  - There are interprocedural optimizations (IPO), but they are expensive
  - Sometimes the compiler doesn't have access to source code of other functions because they are object files in a library. Link-time optimizations (LTO) comes into play, but are expensive as well.

- Remedies:
  - Use of inline functions
  - Do your own code motion

```
inline void swap(int *m, int *n) {
 int tmp = *m;
 *m = *n;
 *n = tmp;
}


void foo () {
  swap(&x, &y);
}
```
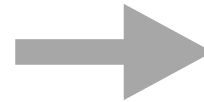
```
void foo () {
  int tmp = x;
  x = y;
  y = tmp;
}
```

Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

Value of b:

```
init:  [x, x, x]
```

Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,   1,   2,
    4,   8,  16,
   32,  64, 128};
```

Value of b:

```
init:  [x, x, x]
```

```
i = 0: [3, x, x]
```

Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    4,    8,   16,
   32,   64,  128};
```

Value of b:

```
init:   [x, x, x]
```

```
i = 0: [3, x, x]
```

```
i = 1: [3, 28, x]
```

Optimization Blocker #2: Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    4,    8,   16,
   32,   64,  128};
```

Value of b:

```
init:  [x, x, x]
```

```
i = 0: [3, x, x]
```

```
i = 1: [3, 28, x]
```

```
i = 2: [3, 28, 224]
```

A Potential Optimization

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]. Memory accesses are slow, so…

A Potential Optimization

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]. Memory accesses are slow, so…

```
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
```

Every iteration updates `val`, which could stay in register. Update memory only once.

A Potential Optimization

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Every iteration updates memory location b[i]. Memory accesses are slow, so…

```
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
```

Every iteration updates `val`, which could stay in register. Update memory only once.

Why can't a compiler perform this optimization?

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    4,    8,   16,
   32,   64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b ⟶

Value of b:

```
init:  [4, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    4,    8,   16,
   32,   64,  128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b →

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,   1,   2,
    3,   8,  16,
   32,  64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b →

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    3,    0,   16,
   32,   64,  128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b →

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,   1,   2,
    3,   3,  16,
   32,  64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b →

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    3 ,    6 ,   16,
   32,   64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,    1,    2,
    3,   22    16,
   32,   64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

b →

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

b →

```
double a[9] =
  { 0,   1,   2,
    3,  22   16,
   32,  64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

Memory Aliasing

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

Value of a:

```
double a[9] =
  { 0,   1,   2,
b   3,   22   16,
   32,  64, 128};

double b[3] = a+3;

sum_rows1(a, b, 3);
```

Value of b:

```
init:  [4, 8, 16]
```

```
i = 0: [3, 8, 16]
```

```
i = 1: [3, 22, 16]
```

```
i = 2: [3, 22, 224]
```

Optimization Blocker: Memory Aliasing

- Aliasing
  - Two different memory references (array elements or pointers) specify the same memory location
  - Easy to have in C
    - Since C allows address/pointer arithmetic
    - Direct access to storage structures
  - Get in habit of introducing local variables
    - Accumulating within loops
    - Your way of telling compiler not to check for aliasing

Today: Optimizing Code Transformation

- Hardware/Microarchitecture Independent Optimizations
  - Code motion/precomputation
  - Strength reduction
  - Sharing of common subexpressions
- Optimization Blockers
  - Procedure calls
  - Memory aliasing
- **Exploit Hardware Microarchitecture**

Exploiting Instruction-Level Parallelism (ILP)

- Hardware can execute multiple instructions in parallel
  - Pipeline is a classic technique. Multiple instructions are being executed at the same time
- Performance limited by control/data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Baseline Code

```
for (i = 0; i < length; i++) {
  t = t * d[i];
  *dest = t;
}
```

```
.L519:
  imulq (%rax,%rdx,4), %ecx
  addq  $1, %rdx          # i++
  cmpq  %rdx, %rbp        # Compare length:i
  jg    .L519            # If >, goto Loop
```

⟵ Real work

⟵ Overhead

Loop Unrolling (2x1)

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x = (x * d[i]) * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x = x * d[i];
}
*dest = x;
```

- Perform 2x more useful work per iteration
- Reduce loop overhead (comp, jmp, index dec, etc.)
- What's the trade-off here?

Loop Unrolling with Separate Accumulators

```
long limit = length-1;
long i;
/* Combine 2 elements at a time */
for (i = 0; i < limit; i+=2) {
    x0 = x0 * d[i];
    x1 = x1 * d[i+1];
}

/* Finish any remaining elements */
for (; i < length; i++) {
    x0 = x0 * d[i];
}
*dest = x0 * x1;
```

Data-Flow Graph (DFG)

```
x0 = x0 * d[i];
x1 = x1 * d[i+1];
```

- What changed:
  - Two independent "streams" of operations
  - Reduce data dependency
- What was the DFG like using just one accumulator?
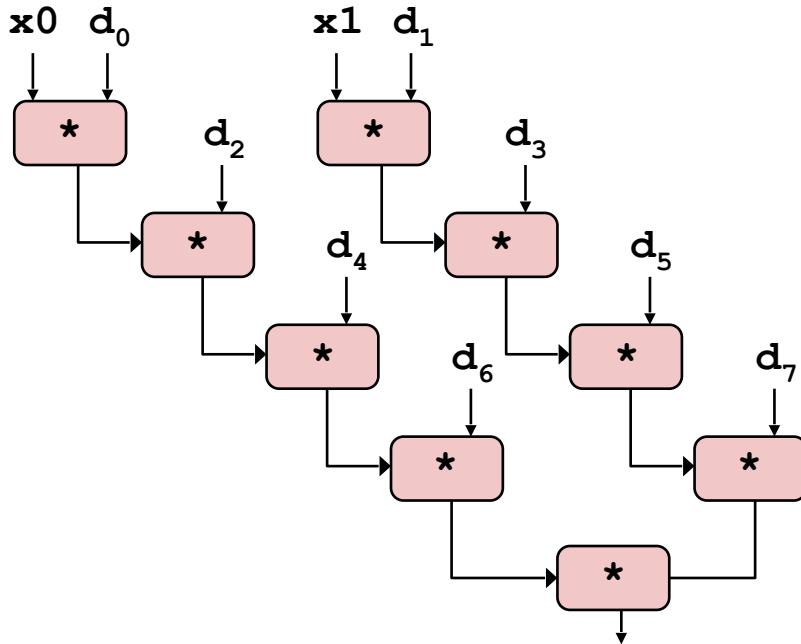
Code Optimization Summary

- From a programmer's perspective:
  - What you know: the functionality/intention of your code; the inputs to the program; all the code in the program
  - What you might not know: the hardware details.
- From a compiler's perspective:
  - What you know: all the code in the program; (maybe) the hardware details.
  - What you might not know: the inputs to the program; the intention of the code
- From the hardware's perspective:
  - What you know: the hardware details; some part of the code
  - What you might not know: the inputs to the program; the intention of the code
- The different perspectives indicate that different entities have different responsibilities, limitations, and advantages in optimizing the code

Aside: Profile-Guided Optimization

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except…Profile-guided optimizations:

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except…Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of x and y (statistically).

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except…Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of x and y (statistically).
  - If let's say 99% of the time, x = 2 and y = 5, what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

Aside: Profile-Guided Optimization

- As a programmer, if you know what x and y will be, say 5, you could direct return the results 23769.8 without having to the computation
- Compiler would have no idea
- Except…Profile-guided optimizations:
  - Run the code multiple times using some sample inputs, and observe the values of x and y (statistically).
  - If let's say 99% of the time, x = 2 and y = 5, what could the compiler do then?

```
float foo(int x, int y)
{
    return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

```
float foo(int x, int y)
{
  if (x == 2 && y == 5) return 23769.8;
  else return pow(x, y) * 100 / log(x) * sqrt(y);
}
```

So far in 252…
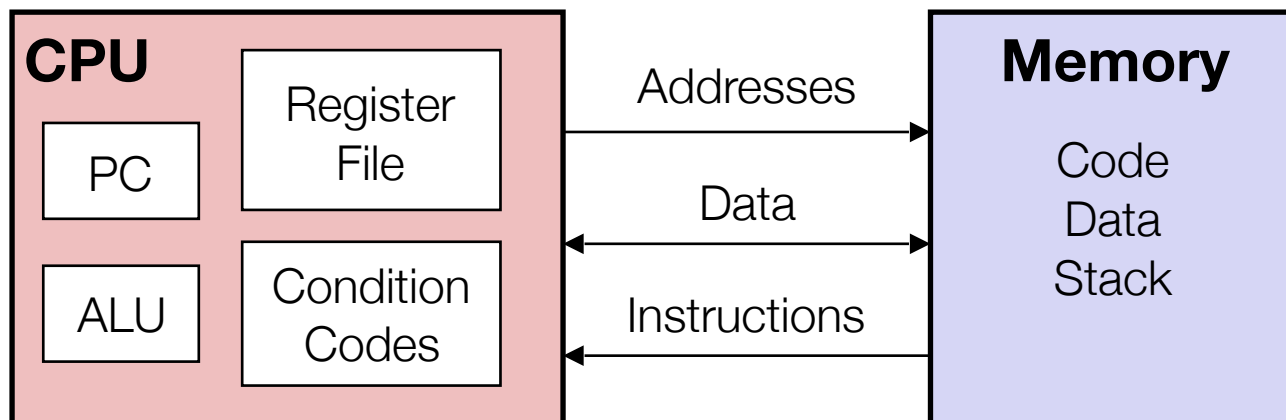
- We have been discussing the CPU microarchitecture
  - Single Cycle, sequential implementation
  - Pipeline implementation
  - Resolving data dependency and control dependency
- What about memory?

Memory in a Modern System
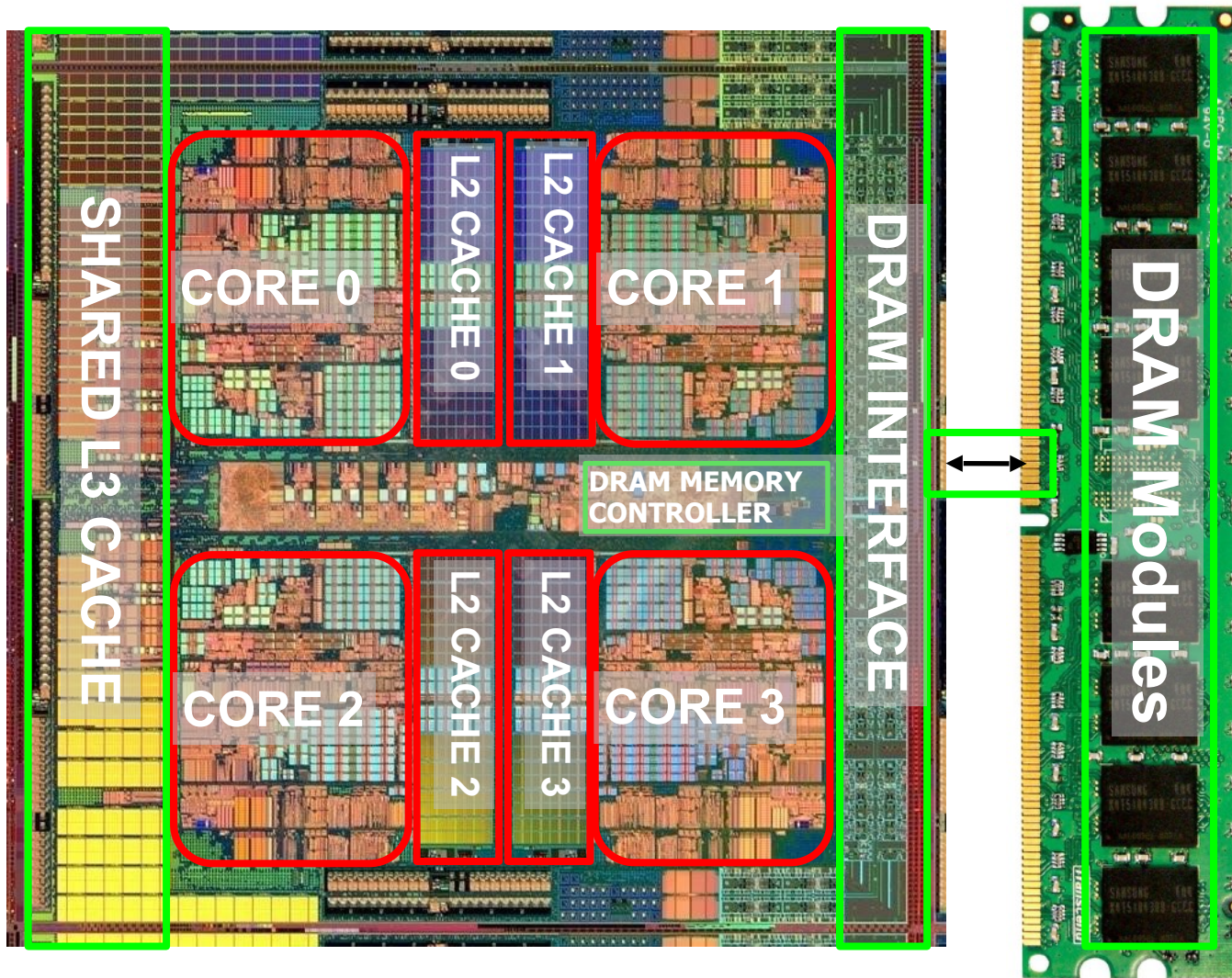
Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger $\rightarrow$ Takes longer to determine the location

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger ➔ Takes longer to determine the location

- Faster is more expensive

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger ➔ Takes longer to determine the location

- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger ➔ Takes longer to determine the location

- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape

- Higher bandwidth is more expensive

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger ➔ Takes longer to determine the location

- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

Memory Technology: D Flip-Flop (DFF)
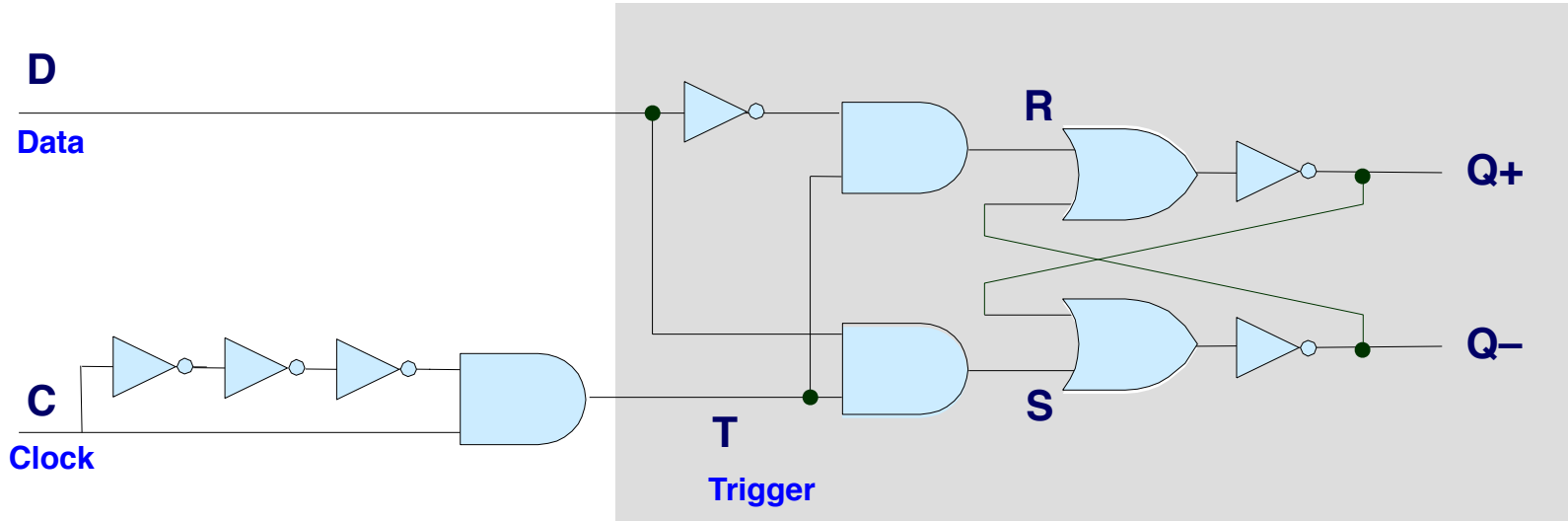
- Very fast
- Very expensive to build
  - 6 NOT gates (2 transistors / gate)
  - 3 AND gates (3 transistors / gate)
  - 2 OR gates (3 transistors / gate)
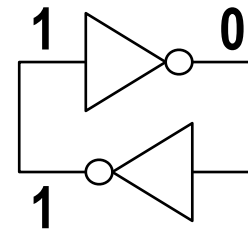 - 27 transistors in total for just one bit!!
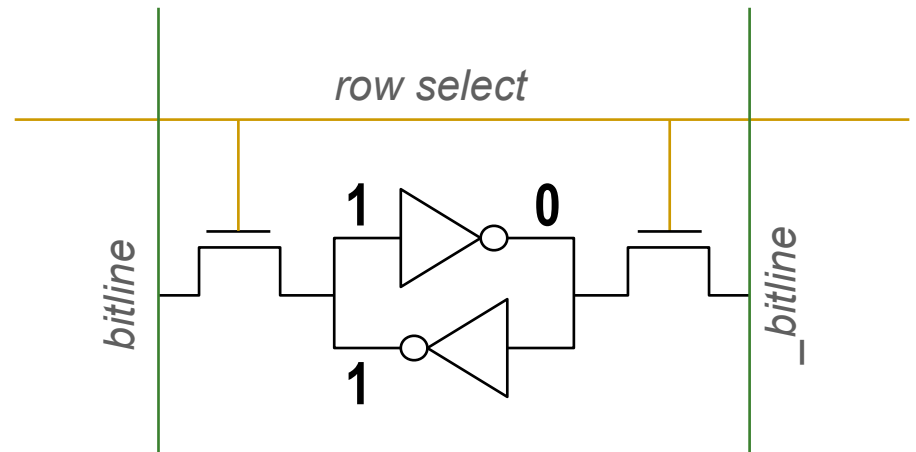
Memory Technology: SRAM

Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
    - Feedback path enables the stored value to persist in the "cell"
    - 4 transistors for storage
    - 2 transistors for access
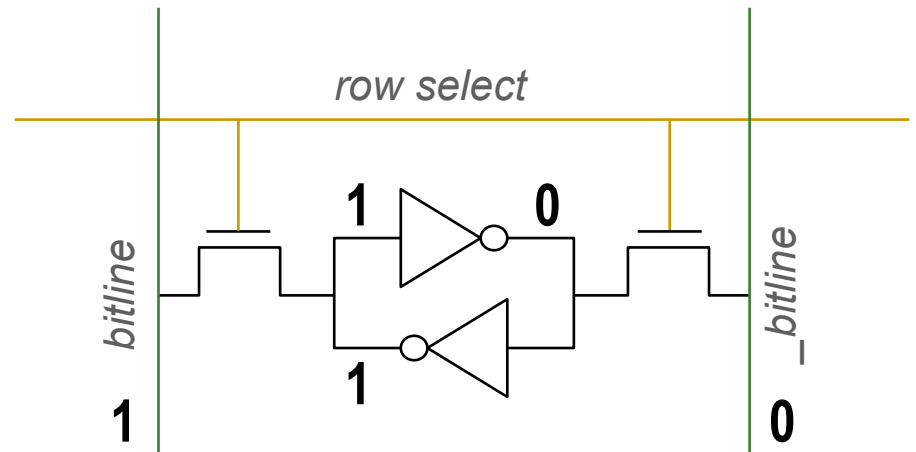   - 6 transistors in total per bit

Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
    - Feedback path enables the stored value to persist in the "cell"
    - 4 transistors for storage
    - 2 transistors for access
    - 6 transistors in total per bit

Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
  - 4 transistors for storage
  - 2 transistors for access
 - 6 transistors in total per bit
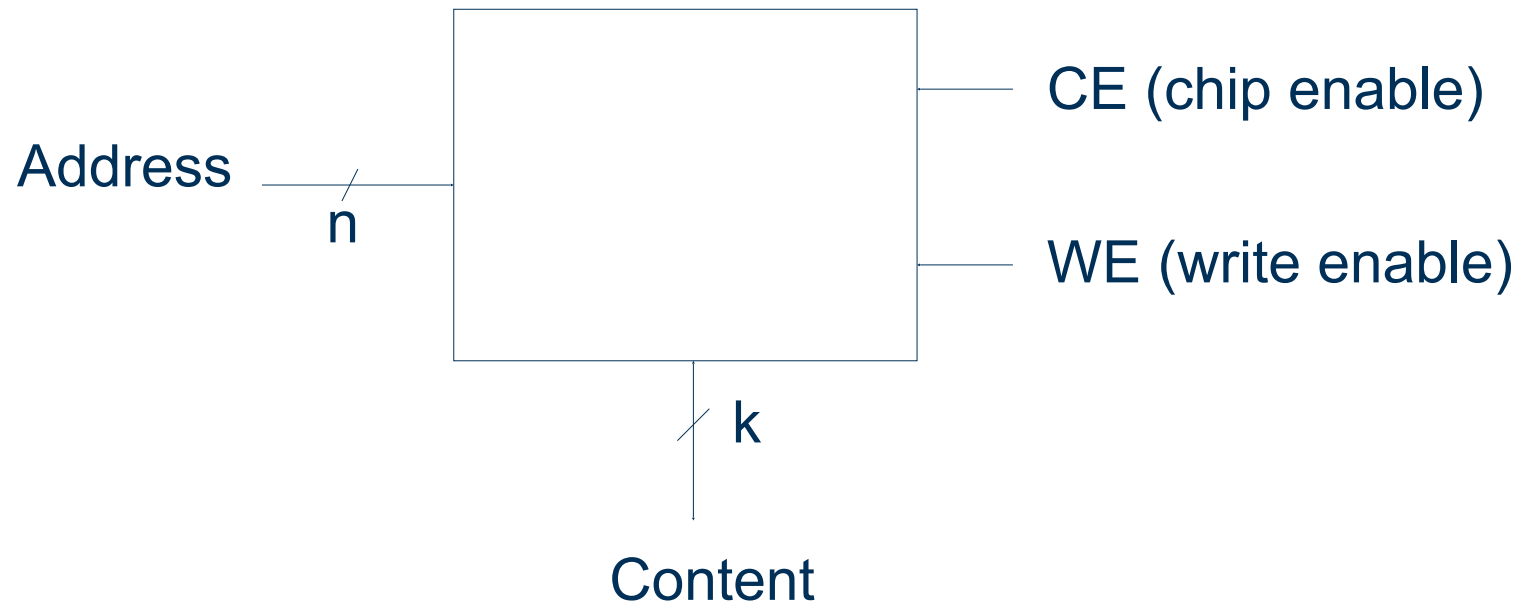
Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
  - 4 transistors for storage
  - 2 transistors for access
  - 6 transistors in total per bit

Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
  - 4 transistors for storage
  - 2 transistors for access
  - 6 transistors in total per bit

Memory Technology: SRAM

- Static random access memory

- Random access means you can supply an arbitrary address to the memory and get a value back

- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the "cell"
  - 4 transistors for storage
  - 2 transistors for access
  - 6 transistors in total per bit
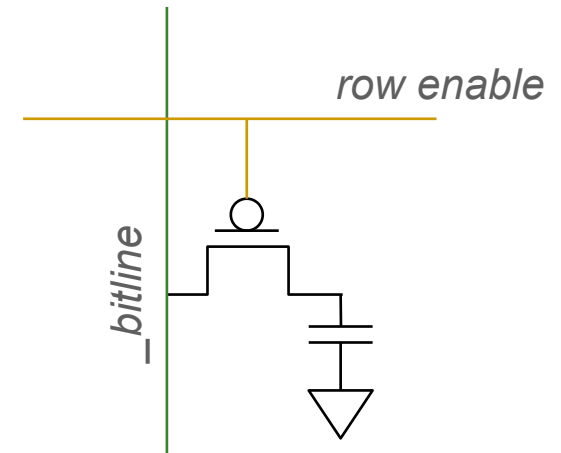
SRAM Array
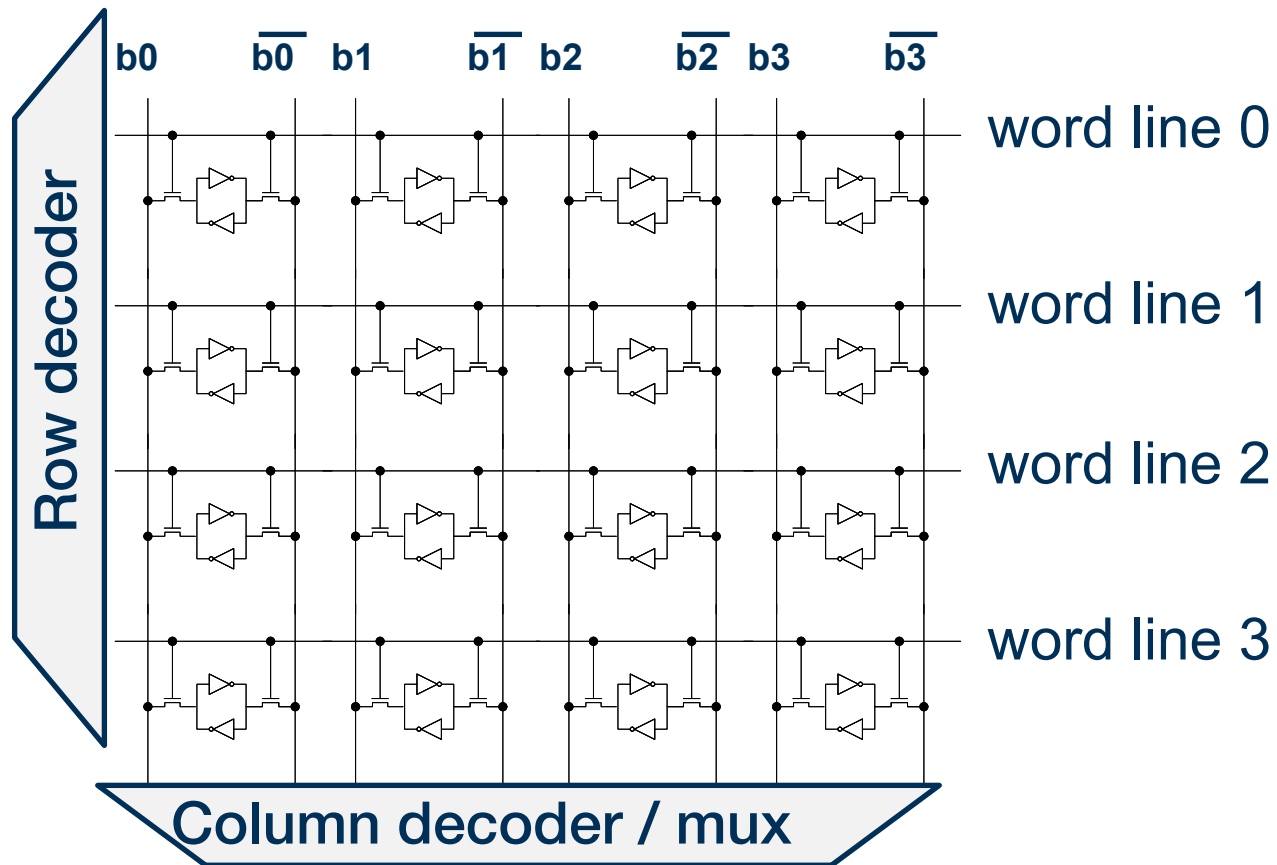
Abstract View of SRAM

Address —— /n —— [ ] —— CE (chip enable)
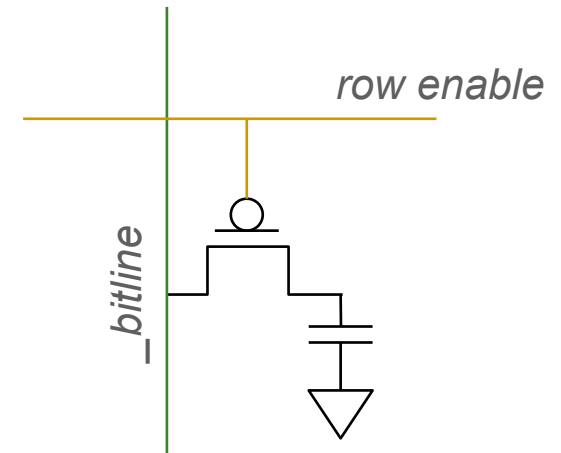
[ ] —— WE (write enable)

/k Content

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
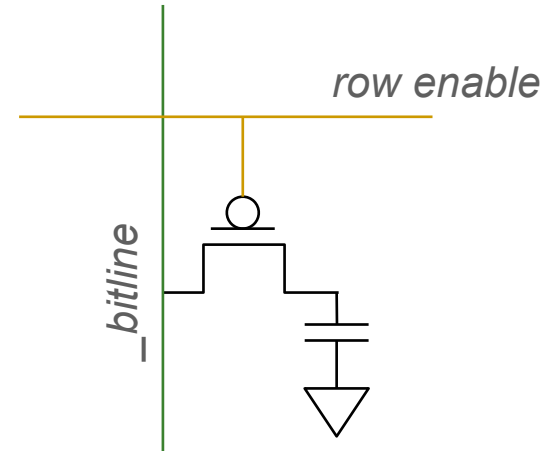 - 1 capacitor

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitors will leak!

row enable

_bitline

Memory Technology: DRAM

- Dynamic random access memory

- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

- Capacitors will leak!
  - DRAM cell loses charge over time

row enable

_bitline

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed.**
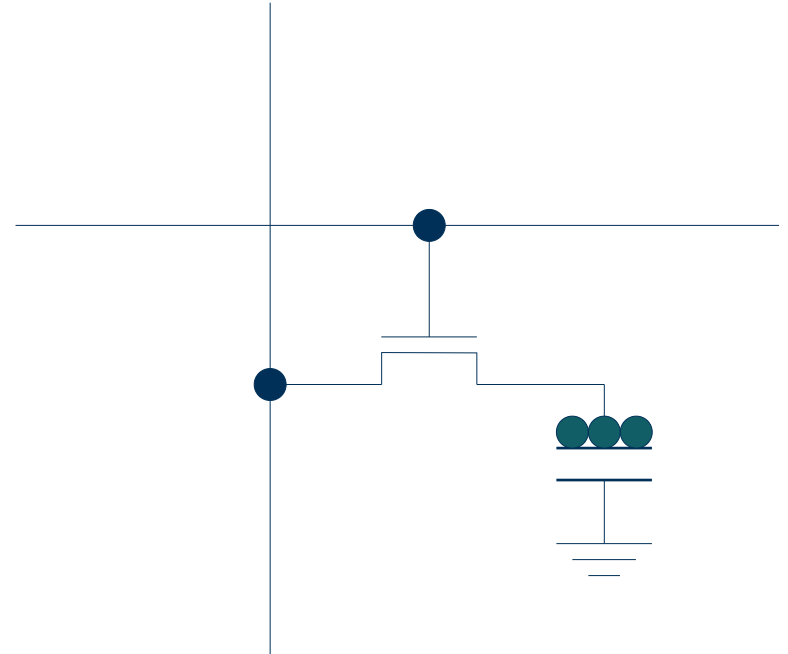
row enable

_bitline

Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor
- Capacitors will leak!
  - DRAM cell loses charge over time
  - DRAM cell needs to be **refreshed.**
  - Refresh takes time and power. When refreshing can't read the data. A major issue, lots of research going on to reduce the refresh overhead.
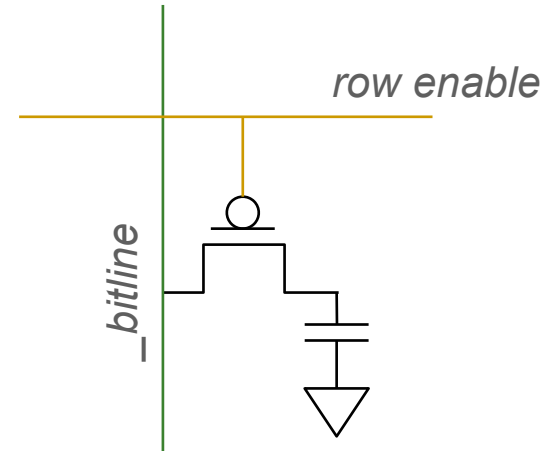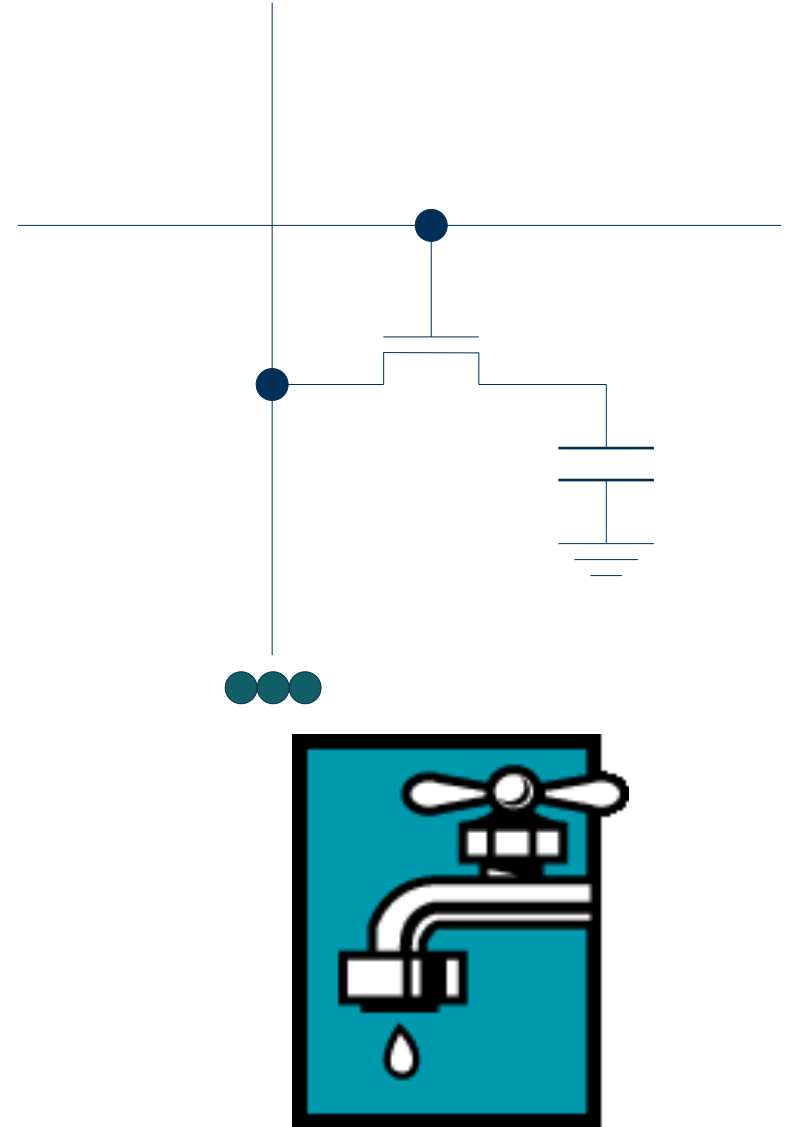
row enable

_bitline

DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)

- How do you maintain the values in DRAM?
  - Refresh periodically
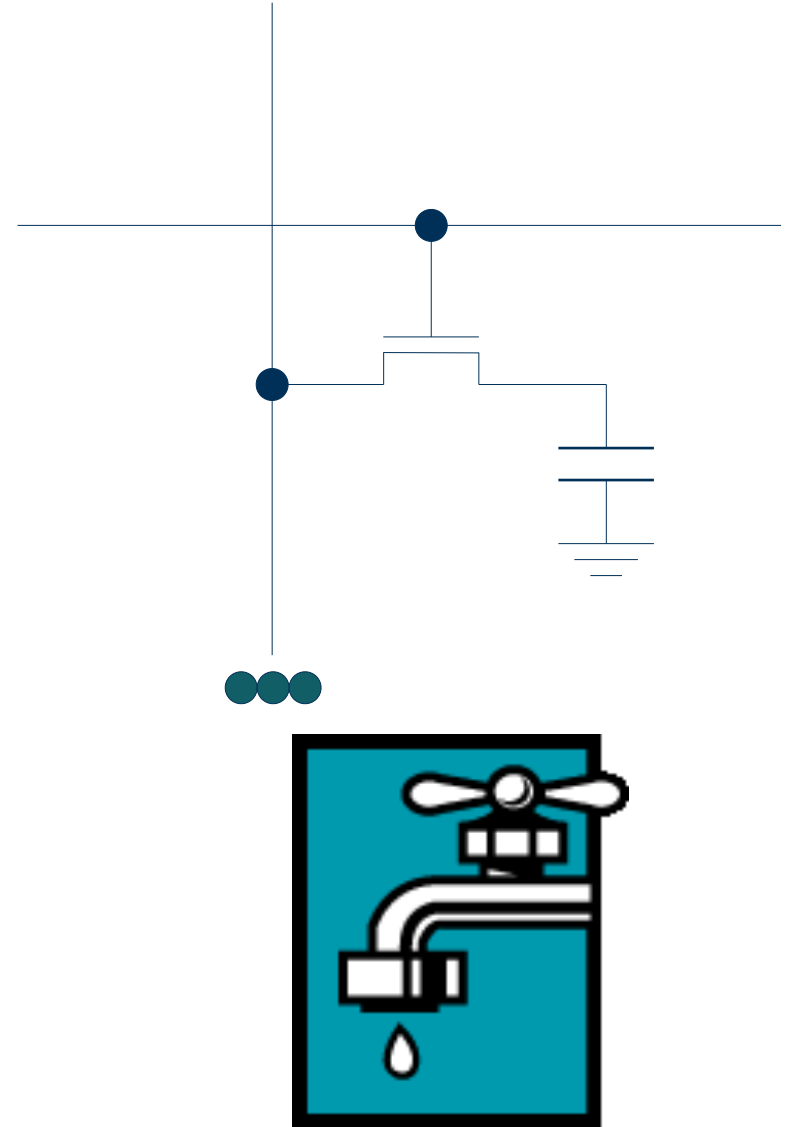 - A major source for power consumption in DRAM

DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)

- How do you maintain the values in DRAM?
  - Refresh periodically
  - A major source for power consumption in DRAM

DRAM Cell

- Capacitor holding value leaks, eventually you will lose information (everything turns to 0)

- How do you maintain the values in DRAM?
  - Refresh periodically
  - A major source for power consumption in DRAM

Latch vs. DRAM vs. SRAM

- DFF
  - Fastest
  - Low density (27 transistors per bit)
  - High cost
- SRAM
  - Faster access (no capacitor)
  - Lower density (6 transistors per bit)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)
- DRAM
  - Slower access (capacitor)
  - Higher density (1 transistor + 1 capacitor per bit)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

Nonvolatile Memories

Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.

Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)

Nonvolatile Memories

- DFF, DRAM and SRAM
  - Lose information if pow
- Nonvolatile memories re
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)

Rewriting Life

Microsoft Has a Plan to Add DNA Data Storage to Its Cloud

Tech companies think biology may solve a looming data storage problem.

by Antonio Regalado   May 22, 2017

Based on early research involving the storage of movies and documents in DNA, Microsoft is developing an apparatus that uses biology to replace tape drives, researchers at the company say.

Computer architects at Microsoft Research say the company has formalized a goal of having an operational storage system based on DNA

Nonvolatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)
  - DNA (centuries)
- Uses for Nonvolatile Memories
  - Firmware (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems,…)
  - Files in Smartphones, mp3 players, tablets, laptops
  - Backup