CS61 Section Notes With Solutions
Week 4 (Fall 2011)

Outline for this week:

Processor Architecture
- Logic gates
- Instruction encoding
- Sequential processing with stages
- Pipelining

Program Optimization
- Code motion
- Memory accesses
- Loop unrolling
- Pipelining
- Demo of efficiency gains

All page numbers cited below refer to the course textbook, *Computer Systems: A Programmer's Perspective, Second Edition* by Randal E. Bryant and David R. O'Hallaron.

Processor Architecture

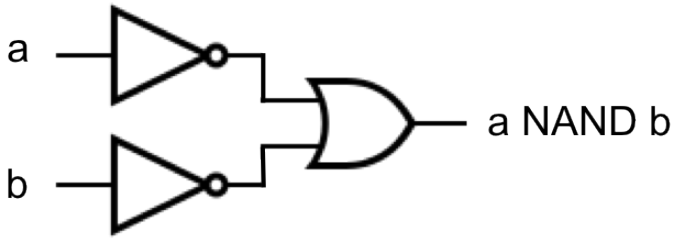
1. Logic gates

AND **OR** **NOT**

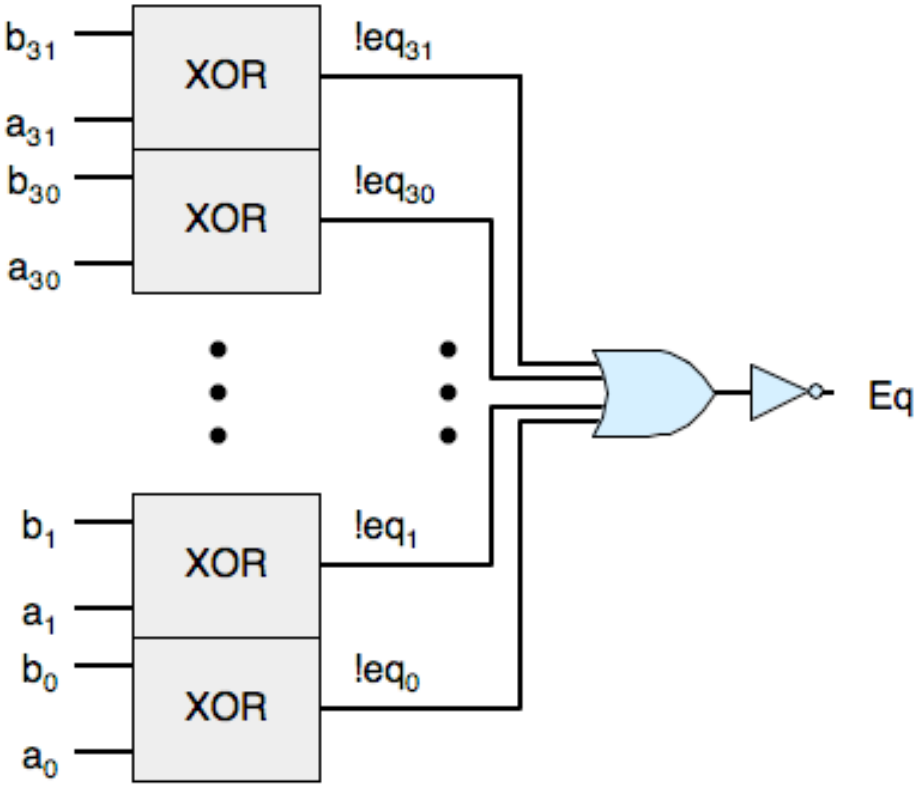
Warm up: Fill in the truth table below for NAND.
To the right, design a circuit for bit-level NAND using OR and NOT gates.

We implement NAND using a circuit equivalent to the expression !a || !b.

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0



In class, we introduced a circuit for XOR. Suppose you want to implement a word-level equality circuit using XOR rather than from bit-level equality circuits. Design such a circuit for a 32-bit word consisting of 32 bit-level XOR circuits and two additional logic gates.[1] (Assume that you can have up to 32 inputs to AND and OR gates.)



What determines the length of time to evaluate a circuit?

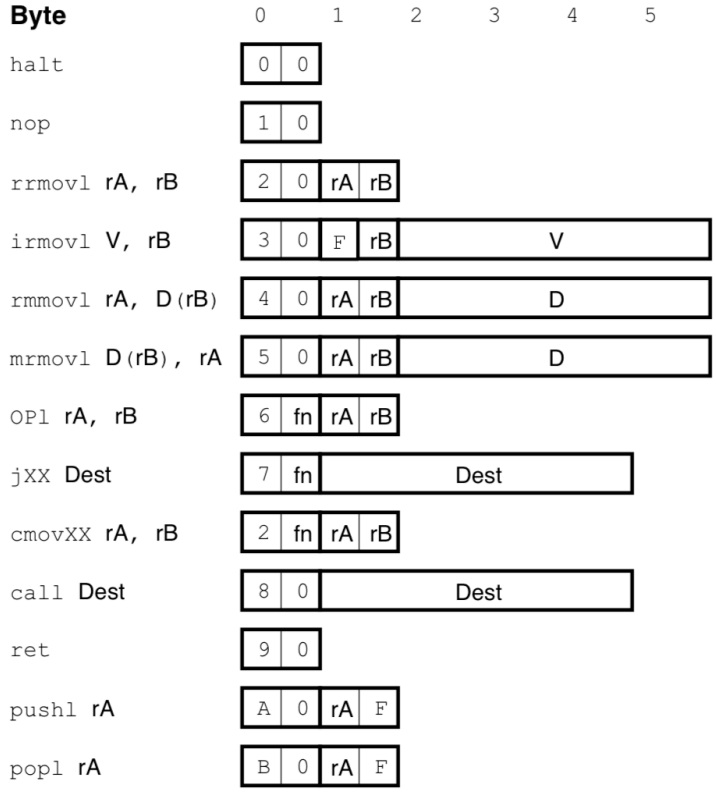
The time it takes for a signal to propagate through a logic gate is much greater than the time it takes to propagate through the wire between gates. Thus, the **timing specification** of a combinational circuit is approximately the time for a single logic gate to produce stable output from stable input signals, multiplied by the **circuit depth** (i.e., the maximum number of logic gates on any path from input to output).

[1]Adapted from Practice Problem 4.9, p. 356.

2. Instruction encoding

Consider the instruction set, register identifiers, and function codes depicted below. These are from the Y86 instruction set architecture introduced in the book. It is NOT important to memorize the details of Y86, which is inspired by IA32 but simpler and reduced in design. The point of the following exercises is simply to get a taste of instruction encoding and decoding.

Instruction set.



Register identifiers.

Number	Register
0	%eax
1	%ecx
2	%edx
3	%ebx
4	%esp
5	%ebp
6	%esi
7	%edi
F	No register

Function codes.

Operation	Code
addl	60
subl	61
andl	62
xorl	63

Branches	Code
jmp	70
jle	71
jl	72
je	73
jne	74
jge	75
jg	76

Moves	Code
rrmovl	20
cmovle	21
cmovl	22
cmove	23
cmovne	24
cmovge	25
cmovg	26

Determine the byte encoding of the following instruction sequence. Using the byte encodings that you compute, also determine the addresses of the instructions below, as well as a hypothetical next instruction. [2] **The first instruction (`irmovl $15, %ebx`) starts at address `0x100`.**

```
0x100 : 30f3 0f00 0000              | irmovl $15, %ebx  # Load 15 into %ebx

0x106 : 2031                        | rrmovl %ebx, %ecx # Copy 15 into %ecx

0x108 : next_encoding               | next_operation
```

Some features of the encoding are worth noting:

(1) Decimal 15 in hex is `0x0000000f`. Writing the bytes in reverse order gives 0f 00 00 00.

(2) The code starts at address `0x100`. The first instruction requires 6 bytes, while the second requires 2, so the next instruction will be at `0x108`.

For the byte sequence below, determine the instruction sequence it encodes. We show the starting address, then a colon, then the byte sequence.[3]

`0x400: 6113 7300 0400 0000`

The above encoding contains a jump operation:

```
0x400:                | loop:
0x400: 6113           |     subl %ecx, %ebx
0x402: 7300 0400 00   |     je loop
0x407: 00             |     halt
```

[2] Adapted from Practice Problem 4.1, p. 341.
[3] Adapted from Practice Problem 4.2, p. 341.

3. Sequential processing with stages[4]

Consider a sequential processor that executes in stages. Some of the hardware units for the stages have state that may be updated during the execution of an instruction. Indicate which of the following hardware units hold state that might be updated:

Program counter (PC)	_____	**Arithmetic logic unit (ALU)**	_____
Condition code register (CC)	_____	**Data memory**	_____
Register file	_____	**Instruction memory**	_____

The PC, CC, register file, data memory are all state elements. Computations propagate through the ALU but do not set any state. Instruction memory is only used to read instructions.

The sequential processor introduced in class and the textbook breaks the processing of a single instruction into six stages: Fetch, Decode, Execute, Memory, Write back, and PC update. Consider the processing of a single instruction. Do the stages update state in a serial fashion, i.e. do the stages update state in some order? When do these updates occur?

No, all state updates occur simultaneously, and only when the clock rises to start the processing of the next instruction. (p. 380)

Principle: The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction. For example, some instructions (integer operations) set condition codes, and some instructions (jumps) read condition codes, but no instruction both sets and reads condition codes. This way, the condition codes are always up-to-date before another instruction needs to read them. (p. 380-1)

4. Pipelining

The latency of a stage is the time required to execute that stage. Throughput is the rate at which stages are executed, and so latency is equal to the inverse of throughput. E.g., If a car wash has a throughput of 2 cars per minute, its latency is ½ minutes = 30 seconds.

Suppose a pipelined processor executes N stages in parallel. Label the stages $s_1, s_2, \ldots, s_N$ such that $t_1 < t_2 < \ldots < t_N$, where t_i is the latency of stage i. What is the throughput of the processor when executing all N stages in parallel?

The throughput is limited by the slowest stage. This is an important concept to keep in mind when thinking about parallelization in general. The slowest stage has latency t_N. Since throughput = 1 / latency, the overall throughput is 1 / t_N.

[4]Material and language from Section 4.3.

Pipelining divides a computation into separate stages separated by pipeline registers. What happens as we divide the computation into increasingly shorter stages?

Deeper pipelining yields diminishing returns as throughput becomes limited by the time required to load pipeline registers.

Program Optimization

Ideally, a compiler should be able to take any code and generate the most efficient machine-level program with the correct behavior. In reality, compilers can only perform limited transformations like code motion, strength reduction, loop unrolling, etc. Even these transformations can be thwarted by *optimization blockers* – aspects of of the program's behavior that depend strongly on the execution environment.

Consider the following C code...

```
struct vector {
      int length;
      int *data;
}

typedef struct vector *vec_ptr;
int get_vec_length(vec_ptr v);
int *get_vec_data(vec_ptr v);

/* Multiply the elements of a vector. */
void multiply1(vec_ptr v, int *dest) {
      int i;
      int *data = get_vec_data(v);

      *dest = 1;
      for (i = 0; i < get_vec_length(v); i++) {
            *dest = *dest * data[i];
      }
}
```

In this section, we will consider what optimizations are possible here.

1. Code Motion - Move code around to reduce the number of times it executes.

The call `get_vec_length(v)` can be moved outside the loop, because we know the length of the vector won't change from one iteration to the next.

```
void multiply2(vec_ptr v, int *dest) {
      int i;
      int *data = get_vec_data(v);
      int length = get_vec_length(v);

      *dest = 1;
      for (i = 0; i < length; i++) {
            *dest = *dest * data[i];
      }
}
```

Could `gcc -01` make this optimization? If not, why not, and could we change anything to help?

`gcc -01` cannot make this optimization because optimization is typically within-function; from within `multiply`, the procedure call to `get_vec_length` appears as a "black box" with unknown properties. If `get_vec_length` were declared as `inline`, then the compiler would copy its code into `multiply` and presumably notice this optimization. At higher levels of optimization such as `gcc -02`, the compiler attempts to aggressively `inline` functions.

2. Reduce Memory Access

Introduce local variable `x` to store intermediate computations, and copy the result at the end to `*dest`.

```
void multiply3(vec_ptr v, int *dest) {
      int i;
      int *data = get_vec_start(v);
      int length = vec_length(v);
      int x = 1;

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

Why is this an optimization?

Memory accesses are slow! Storing things locally when possible can give big efficiency improvements.

Could `gcc` make this optimization? If not, why not?

`gcc` cannot make this optimization because it changes the meaning of the code. If `dest` points somewhere inside `data`, then `multiply3` may produce a different result than `multiply1`. It is always important to consider the scenario in which pointers *alias* the same data.

3. Loop Unrolling - Decrease the number of iterations by doing more work per iteration.

Consider the following implementation of `multiply` (assume `length % 2 == 0`).

```
void multiply4(vec_ptr v, int *dest) {
      int i;
      int *data = get_vec_start(v);
      int length = vec_length(v);
      int x = 1;
      int y = 1;

      for (i = 0; i < length; i += 2) {
            x = x * data[i];
            y = y * data[i + 1];
      }
      *dest = x * y;
}
```

This implementation is even more efficient! Why? (There are two principal reasons.)

(1) Loop unrolling reduces the amount of per-iteration overhead of managing the loop, such as the `jmp` instruction.

(2) It also increases *processor pipeline parallelism*. In `multiply3`, each multiplication of `x` depends on the previous multiplication, so the next multiplication cannot start until the previous one has finished. In `multiply4`, `x` and `y` are independent registers and can be pipelined simultaneously.

Wouldn't it be even better to unroll the loop more, say to `i += 20`?

Loop unrolling has significant diminishing returns: in particular, an x86 only has 8 registers. If our unrolled loop has to use memory to store values, it becomes much less efficient.

A Practical Note - Loop unrolling and taking advantage of pipelining are examples of optimizations that can be very hard for human programmers to implement effectively in practice. Additionally, we usually can rely on our compiler to do these types of optimizations for us!

Interactive Optimization Demo

At this point, please open up your laptops and follow the instructions below, substituting your username for user where appropriate. First, ssh to your CS 61 VM. Copy the file from Prof Chong's home directory to your own:

```
$ cp ~stephenchong/optimization.tar ~
```

You should now have a file called optimization.tar in your home directory. Let's extract the files in the tar.

```
$ tar xvf optimization.tar
$ cd optimization
```

This code is a function that implements a simple substitution cipher. It's pretty silly code, and there are dummy functions intended to let you practice various optimizations in the file. Get in small groups and discuss the types of optimizations you might implement!

Take a look at the file `benchmark.c`, in particular the function `run_benchmarks()` at the bottom of the file. As you can see, this function benchmarks a series of implementations of the substitution cipher: `cipher_orig`, `cipher_better`, `cipher_faster`, and `cipher_fast`. Let's run the benchmarks.

To run the benchmarks without compiler optimization (i.e., with `-O0`), type:

```
$ make run
```

To run the benchmarks with compiler optimization (i.e., with `-O3`), type:

```
$ make run-opt
```

Initially, all four of the cipher functions do the same thing, so running it should give very similar results for all four trials.

Take a look at the code for `cipher_orig`. Do not edit this function: it is used to provide a baseline for the improvements you'll be making. You'll be editing the functions: `cipher_better`, `encode_char_better`, `cipher_faster`, and `cipher_fast` (in that order). Follow the instructions in the comments in `benchmark.c` and from your TF. Feel free to look in `support.c` to inspect the main driver routine, including timing, etc.

When you're done implementing your optimizations, come back together and discuss what you tried, what worked, what didn't, and what things the compiler optimizations took care of for you.

A couple of final take-away points:

- Readability/maintainability is often more important than minor optimizations.
- The compiler is often smarter than you are.
- Low-level things like loop unrolling are often architecture-dependent, and should be done only with extreme caution! Nevertheless, it's good to understand how the CPU works.
- Be sure you know what you're optimizing and why before you start. It's easy to waste days only to realize that all your hard work was unimportant in the grander scheme of things.
- Testing is even more key. It doesn't matter how fast your code runs if it doesn't work!

EXTRA BONUS SECTION

Program Optimization (continued)

4. Pipelining - Take advantage of how the machine pipelines instructions.

Pipelining, the process of starting the next instruction before the previous one has finished, can be extremely useful, especially when you're looking at expensive operations like multiplication, but it won't work if the operations need to be executed in sequence.

Consider the following C code...

```
void copy_array(int *src, int*dest, int n) {
      int i;
      for (i = 0; i < n; i++)
            dest[i] = src[i];
}
int a[1000] = {0, 1, 2, ..., 999};
```

It turns out `copy_array(a+1, a, 999);` executes about twice as fast as `copy_array(a, a+1, 999);`. Why is this?

`copy_array(a, a+1, 999);` copies a[0] to a[1], a[1] to a[2], etc. Conversely, `copy_array(a+1, a, 999);` copies a[1] to a[0], a[2] to a[1], etc. In the former case, a[1] is stored in iteration one and loaded in iteration two; in the latter case, a[1] is loaded in iteration one and stored in iteration two. Loading occurs early in the processor pipeline, and storage occurs late. So in the former case, iteration two waits early for something late in iteration one; whereas in the latter case, iteration two waits late for something early in iteration one.