E21: Digital System Design Lectures, Fall 1999

Dr. Bruce A. Maxwell, Asst. Professor
Department of Engineering
Swarthmore College

Course Description

This course covers digital system design. Topics include Boolean logic, digital representations, and techniques for design of combinational, sequential, and asynchronous circuits. We also study I/O interfaces, communication protocols, and micro-controller architecture. Labs focus on CAD techniques, VHDL [Very high speed integrated circuit Hardware Description Language], and programmable logic devices.

Prerequisites: CS21, E11, or permission of instructor

© 1999 Bruce A. Maxwell

This material is copyrighted. Individuals are free to use this material for their own educational, non-commercial purposes. Distribution or copying of this material for commercial or for-profit purposes without the prior written consent of the copyright owner is a violation of copyright and subject to fines and penalties.

F99 E21 Lecture #1

Introduction

Why digital logic?

- Reproducibility of results
  - If you have the same inputs, a well-designed circuit gives you the same outputs
- Ease of design
  - You don’t have to use calculus to model a digital circuit at the functional level
  - Small digital circuits can be visualized mentally without difficulty
- Flexibility and functionality
  - Try building a hard-to-break easy-to-use encryption algorithm using analog components
- Programmability
  - Digital circuits, if properly designed, can be programmed to do different things
- Economy
  - Digital circuits are cheap and small
- Advancing technology
  - Digital technology continues to improve: faster, cheaper, smaller, easier to use

Administrivia

The most important thing...

http://www.palantir.swarthmore.edu/~maxwell/classes/e21

- homeworks
- readings
- syllabus
- labs
- other course links
  - VHDL sites, hardware sites, tutorial sites, etc..

Focus of the course

- Basics of digital logic (pretty simple)
- Programmable digital devices: flexible design
- Trouble-shooting digital circuits
- Sensors and how to integrate them into digital systems
- Microcontrollers and FPGAs as the "brains" of small (embedded) digital systems

Final project

- The last 4-5 weeks you will be doing your own final project, start thinking about it now
- Last year: keyed security system, and it worked
- This year:
  - Security system for your room
  - Interface with a computer
  - Your own LED programmable scoreboard
  - Additional smart sensors for a robot

- ASIC for solving a particular problem (like finding lots of primes, quickly)
- Anything else you can dream up...

Expectactions

- Keep up: we will move quickly through much of the introductory digital logic topics
- Let me know if we're moving too quickly or too slowly
- Lab work is very important, because you won't learn about digital design unless you do it
- Digital design is 10% fun (at best) and 90% turn the crank
- We should be using good tools to make turning the crank faster & easier

Syllabus

- Textbook: only one required, I suggest getting a separate VHDL text
- Problem sets: Handed out Monday of each week, due the following Monday, in class.
- Labs: you can work in pairs.
- Labs: will occur approximately every 2 weeks
- Final project: labs will lead into the project, for which you will have the last 4-5 weeks
- Lab Notebooks: each of you will need to keep a lab notebook (to be handed in with each lab). In this notebook you need to keep enough information for someone else to recreate your work. You should also sign and date each entry as you complete it. Some of your work could potentially lead to a patent.
- Late policy

The basis of digital systems

1's and 0's

- Using electricity we can represent logical values as voltages
- Hi range v. Lo range with a section in the middle (unspecified value)
- Different hardware families have different ranges
- As long as we can guarantee the specs, a digital system will reproduce a signal given the same set of inputs.

Logic gates

- AND, OR, NOT (and symbols)
- NAND
- NOR
- XOR, Buffer, Multiple input AND, OR, NAND, and NOR functions

SSI components

Small-Scale Integration components were the first building blocks.

- Usually use 20 gates or less
- Usually cost around 25 cents
- 7400 series (letters tell you the family, numbers the configuration & function)
- 7400 is a quad 2-input NAND gate
- 7404 is a hex inverter
- 7432 is a quad 2-input OR gate (and so on)

SSI components are usualy used to interface with other, more complex components. It's not efficient to use them as the base building blocks except in some prototypes.

F99 E21 Lecture #2

The basis of digital systems

Number Systems

- Living with 1's and 0's is fine if you can use them to represent something.
  - Binary is a number system (base 2)
  - Conversion from decimal to binary: keep dividing by 2
  - Conversion from binary to decimal: sum up the value of each place containing a one
- Using binary numbers can be a real pain, and they are difficult to read.
  - Octal and Hexadecimal are two number systems that make it easier to read "bits"
  - Octal is base 8, and 3 bits (binary numbers) make one octal number.
  - Hexadecimal is base 16, and 4 bits make one hexadecimal number
  - Hexadecimal is preferred, we think in terms of bytes (8 bits) not 3, 6, or 9 bit values
- Explain hexadecimal (0-9, A-F)
- Convert 61453 (decimal) to hex -> F00D

So how do we represent negative numbers?

- Use a sign bit
  - makes adding and subtracting a pain
  - two representations of zero
- Use two's-complement representation (add binary value of low n-1 bits to $-(2^{n-1})$)
  - Think about it as a wheel
  - Like signed binary, the high bit indicates the sign of the number
    - sign extension: pad the high bits with whatever was in the old high bit
  - To invert In binary, complement each digit and then add 1
  - Range: one more negative digit than positive: $-r^n$ to $(r^n - 1)$
  - adding two’s complement
    - add them just like two binary numbers, just ignore any carry out of the high bit
    - overflow rule: when the two numbers have the same sign and the sign changes
    - subtraction: invert one number and add them
    - Examples: -4 + 4, 5 + 6, -7 + 5, 9 + 9

Other representations...

- Binary Coded Decimals
  - 0-9 using 4 bits
  - low 4 bits of the ASCII representation are BCD
  - useful for representing money transactions
- Packed BCD representation
  - use a fixed number of 4-byte chunks with one chunk being the sign
  - sign chunk is usually all 1’s
- Gray codes: used for encoding position: only one bit changes at a time
  - Gray codes are used in altimeters, motor encoders, etc.. (why?)
- ASCII codes: used for encoding both actions and characters

- Encoding actions: traffic light example:
  - NS go, NS wait, NS delay, EW go, EW wait, EW delay, G/Y/R for each,
  - 6 codes: how many bits?
- Often choose an encoding that minimizes the number of gates needed
  - balance experimentation with time

F99 E21 Lecture #3

Other representations...

- Gray codes: used for encoding position: only one bit changes at a time
  - Gray codes are used in altimeters, motor encoders, etc.. (why?)
  - Rule for developing gray codes
    - A 1-bit Gray code has two code words: 0 and 1
    - The first 2^n code words of an (n+1)-bit Gray code equal the code words of an n-bit Gray code, written in order with a leading 0 appended.
    - The last 2^n code words of an (n+1)-bit Gray code equal the codes words of an n-bit Gray code, but written in reverse order with a leading 1 appended.
- Encoding actions: traffic light example:
  - NS go, NS wait, NS delay, EW go, EW wait, EW delay, G/Y/R for each,
  - 6 codes: how many bits?
- Often choose an encoding that minimizes the number of gates needed
  - balance experimentation with time
  - sometimes trying to optimize a circuit can cause debugging problems

Error-detection and error correction

Error detection: use a dictionary of code words that is a subset of the 2^n possible code words

- A failure occurs when an invalid code word is sent

Say we want to know all cases where a single bit might be flipped

- Think about a hypercube, where each node represents a bit string
- Each node only takes a one bit modification to get to its neighbor
- Distance is defined as how many steps you have to take to get between two nodes

If we want to know when any single bit is flipped, which nodes on the hypercube can we use?

- Any set of nodes such that no two valid code words are less than a distance 2 apart
- using a parity bit (either even or odd parity) allows such a code

Now say we want to know how to correct one-bit errors and detect 2-bit errors

- Create a code with a minimum distance of 3
- A one bit error will cause an invalide code word, but it will be closest to one valid code
- A two bit error will be detectable, but not repairable
- Need to use more bits if you want to detect and correct 2 bit errors
  - In general, use 2c + 1 bits to correct up to c bit errors.

Hamming Codes

- A general method for constructing error correcting codes of minimum distance 3
  - There are i check bits for 2^i-1-i information bits
  - The bits in positions that are powers of 2 are the check bits (1, 2, 4, ...)
  - The check bits deal only with the bits in those positions that have a 1 in the same bit when their position is expressed in binary: for a 7-bit code: 1 -> 3, 5, 6, 2->3, 6, 7; 4->5, 6, 7
  - The check bits will have even parity for their group
  - There are at least two parity bits for any information bit

- To correct a bit, we locate the column that possesses all of the odd parity groups

Designing Simple Logic Circuits

Combinational Logic

- Circuit output is a function only of the current inputs.
- Inputs: signals, or variables
- Circuit: expressions, which are variables related by Boolean expressions
  - Boolean logic is defined by a set of axioms
  - Boolean logic: two-value logic (X = 0 iff X != 1)
  - Boolean product: AND function (1*X = X, 0*X = 0)
  - Boolean sum: OR function (1 + X = 1, 0 + X = X)
  - Boolean inverse: 1 -> 0, 0 -> 1
- Boolean theorems
  - Sum axioms
    - X + 0 = X
    - X + 1 = 1
    - X + X = X
    - X + X' = 1)
  - Product axioms
    - X*1 = X
    - X*0 = 0
    - X*X = X
    - X * X' = 0)
  - DeMorgan's theorem s
    - (X1 * X2 * ... * Xn)' = X1' + X2' + ... + Xn'
    - (X1 + X2 + ... + Xn)' = X1' * X2' * .... * Xn'

F99 E21 Lecture #4

Designing Simple Logic Circuits

Designing your first circuit: an adder

Truth table for a half-adder: A, B -> S, Cout

Truth table for a full-adder: A, B, Cin -> S Cout

Standard Representations

SOP representation: all of the 1 combinations

POS representation: all of the 0 combinations, but as inverted sum expressions

Question: can we do it better?

Minimization using Boolean logic...

Is there a better way?

Group minterms together that share bits

Do it graphically: Karnaugh maps

- 3 and 4 variable K-maps
- 5 and 6 variable K-maps (Bill's method)
- Group the 1's

Procedure for designing a combinational circuit of 6 variables or less

- Write the truth table
- Fill a Karnaugh-map for each output variable
  - Group the 1s by powers of 2
  - Express each group as a term in the Boolean expression
- Write the Boolean expression
- Draw the logic diagram
- Build the circuit

F99 E21 Lecture #5

Designing Simple Logic Circuits

More on Karnaugh Maps

- When you design a circuit, you don't always care about every input all the time
  - Priority situation: Allows you to collapse the truth table to just a few lines
  - Example: Bev, Pop, 1, 2 case
  - Drawing the Karnaugh map becomes easy
- Sometimes you don't care about the outputs
  - You usually care with combinational circuits, but not necessarily with sequential ones
  - Example: case where you only care about 5 of the 8 cases (last three will never occur)
    - You can label a don't care as either a 1 or a 0 on the Karnaugh map
    - You can use it in a group as you see fit to minimize the expression

When Karnaugh maps don't work...

- Karnaugh maps are good up to 6 variables
  - Difficult to visualize hypercubes beyond 6 dimensions

Quine-McCluskey Method

tabular method works for 6-10 variables, may a few more

1. Find all of the size 1-subcubes (minterms in the truth table)
2. n = 1
3. Group the minterms according to how many 1's they have in them
4. Use terms in adjacent groups to form (n+1)-subcubes
5. Label the level n minterms as to whether they are covered by a level n+1 subcube
6. Increment n and loop back to step 3

The list of terms not covered by a subcube at a higher level are the list of Prime Implicants

- Example 1: Bev, pop, 1, 2, [minterms 2, 3, 4, 5, 6, 7]
- Example 2: random function [minterms 0, 2, 4, 3, 6, 9, 7, 11, 13, 15]

Minimal Cover Generation

The complete list of Prime Implicants [PI] do not necessarily form the minimal cover expression.

1. Find the Essential Prime Implicants [EPI]
   - Essential prime implicants cover a minterm not covered by any other PI
   - Make a table with a row for each PI and a column for each minterms
   - Mark the minterms covered by each PI
   - Any column with only one mark identifies an EPI, which must be included in the function
2. Find the Minimal Cover Expression
   - Intelligently select from the remaining PIs to get a minimal cover expression
   - Method 1:
     - Form a POS, where each sum is the set of PIs that cover a particular minterm
     - Multiply it out to get a SOP term
     - Remove any expressions from the SOP that are covered by simpler expressions

    - The remaining terms are the possible choices to combine with the EPIs
- Method 2:
    - Depth-first search, and compare the resulting possibilities

F99 E21 Lecture #6

Designing Simple Logic Circuits

Technology Mapping

- Converting AND-OR gates to NAND/NOR gates
  - Rule 1: xy = ((xy)')': AND gate is a NAND gate with an inverter on the output
  - Rule 2: x+y = ((x + y)')' = (x'y')': OR gate is a NAND gate with inverters on the inputs
  - Rule 3: xy = ((xy)')' = (x' + y')': AND gate is a NOR gate with inverters on the inputs
- Example: ab + b’c + a’c
  - convert to NAND gates
  - convert to NOR gates
- After conversion, optimize by removing inverter pairs
- Example: full-adder function
  - F = x’yc’ + xy’c’ + x’y’c + xyc

Term Decomposition

- Sometimes we need to use gates with fewer inputs
  - programmable logic often requires a fixed number of gate inputs
- Term decomposition turns m-input gates into 2 or more n-input gates where n < m.
- Term decomposition must occur before technology mapping
- Process
  - Make a table with the levels, number of inputs, and number of gates
  - Each level takes an integral number of gates (inputs / inputs per gate)
  - Remaining inputs go to the next level
  - There will be log_m(n) levels
  - There may be numerous possible decompositions, you want to optimize it
- Timing optimization
  - There are often multiple different decompositions you can design
  - Pick the one that optimizes timing
    - minimize the longest path through the circuit

Logic Families

History

- 1930’s: Relays
- 1940’s: Vacuum tubes (350 volts = high) (MOSFETs developed)
- 1950’s: Bipolar junction transistors
- 1960’s: Integrated circuits, introduction of logic families, TTL, MOSFETs become practical
- 1980’s: CMOS development, CMOS replaces TTL logic
- 1990’s: 1-10 million transistor ICs
- 2000’s: 100-200 million transistor ICs

CMOS: Complementary metal oxide semiconductor

- CMOS logic levels: traditionally 0-1.5V = low, 3.5-5V = ground, 1.5-3.5 = undefined
- Other power levels are used as well: 3.3V, 2.7V, or lower for large ICs
- A MOSFET can be viewed as a voltage controlled resistor
  - For a NMOS transistor, current flows from the drain to the source when the gate voltage is higher than the source.
  - For a PMOS transistor, current flows from the source to the drain when the gate voltage is less than the source.
  - In both cases, when there is current flowing, the MOS can be viewed as a resistor with a small resistance (10-100 ohms)
  - When no current is flowing, the transistor appears as a large resistor (1 Mohm or more)
  - Regardless of the voltage, very little current flows through the gate

F99 E21 Lecture #7

Logic Families

A basic inverter

- NMOS and PMOS are used in a complementary fashion to design CMOS
- Put a PMOS and an NMOS transistor in series
- Connect the PMOS to Vcc and the NMOS to ground
- Connect the input to the gates of both transistors
  - When the input is high, the N-channel transistor is open, pulling the output to ground
  - When the input is low, the P-channel transistor is open, pulling the output to Vcc
  - When the input switches, there is a transistion period where current flows from Vcc to Gnd
  - If the output is connected to another CMOS gate, then almost no power is used except when the input switches
  - There is a capacitance at the gate that must be filled up/drained in order for the gate to reach steady-state
    - This capacitance draws power
    - This capacitance slows down the reaction time of the circuit

CMOS NAND and NOR gates

- k-input NAND and NOR gates can be created with CMOS logic
  - For k-inputs there are k p-channel and k n-channel transistors
- For a NAND gate
  - connect the p-channel transistors in parallel to Vcc
  - connect the n-channel transistors in series between the p-channel transistors and ground
  - connect each input to the gate of one p-channel and one n-channel transistor
  - If any of the gates are low, the output is pulled to Vcc
  - If all of the gates are high, the output is pulled to ground
- For a NOR gate
  - connect the n-channel transistors in parallel to ground
  - connect the p-channel transistors in series between the n-channel transistors and Vcc
  - connect each input to the gate of one p-channel and one n-channel transistor
  - If any of the gates are high, the output is pulled to ground
  - If all of the gates are low, the output is pulled to Vcc

Fanin

How many inputs can you have?

- Each transistor has a certain resistance
- Having more inputs means a higher resistance for the current to go through
- Each transistor has a certain capacitance
- Having more inputs means a higher capacitive load

Typically you don’t want to have more than 4 (NOR) to 6 (NAND) inputs on a single gate

(NMOS have a lower resistance than PMOS, hence the difference)

If you want more than that, use cascaded gates

Noninverting gates

You get inverting gates “for free” with CMOS technology (with most, in fact)

- AND
  - Connect an inverter on the end of a NAND gate
- OR
  - Connect an inverter on the end of a NOR gate

Reading a spec sheet

Look at page 98 of Wakerley

Fanout

How many gates can you connect to the output of a CMOS circuit?

  - depends upon how much current you’re going to draw from it
- Look at an inverter and treat the NMOS and PMOS transistors as appropriate resistances
- Model the load as a resistive load
  - Example: voltage divider with 1k (top) and 2k (bottom) resistors
  - Thevinin equivalent is a 667 ohm resistor and a 3.3 V source
  - Case one: low output, .43 V (sinking current)
  - Case two: high output, 4.61 V (sourcing current)
- Note: non-ideal inputs can cause problems because the transistor resistances become smaller
- Manufacturer will usually specify a load current that indicates what the maximum loads can be in order to maintain voltages that are less than (greater than) Volmax (Vohmin)
- Typical fanout from a CMOS device to other CMOS devices is about 20
- Note: at switching speeds, we also have to worry about AC characteristics
  - Capacitances slow down the switching speed
  - More fanout, means a single source is driving more capacitances in parallel

F99 E21 Lecture #8

Logic Families

CMOS Inputs

CMOS inputs should always be connected

- Gate will generally show a “low” value
- Only small amounts of current (static) will switch the value
- Tie inputs to ground or a voltage source

Schmitt-Trigger Inputs

- Because of feedback in the transistor, it follows a hysteresis curve
  - If the input is a low value, then it won’t go low again until the voltage rises above 2.9V
  - If the input is a high value, then it won’t go high again until the voltage drops below 2.1V
- The hysteresis helps to convert a noisy signal into a clean signal

Three-state Outputs

- Sometimes you want a circuit to have no output
  - When you have multiple circuits connected to a single wire, like a bus
- A three-state output has three states: high, low, and high-impedence
- You can think of this state as occurring when both the P- and NMOS transistors are off
- A three-state buffer takes as input the signal and an enable input
  - when the enable is high, the output has the same value as the signal
  - when the enable is low, both transistors are in the off state
  - Connect a NAND gate to the PMOS gate, with EN and A as inputs
  - Connect a NOR gate to the NMOS gate, with A and $\overline{\text{EN}}$ as inputs

CMOS families

Format: 74FFnn (54FFnn series are military grade)

- FF = family designation
- nn = number

- HC = High-speed CMOS
- HCT = High-speed CMOS, TTL compatible
  - Input ranges are compatible with TTL input logic levels
- AC = Advanced CMOS (introduced in the mid 80’s)
  - Can sink lots of current in both directions
  - Have extremely fast rise and fall times
  - Are so fast in their rise and fall times that they can be a major source of analog noise
- ACT = Advanced CMOS, TTL compatible
- FCT = Fast CMOS, TTL compatible
  - Faster switching times
  - Lower power consumption

Bipolar Logic

Basis of bipolar logic is the bipolar junction transistor.

- Diode: pn junction
  - A positive voltage drop across a pn junction allows current to flow
  - A negative voltage drop blocks current
  - A large negative voltage drop lets lots of current flow
- Transistor: npn or pnp junction
  - A positive voltage across the base-emitter pn junction allows current to flow
  - It also allows current to flow across the np junction from the collector to the base
  - The emitter output current is the sum of the base and collector input currents
  - The collector current is a multiple of the base current up to a saturation point

Common-emitter configuration

- Confguration
  - Resistor on the base input
  - Resistor on the collector input
  - Emitter connected to ground
- Effect
  - A positive input voltage on the base causes current to flow
  - A zero voltage on the base causes little or no current to flow

A basic inverter

- The simplest case is a single transistor with two resistors (base and collector)

TTL Families

- 74 = Basic family
- 74H = High-speed TTL
  - Uses smaller resistors for high switching speeds
- 74L = Low-power TTL
  - Uses larger resistors for lower power dissapation
- 74S = Schottky
  - Uses Schottky transistors that don’t saturate (diode connecting the base and collector)
  - 3ns / 19mW
- 74LS = Low-power Schottky
  - Same switching speed than 74H, but higher resistances and 1/5th the power consumption
  - 9ns / 2mW
- 74AS = Advanced Schottky
  - Twice the switching speed of S, with the same power consumption
  - 1.7ns / 8mW
- 74ALS = Advanced Low-power Schottky
  - Faster than LS, with lower power consumption
  - 4ns, 1.2mW

F99 E21 Lecture #9

VHDL: VHSIC Hardware Description Language

Digital Design (Old style)

- Using hierarchical methods, develop a design on paper
- [Post 1980: graphically design the circuit in CAD, or textually describe the structure
- Run the design through a discrete-event simulator]
- Build the circuit
- Test & debug cycle

How would you like to do digital design?

- Design in terms of circuit behavior and/or dataflow
- Use hierarchical methods
- Be able to describe and simulate the circuit at different levels
- Have designs be portable and reusable (modular)

Why do you want a hardware description language?

- To simulate hardware before implementing it
- To make design entry simpler than moving boxes and connecting wires
- To simplify verification and testing
- To simplify communication between different tools in a CAD environment
- To permit a standard for specification of inputs, outputs, and behavior of digital circuits

VHDL offers a standard language (IEEE Standard 1076 plus some additional standards)

- There are competitors (AHDL, Verilog HDL)
- Latest VHDL standard is 1993
- IEEE Standard 1164 standardizes data types for simulation
- IEEE Standard 1076.3 standardizes data types as they relate to hardware

How do you use VHDL?

- Design specification stage
  - can describe subcomponents in VHDL using input/output specifications
- Design synthesis
  - following certain styles of VHDL, you can map VHDL onto hardware
- Design simulation
  - you can simulate your design before putting it into hardware
- Design documentation
  - since VHDL is a standard, you can use the VHDL code as a formal specification
  - provides an alternative to schematics
  - provides an alternative to proprietary languages (Altera's AHDL is one example)

What is the overall structure of a hardware description language?

- At its essence, the HDL consists of methods of describing the two parts of a circuit
  - The input/output characteristics of the circuit (entities)
  - The internal functioning of the circuit (architectures)
- The other main structures of the language are designed to allow hierachical design

Simple Example 1: A comparator

```
entity compare is
      port(A, B: in bit_vector(0 to 7);
            EQ: out bit);
end compare;

architecture compare1 of compare is
begin
      EQ <= '1' when (A=B) else '0';
end compare1;
```

Entities & Architectures

- The entity describes the input/output behavior of the circuit
  - bit is a data type that describes a single wire carrying a Boolean value
  - bit_vector is a data type that describes a set of wires carrying Boolean values
- The architecture describes the function of the circuit
  - When the bits on the A and B wire sets are identical, then the circuit outputs a '1'
  - Otherwise, it outputs a '0'
  - The conditional assignment statement is a basic tool of VHDL
  - Note the architecture has to have a unique name, since there can be more than on architecture for a given entity
- A complete VHDL design must contain at least one entity and at least one architecture for that entity.

Other major language elements

- Packages
  - Like the class declaration for a C++ class
  - Specify constants, data types, function prototypes, aliases, etc..
  - Tell the rest of the world how to use the functions declared in the related package body
- Package Body
  - A set of functions/routines/circuits you can put into a library for reuse
- Configurations
  - Specify which architectures go with which entities, etc.

Levels of Abstraction (styles)

- Behavioral
  - Performance descriptions
  - Test Benches
  - Sequential Descriptions
  - State machines

You can imply registers and register transfers using behavioral language, but there is no guarantee the circuit will work exactly as you want it to. Altera can handle some behavioral constructs

- Dataflow (Register-Transfer Level)
  - State machines
  - Register transfers
  - Selected assignments

- Arithmetic operators
- Boolean equations

Dataflow is often the highest level that synthesis tools can handle

- Structural
  - Boolean equations
  - Hierarchy
  - Physical Information

The structural representation is considered a netlist representation of the circuit

Ditched the rest of this and just did an example of using dataflow (case 1) and behavioral modeling (case 2) to demonstrate a 3-bit priority encoder for the microwave input circuit

Behavioral Modeling

Think about circutis as implementing an algorithm (software = hardware)

- Process statement is the key: things within a process are implemented sequentially
- You can write a general algorithm within a process
- A process is executed continuously, but must have some sort of wait condition
  - Until the wait condition is reached, you can think about all processes executing in the same delta-t.
- Within a process you can describe an algorithm

Designing a shifter using a Process

A process allows you to specify actions over time (sequential actions)

```
library ieee
use ieee.std_logic_1164.all;

entity rotate is
      port(Clk, Rst, Load: in std_logic;
            Data, in std_logic_vector(0 to 7); -- eight bits
            Q: out std_logic_vector(0 to 7);   -- eight bits
end rotate

architecture rotate1 of rotate is
      -- this line creates an 8-bit register in the design
      -- note that signal assignments come before the begin stmt
      signal Qreg: std_logic_vector(0 to 7);
begin
      -- process activates when either Rst or Clk changes
      -- process name in this case is reg
      reg:process(Rst, Clk)
      begin
            if Rst = ‘1’ then -- Asynchronous reset part, assign Q 0 vector
                  Qreg <= “00000000”;
            -- the following says if Clk is 1 and it was Clk that changed
            elsif (Clk = ‘1’ and Clk’event) then
                  if(Load = ‘1’) then
                        Qreg <= Data;
                  else
                        -- modifies the order of Q
                        Qreg <= Qreg(1 to 7) & Qreg(0); -- & is concatenate
```

```
                -- need an end if statement (note semi-colon
                end if;
            end if;
        -- need an end process statement
        end process;
        Q <= Qreg;
end rotate1;
```

- The process will activate any time there is an event on Rst or Clk (sensitivity list)
- You can consider a process as executing in zero time

F99 E21 Lecture #10

IEEE 1164 Data Types

std_ulogic

- Treat it like a bit, ‘0’, ‘1’ are what you will use most of the time

std_logic

- Treat it like a bit, ‘0’, ‘1’ are what you will use most of the time
- Different is that std_logic is “resolved”, which means that there is a matrix that defines the output if two bits of type std_logic are traveling over the same wire

std_logic_vector

- Vector of bits, need to declare it with the range of the indices and order left to right
- A: in std_logic_vector (2 downto 0);
- B: out std_logic_vector (0 to 7);
- Access by A(i) or B(i)

Combinational Logic Design Components

Given a basic understanding of VHDL, we can now look at how we can represent MSI components as logic, netlist, dataflow, and behavioral elements.

Read sections 5.1 and 5.2 on documentation and timing and follow them as appropriate

Aside: Rules for drawing MSI circuit elements

- Inputs on the left
- Outputs on the right
- Inputs that are inverse logic have a bubble (or overline)
- Outputs that are inverse logic have a bubble (or overline)
- Power connections are not normally shown

Decoder

- Function
  - Binary number as input
  - One of 2^n outputs is high
  - With enable inputs, you can cascade decoders together to handle larger words
  - Commercial binary decoders use negative logic on the output
- Logic diagram
  - Each output has exactly one entry in the truth table
  - Each output has a unique AND gate combination
  - Enable in an additional input to each AND gate
- Netlist representation
  - Table 5-14, Wakerley
- Dataflow representation 1
  - with A select Y_L_i <= “01111111” when “000” ...
- Dataflow representation 2

    - do all level conversion at the front and back end, and make the with statement positive
- Dataflow representation 3 (hierarchical)
    - create a V3to8dec entity which is called by the 74x138 architecture
- Behavioral style representation 1
    - Use a process, with a case statement
    - case A is when “000” => Y_s <= “10000000”; ...
    - if (G1 and G2 and G3) = ‘1’ then Y <= Y_s; else Y <= “00000000”; end if;
- Behavioral style representation 2
    - Use a process with a for loop
    - Y <= “00000000”
    - if(G1 and G2 and G3) = ‘1’ then
    - for i in 0 to 7 loop
    - if i = CONV_INTEGER(A) then Y(i) <= ‘1’; end if; end loop; end if;

F99 E21 Lecture #11

Libraries in VHDL

To use a library in VHDL you have to include it (just like a .h file in C)

- library ieee; -- library statement says to find this library
- use ieee.std_logic_1164.all; -- use statement indicates what parts to include

You will probably put those two lines at the top of all of your VHDL designs

The definition of the std_logic package is given in its entirety on pp. 394-5 of Wakerley

Combinational Logic Design Components

Decoder

- Function
  - Binary number as input
  - One of 2^n outputs is high
  - With enable inputs, you can cascade decoders together to handle larger words
  - Commercial binary decoders use negative logic on the output
- Behavioral style representation 1
  - Use a process, with a case statement
  - case A is when “000” => Y_s <= “10000000”; ...
  - if (G1 and G2 and G3) = ‘1’ then Y <= Y_s; else Y <= “00000000”; end if;
- Behavioral style representation 2
  - Use a process with a for loop
  - Y <= “00000000”
  - if(G1 and G2 and G3) = ‘1’ then
  - for i in 0 to 7 loop
  - if i = CONV_INTEGER(A) then Y(i) <= ‘1’; end if; end loop; end if;

Encoder

- Function
  - 2^n inputs, any number of which may be high
  - One binary number (n bits) as output, encoding the highest priority line that is high
- Most useful definition is behavioral
  - Behavioral 1: use an if statement that expresses the priorities of the inputs
  - Behavioral 2: use a for loop that loops through the inputs in order or priority

Note: with an encoder, decoder, and tri-state buffer we can start to make a bus and bus controller

Multiplexer (MUX)

- Function
  - 2^n input signals
  - n input address
  - enable signal
  - 1 output, which is the signal indicated by the address

- Cool things you can do with a MUX
  - Implement an arbitrary truth table for N variables
  - Use it as an implementation of a ROM (identical to a truth table)
  - Make really big MUXs by cascading them using the enable inputs and putting the results through an OR gate
- Dataflow: Could have one conditional assignment statement (long)
- Behavioral 1: case statement is a multplexor
- Behavioral 2: convert the address to an integer and index into the input array to get the output

Demultiplexer

- Function
  - One input and one address as inputs
  - 2^n outputs, one of which gets the signal
  - Same function can be executed with a decoder with an enable value
- Dataflow: one conditional case statement for each output
- Behavioral 1: case statement, one line for each address
- Behavioral 2: set all the outputs to 0, convert the address to an integer and set that output high

F99 E21 Lecture #12

Comparators

An XOR gate acts as a one-bit comparator: low if equal, high if not

- To compare multiple bits, we could cascade XOR gates and AND gates
- A faster way is to compare multiple bits simultaneously and the OR the result

We could also do a cascade system to test for greater-than/less-than

- Greater-than: start with the MSB and work down
- Less-than: start with MSB and work down

Shifters/Rotators

We can use DEMUXs on the inputs to generate a shifter/rotator circuit

- One DEMUX for each input
- Address of DEMUX indicates how much to shift the input
- Each output is an OR gate with inputs from the appropriate DEMUXs

Ripple Adders

The first combinational circuit we looked at was an adder

- Can make a multiple-bit adder by cascading full-adders together
- Each adder takes A, B, Cin, and outputs S and Cout

Carry Lookahead Adders

Ripple-adders are slow: signal has to propagate

- Difficulty is the carry bit: use a circuit to precalculate the carry bits
- At each level Ci+1 = Gi + Pi * Ci
  - Gi = Xi * Yi
  - Pi = Xi + Yi
- pp. 435 of Wakerley shows the equations for four bits of lookahead
- Each circuit only takes 3 levels of logic: faster than propagation
- Full binary adder is on pp. 437
  - Note the use of INVERT-OR-AND logic, which is like the AND-OR-INVERT logic that is so fast since it uses only one or two layers of transistors.
  - Carry equations are slightly different: Ci+1 = Gi * Pi + Pi * Ci
  - Same equation, though, because Pi is always 1 if Gi is 1
  - Allows factoring: Ci+1 = Pi * (Gi + Ci) (this is the INVERT-OR-AND logic)

When you build large adders, you use carry-lookahead between blocks as well as within a group

ALUs

MSI Arithmetic Logic Units are all purpose computational circuits: see Wakerley pp. 439

- Shifting
- Rotating
- Bit-wise logical functions
- Subtraction
- Addition

Programmable Logic Devices

Circuits that use AND-OR combinations are common

- Can generate arbitrary truth tables using minterms

Programmable Logic Array: PLA

- N inputs and their inverses at the first level
- P AND gates with all possible inputs at the second level ($P < 2^N$)
- M OR gates with all possible inputs at the second level (M = number of outputs)
- The AND and OR gate levels both have fuses on all lines into each gate
- Blow the fuses to program the circuit

Programmable Array Logic: PAL

- OR-gate array is fixed
- Outputs can be inputs of one block can be inputs to another block
- Output pins are bi-directional: one AND gate controls the direction of the pin
- Diagram on pp. 341

Generic Array Logic: GAL

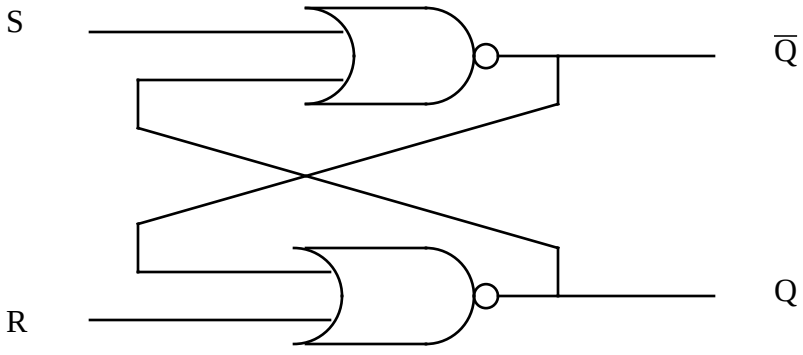
- Output polarity can be controlled by blowing a fuse on the input to an XOR gate
- Can be set up to do some sequntial logic as well by encoding a flip-flop

Large Scale PLDs

- Altera MAX7000 family: thousands of logic gates in blocks
- Electrically programmable
- Electrically erasable
- A “fitter” puts the designs on the chip

Sequential Logic

Flip-flop: 2 NAND gates



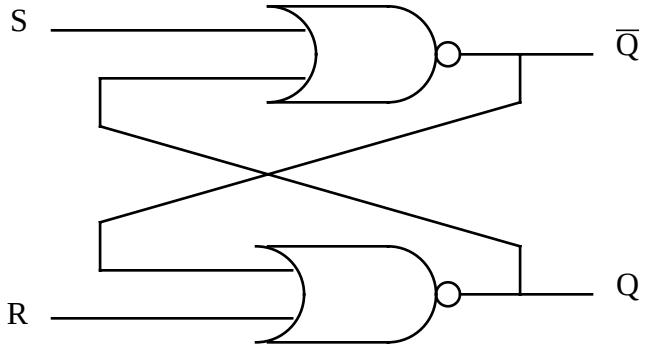
F99 E21 Lecture #13

Sequential Logic

Flip-flop: 2 NAND gates

What does this do?

- S-R Flip-flop
- Bistable device
  - Give starting values to Q/$\overline{Q}$
  - Assign values to S/R
  - Follow through the circuit
- S = 1: sets the flip-flop to 1
- R = 1: resets the flip-flop to 0
- S = R = 0: holds the flip-flop value
- S = R = 1: both outputs are 0, indeterminate value if S and R -> 0 simultaneously



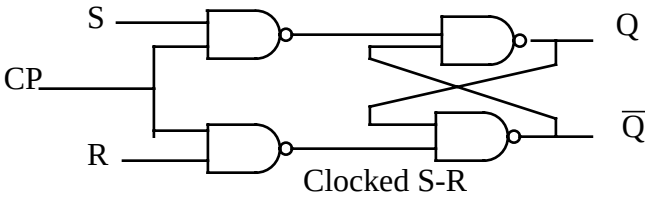
Basic S-R

Useful Flip-flops

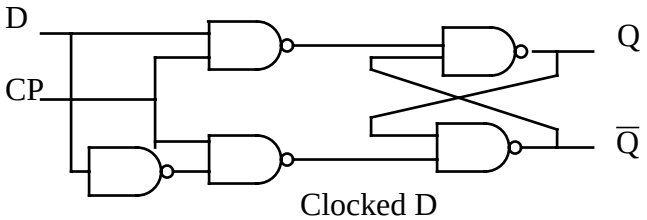
- Clocked S-R
- D Flip-flop
- JK Flip-flop
- T Flip-flop

Problem with basic flip-flops is that they can change as long as the clock is high

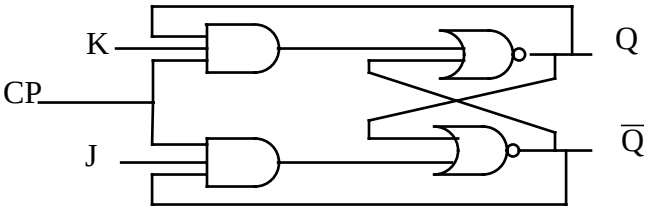
- Master-Slave Flip-flop
  - Two-flip-flops, with the clock pulse inverted on the second
  - Ouput only changes on the falling edge
- D edge-triggered Flip-flop
  - pp. 215, [Mano 1991]
  - Output only changes on the rising edge



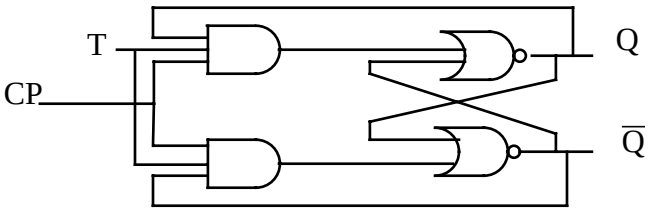
Clocked S-R



Clocked D



Clocked J-K



Clocked T

Designing a Sequential Circuit

Set up a state table

- Inputs
- State values
- Next state values
- Outputs

Select a flip-flop type

Calculate the values to set the FFs

F99 E21 Lecture #14

Sequential Logic

State Machines

A sequential circuit can implement a state machine

- A state machine is a way of representing a time/history dependent process
- A state is a set of variables that represent all of the knowledge you need in order to predict the next state of the system given a set of inputs.
- State variables are held in flip-flops
- Next state calculation is a combinational circuit
  - Feedback loop from the current state
  - Also depends upon the current input
- Outputs are combinational circuits with at least the state variables as inputs
  - Moore: output is only a function of the state variables
  - Mealy: output is a function of both the state variables and the inputs

Example: Timer circuit

- We want to design a timer circuit that repeatedly counts to eight and the outputs a 1
- There are no inputs (other than the clock)
- The 8 states are defined by the three binary variables
- The output logic is a function of the state variables (1 when the state is 111)

Design Methodology

Draw a state diagram

- States are represented as circles
- Directed edges connect the states
- Inputs are list on the edges, as are outputs in a IN / OUT notation
- There should be one edge leaving each state for each possible input variable combination

Assign state values to the states

- Decide how many flip-flops are necessary (state variables)
- Decide how to assign values to the states
  - binary
  - gray codes
  - based on output logic considerations
  - based on similar meanings
  - use all 0’s or all 1’s for the initial state (reset state)

Set up a state table

- Inputs are the circuit inputs and current state values
- Outputs are the next state and the state machine outputs

Minimize the number of states in the state table/diagram

- Two states are the same if
  - They produce the same outputs for the same inputs
  - They go to the same next states (or equivalent next states) for all inputs

Select a flip-flop type

- Based on your requirements
- Based on which one is easiest to use
- Based on which one optimizes the circuit

Use the characteristic equation for the flip-flop type to set up the flip-flop inputs in the state table

Design a combinational circuit for each flip-flop input and each output variable

Draw the logic diagram

Build the circuit

Example: Timer circuit II

Design a repeating 20 second clock that pulses at 14 and 19

- Draw the state diagram
- Set up the state table
- Select a flip-flop: JK flip-flops
- Fill in the state table for the JK inputs
- Design the next state and output circuits
- Draw the logic diagram

Example: Traffic light

- The traffic light moves between six states
- The output is a function of the state variables
- The input is a variable indicating whether to switch states or not
- Draw the state diagram
- Set up the state table
- Select a flip-flop: T flip-flops
- Fill in the state table for the JK inputs
- Design the next state and output circuits
- Draw the logic diagram

Practical design considerations

- Design a reset into your circuit
  - Asynchronous reset is good (power-up)
  - Synchronous reset is necessary if your flip-flops don’t have an asynchronous reset
- Design any extra states so they go to logical places

F99 E21 Lecture #15

Administrivia

Exam Friday: 1 8.5” x 11” page of notes (single sided)

Homework should be in an envelope outside my office by tomorrow noon

Sequential Logic

Example: Recognize the sequence 010 in a bit stream

- Draw the state diagram
- Set up the state table
- Select a flip-flop: D flip-flops
- Fill in the state table for the D inputs
- Design the next state and output circuits
- Draw the logic diagram

Example: Timer circuit II

Design a repeating 20 second clock that pulses at 14 and 19

- Draw the state diagram
- Set up the state table
- Select a flip-flop: jk flip-flops
- Fill in the state table for the JK inputs
- Design the next state and output circuits
- Draw the logic diagram

Table 1: JK Excitation

Q	Q’	J	K
0	0	0	d
0	1	1	d
1	0	d	0
1	1	d	0

Example: Traffic light

- The traffic light moves between six states
- The output is a function of the state variables
- The input is a variable indicating whether to switch states or not
- Draw the state diagram
- Set up the state table
- Select a flip-flop: JK flip-flops
- Fill in the state table for the JK inputs
- Design the next state and output circuits
- Draw the logic diagram

Practical design considerations

- Design a reset into your circuit
  - Asynchronous reset is good (power-up)
  - Synchronous reset is necessary if your flip-flops don’t have an asynchronous reset
- Design any extra states so they go to logical places

F99 E21 Lecture #16

Exam Review

Any questions on the exam?

Useful Sequential Circuits

Simple Registers

A set of D flip-flops with the load line being the clock pulse

Need logic attached to the clock pulse to enable/disable the load feature of the register

- Usually a system will have a master clock
- We want delays in clock to be minmal at best
- Logic between the clock and the flip-flop inputs delay the system

SR flip-flop register with parallel load and clear

- Load signal goes through a buffer
  - Reduce the fan-out seen by other circuits
- Inputs to SR go through AND gates with the load signal
- Clear signal goes directly to the FF clear input
- Register clock goes through an inverter to decrease the load on the master clock
- Flip-flops change state on the falling edge of the master clock

D flip-flop register with parallel load and clear

- When load line is high, the inputs are ANDed with the load line to drive the D inputs
- When the load line is low, the D outputs are ANDed with the inverse load line to drive the D inputs
  - D flip-flop value must be fed back around in order to maintain its value when the load line is low because there is no way to tell a D flip-flop to maintain its last value

Example: ROM and a Register

If we have a large truth table, we may want to implement the circuit with ROM and a register.

- 2^N states
- Given a truth table with I inputs, N state variables, N next state values, and M outputs
- Create an N + I input ROM (2^{N+I} words) with N + M outputs
- Feed N of the output values back into the register at each clock cycle
- Outputs come directly from the ROM
- 2-chip circuit implementation of a very complex state machine
- simple uCode controller for a processor

Shift Register

Bi-directional shift register with parallel load

- Register can shift the values left one bit, right one bit, load in parallel, and maintain
- Register also has a parallel clear input
- You can implement one using D flip-flops and a MUX for each bit.

1. Clear control to clear the register to 0
2. CP input for a clock pulse
3. Control lines: 00 = maintain, 01 = load input, 10 = shift left, 11 = shift right
4. N input lines, as well as left and right input lines

Use MUXs on the D flip-flop input and have 2 input lines:

- Clear is buffered
- Clock is inverted

F99 E21 Lecture #17

Counters

A counter is a register with an incrementer circut on each bit: half-adder

- It's like adding 0 to the current value with a carry-in
- An XOR gate handles the new value of each bit
- An AND gate handles the carry propagation
  - You can do carry-lookahead to speed things up

To make an up-down counter you use a half-adder subtractor

- Add a direction input D: 0 = count up, 1 = count down
- Add an enable input E: 0 = don't change, 1 = follow direction input
- New bit value is still an XOR gate for both directions
- New carry value is $C_{i+1} = \overline{D}Q_iC_i + D\overline{Q}_{\bar{i}}C_i$
- Enable input is the C_0 input

To make a presetable up-down counter you use:

- HAS for each bit
- load inputs
- a 2-1 MUX to handle the loading v. counting

BCD counters

- Set up circuitry so that instead of counting to 10, it loads 0 after reaching 9
- To effect an up-down BCD counter, you want to load either 0 or 9 upon reaching 9 or 0.
- Use a 2-1 selector for each bit of the load inputs to load either 0 or 9.
- Have the load go high when the outputs are either 0 or 9 in the appropriate direction
- Have the direction input specify whether to load the 0 or 9 at the next clock
- This requires a synchronous load in order to work!
- With asynchronous load you have to do other things
  - AND the clock and a 10 output together and feed it into the load signal for counting up
  - AND the clock and a 15 output together and feed it into the load signal for going down

Ripple counters

- You can make a very simple counter using T flip-flops
- Connect the inverse output of each bit to the clock of the next bit
- Have an enable line go in parallel to each T input (1 = enabled, 0 = hold)
- Connect the clock to the first bit
- Ripple counters are slow, but easy to build
- You can make faster ripple counters by using blocks of 4 bits and calculating the carry between chips

Timing signals

You can use a counter as a timer

- Select a flip flop of the timer as an output to divide the clock frequency by a power of 2
- On a synchronous counter ('163), set it up so that when it reaches a certain value it resets or loads a new value on the next rising edge.

Ring counter

- A set of flip-flops where a 1 goes around in a circle

2^N stage pulse timer

- N-bit counter connected to a decoder to create 2^N staggered timing signals

F99 E21 Lecture #18

Register Files

For fast access of a small number of registers (cache, register files in a CPU)

- 2D array of flip-flops that each have the following inputs
  - read enable
  - write enable
  - input
  - clock
- D = WE * Input + $\overline{WE}$ * Q
- Output is a 3-state buffer with output enable = read enable line
- Register files can often read two registers per cycle and write to two registers per cycle
- Decoders determine which register (row) to read/write, with one decoder for each operation.

RAM

Static (SRAM)

- Information stored in flip-flops
  - Faster
  - Valid as long as the power is on
  - Takes more chip space per bit (at least 2 NAND gates (8 transistors) per bit)
  - Typically used for registers and cache

Dynamic (DRAM)

- Information stored in capacitors
  - Slower
  - Charge dissapates every few milliseconds
  - Takes less chip space per chip (one transistor per bit)
  - Typically used for main memory

RAM read/write process

1. Chip select
2. Decode the address
3. Activate the appropriate locations in memory
4. Write data to the memory cells / Interpret and output data

With each bit you need to be able to get:

- Input
- Output
- Read/write
- Select

Like in the ROM, a decoder selects the word (N inputs, 2^N output lines)

We also need to be concerned about timing

- Address decoding takes time
- Chip selection takes time
- RW select takes time

Timing pattern for reading: governed by CS and OE inputs

- Put address on the address lines
- Set chip select [CS] and output enable [OE]
- Output enable time determines when the data is ready to be read
- Address can be changed without affecting the output for a certain amount of time
  - output hold time
- Chip deselect
- Outputs are still good for a certain amount of time
  - output disable time

Timing parameters for read operations

- t_{AA}: Access time from address. This is how long it takes for the data to be valid given a new address on the address lines (and assuming CS and OE are enabled)
- t_{ACS}: Access time from chip select, assuming OE and address. This is usually the same as t_{AA}, but not always.
- t_{OE}: This is how long it takes for the 3-state output buffers to leave the high impedence state once OE and CS are both asserted. This is usually less than t_{ACS}, and can be used to queeze just one more bit of data on the bus.
- t_{OZ}: Output disable time. You need to know this to maximize the throughput on the bus.
- t_{OH}: Output hold time. This tells you how long the output data is valid after you change the address. This can also be used to maximize throughput on a bus when accessing multiple memory locations.

F99 E21 Lecture #19

RAM

SRAM Diagram: Wakerly pp. 857

- WR_L = WE_L * CS_L
- IOE_L = OE_L * CS_L

Timing pattern for writing: cycle governed by CS and WE inputs

- Put address on the address lines (address setup time)
  - Address hold time
- Select chip [CS] and set write enable [WE]
  - Data is put into the flip-flops on the falling edge of CS
- Assert data
  - Data needs to be stable before falling edge of CS (data setup time)
- Deassert CS and RW select
  - pulse width of CS and RW need to be sufficiently long (write pulse width)
  - Data is read into flip-flops
- Deassert address

Timing parameters for the write-cycle

- t_{AS}: Address setup time before write. All of the address inputs must be stable for this amount of time before both CS and WE are asserted, otherwise the data may be written to the wrong place.
- t_{AH}: Address hold time after write. All of the address inputs must be stable for this amount of time after either CS or WE are deasserted, otherwise the data may be written to the wrong place.
- t_{CSW}: Chip-select setup before end of write. CS must be asserted at least this long before the end of the write cycle in order to select a cell.
- t_{WP}: Write-pulse width. WE must be asserted at least this long to reliably latch data into the selected cell
- t_{DS}: Data setup time before end of write. All of the data inputs must be held stable at least this amount of time before the write cycle ends. Otherwise it may not be latched correctly.
- t_{DH}: Data hold time after the end of the write cycle. All of the data inputs must be held at least this amount of time after the write cycle ends.

RAM is never organized like a register file in a 1D array. Instead, it is organized so there is a row and a column combination to access a particular bit, ideally each with N/2 input bits.

You can do this using row and column decoders, or you can do this using a row decoder and column multiplexors (similar complexity to decoders).

For digital design, we often design RAM circuits from smaller components

- Use higher address lines to select the RAM chip
- Use lower address lines to get the data

SRAM Performance

Asynchronous SRAM: 1999

- Largest SRAMs have 4Mbits
  - 256K x 16, 512K x 8, or 1M x 4 bits
  - 10ns access times
- Fastest SRAMs have 4Kbits
  - 1K x 4 bits
  - 2.7ns access time

Synchronous SRAM: 1999

- clocked interface for control, address, and data
- allows for straightforward pipelining of the process and maximum throughput
- can be used in “burst” mode where consecutive addresses are read or written
  - no need to send a new address (internal address register increments)
  - you get new data on each clock cycle
- SSRAMs are available with clock cycles as high as 166MHz
  - Assume you can access 32 bits/cycle
  - 664 MB/s throughput
  - This is still a factor of 4 slower than the processor

DRAM

A transistor and a capacitor

- The goal is a single electron storing the information (is it possible?)
- High densities: factor of 64 x SRAM densities
- Values need to be refreshed every 60ms or so
- Chip is still available for R/W 99% of the time

DRAM timing is similar to SRAM timing

- Asynchronous DRAM timing is strange, and uses both the rising and falling edges
- Synchronous DRAM timing is simpler and now dominates the PC memory market
- DRAMs take much longer to read than SRAMS
  - 60ns - 100ns is typical for a single read cycle
- SDRAMs help because the operations are pipelined and you can run in “burst” mode

This is why you have a memory hierarchy

- CD-ROM
- Hard Drive
- Hard Drive cache (DRAM or SRAM)
- DRAM
- SRAM
- Internal registers of the CPU

F99 E21 Lecture #20

Algorithmic State Machine Design

Algorithmic state machine design is a method for designing data processing circuits

Principles

- Specify the data processing task as a hardware algorithm that consists of register transfer operations and a sequencing mechanism
- Separate the design of data processing logic and control logic
- Gives the designer a way to graphically indicate timing and processing actions

Building Blocks

- State boxes (square)
  - each state box corresponds to one state, and one state encoding
  - the state boxes indicate which output signals are asserted during that state
  - the state boxes indicate register transfer operations that occur during that state
  - the default value of a variable is 0 in any state for which it is not otherwise specified
- Decision boxes (diamonds)
  - Each decision box depends upon one Boolean variable: two exit paths
- Condition boxes (ovals)
  - The condition box indicate what output signals or register transfers should take place along a conditional path.

ASM Blocks

- A state and all of the following decision and conditional boxes form a block
- All outputs of a state appear at the rising edge when the state begins
  - Outputs are a function of the state variables
- All register transfer operations appear at the rising edge when the state ends.
  - Setup time is during the state, and value may be dependent upon state outputs

Designing the Data Processing Unit

ASM states and conditional blocks provide all of the information necessary to create the data processing unit(s).

- Assume the control section is a black box.
- Assume a single line for each control signal
- Data processing unit should be designable by observation

Designing the Control Unit

There are numerous methods you can use to design the control unit.

Design by standard state machine techniques

- Convert the ASM control flow to a state machine
- Build the state transition/output table
- Design based on the transition/output table

Design by using a flip-flop for each state

- Convert the ASM control flow to a state machine
- Use one flip-flop for each state in the state machine
- Design directly from the state machine by ANDing the incoming conditions to each state with the state they came from
- When using a register to hold the flip-flop values:
  - Invert the logic coming out of the first flip-flop so that when the register is cleared at start-up, the first flip-flop has an output value of one
  - Invert the logic coming into the first flip-flop so that it represents negative logic

Design with decoders

To simplify output and input variables that rely on the current state, we can use a decoder at the output of the state machine where a single line represents each state.

This is similar to using N flip-flops for N states in that you have N outputs from the state machine, each one corresponding uniquely to a state.

Design with multiplexers

We can design with multiplexers by using them to calculate the next state values

- The output of the current state register is connected to the selector inputs of the MUX
- The output of the MUX is connected to the state register inputs
- The MUX inputs are determined by the input variables and the control flow
  - Create a table showing all current state -> next state transitions
  - List the different input variable combinations relevant to these transitions
  - The corresponding MUX input is the sum of the 1 terms

Table 1: Example Table for 0110 bit string recognition problem

Q0	Q1	Q0’	Q1’	Inputs	MuxQ1	MuxQ2
0	0	0	0	I		
0	0	0	1	$\bar{I}$	0	$\bar{I}$
0	1	0	1	$\bar{I}$		
0	1	1	0	I	I	$\bar{I}$
1	0	0	1	$\bar{I}$		
1	0	1	1	I	I	1
1	1	0	0			
1	1	0	0		0	0

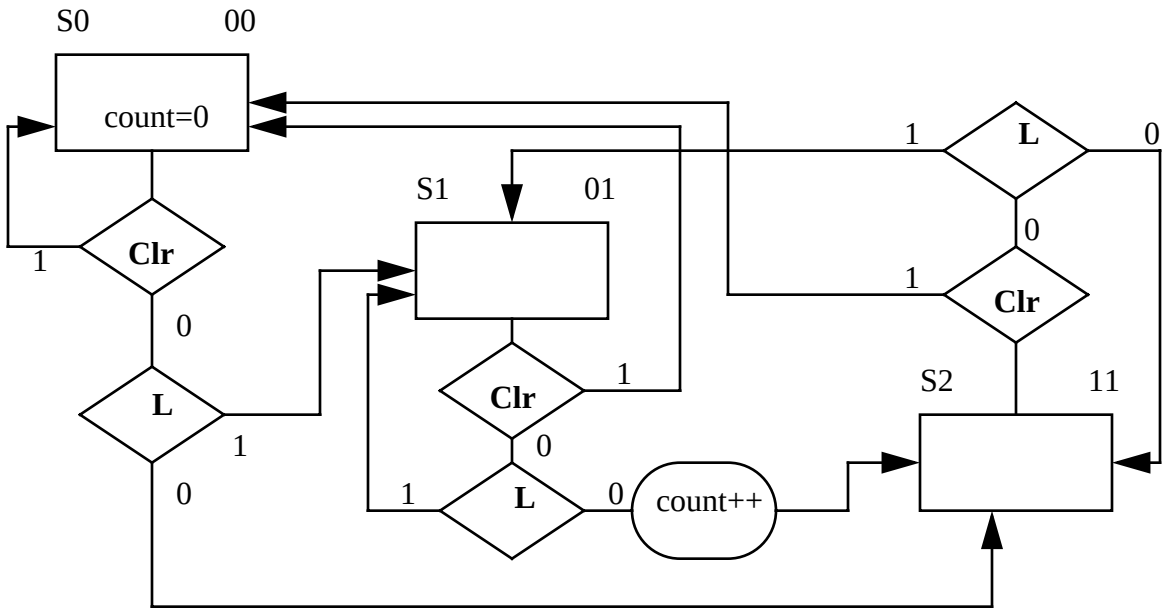
Design with a PLA or a PROM

- Create state transition table
- Burn it onto the chip

Example

Design a circuit that counts the number of times a laser beam is broken (by someone walking through it, for example). The inputs to the circuit are: Clear, Laser. Clear should reset the counter to 0, Laser indicates whether the beam is broken (1) or not (0). If someone decides to stand in the beam for more than one clock cycle, that should still count as only one “break”.

ASM:



Data Processing

- Use an N-bit register with synchronous load (rising edge) and an incrementing circuit
- Use a MUX to specify what to load (0’s, current value, or value + 1)
- MUX input should be 00 in state 0, 01 in state 2, and either 10 or 11 in state 1
  - make some combinational logic for this (3 variables, 2 outputs)
  - 00 = input 0’s
  - 01 = current value
  - 10 = curernt value
  - 11 = increment

Control

- Use MUX design to move between the states
- 2 variables = 2 muxes
- It turns out we don’t really need the MUXs after all, but it’s easy to design

Table 2: Laser beam counter MUX design table

Q1	Q2	Q1’	Q2’	I	MuxQ1	MuxQ2
0	0	0	0	Clr		
0	0	0	1	$\overline{Clr} * L$		
0	0	1	1	$\overline{Clr} * \overline{L}$	$\overline{Clr} * \overline{L}$	$\overline{Clr}$
0	1	0	0	Clr		
0	1	0	1	$\overline{Clr} * L$		
0	1	1	1	$\overline{Clr} * \overline{L}$	$\overline{Clr} * \overline{L}$	$\overline{Clr}$
1	1	0	0	Clr		
1	1	0	1	$\overline{Clr} * L$		
1	1	1	1	$\overline{Clr} * \overline{L}$	$\overline{Clr} * \overline{L}$	$\overline{Clr}$

F99 E21 Lecture #21

Sensing & A/D interfaces

Digital circuits are very nice, because they have a natural buffer against noise

- It takes a lot of noise to send a zero to a one, or vice-versa
- We have straightforward digital means of detecting and correcting single bit-flips

The world is continuous

- How do we get from the analog world to the continuous world?
- We digitize it/quantize it/convert it to information

What can we measure?

- count
  - fingers/toes
- length
  - reference stick
- voltage
  - reference zener diodes

Counting and voltages are the most useful for us to measure

- An oscillator can provide a time-varying signal that can drive a digital circuit (be converted to a clock)
- A voltage can be converted to a set of bits representing its value relative to a range

Keywords

- Precision: number of significant digits in the result
- Accuracy: how close you are to the true value
- Sensitivity: what is the smallest change you can sense reliably
- Resolution (signal): # bits in an ADC
- Resolution (time): sampling frequency
- Dynamic Range: range within which the sensor operates
- Resolution (spectral): how precisely can you measure frequency?
- Nyquist Criterion: you have to sample at least twice the frequency you want to see

A/D converters

- Common circuit
- One or more input pins that can handle voltages within a specific range
- N output pins that output the voltage level converted to a number
- Analog reference voltage(s)
- Digital Power/Gnd
- Enable/Clock input pin

Important characteristics of A/D converters

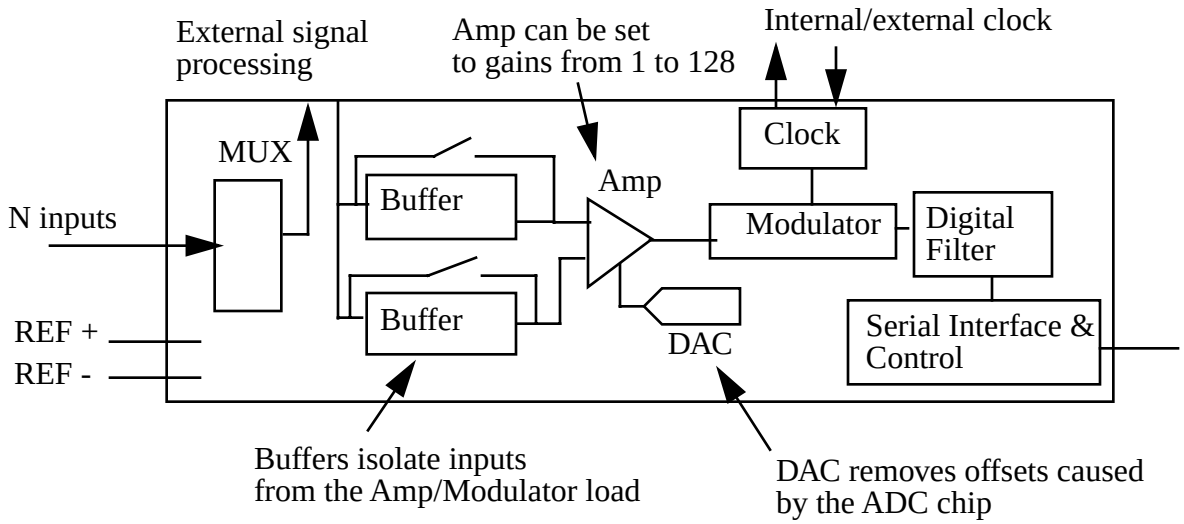
- Resolution: number of bits N
  - The resolution of the A/D converter depends on N
  - N of 8 is common
  - N of 12 is now common, 14, 16 bits are now possible

    - In a 5V range, 8 bits = 20mV, 12 bits = 1.2mV, 16 bits = 80uV
- Number of input lines
- Number of samples that can be take simultaneously
- Parallel/Serial output
- Maximum sampling frequency
    - 1Gsps is now possible at 8 bits: can pick up 500kHz signals (telephone lines)

Design of a high-end ADC: MAXIM MAX1400 ADC ($9 each)

Resolution: 16 bits

Sample frequency: 480sps at 16-bits/sample, or 4800sps at 12-bits/sample



- This chip communicates via a serial interface
- You can set it to scan all of the inputs sequentially and output the results
- The modulator (ADC component) and digital filter have multiple settings
- You can put external signal processing circuitry in between the analog MUX and the capture buffers that hold the signal values

Designing an A/D system

Measurement methodology

First you have to decide how you are going to measure something

Direct measurement

- Directly comparing the measurand to a calibrated standard
    - Meter stick
    - Zener diode

Indirect measurement

- Measuring a quantity that is related to the thing you want to measure
    - Measuring the wall of a furnace rather than sticking the thermometer inside
    - Measuring blood pressure using the “Korotkoff” sounds

Null Measurement

- Comparing the measurand to a calibrated source and then adjusting one of them until the difference between them is zero
  - Take a known reference voltage source connected to a potentiometer and an unknown voltage soruce and put them on a zero-center galvanometer. Adjust the pot until the galvanomter reads zero then read the pot.
  - Scales

F99 E21 Lecture #22

Designing an A/D system

Measurement methodology

Direct measurement

Indirect measurement

Null Measurement

System Components

External stimulus -> sensor/transducer -> amplifier -> analog processing -> A/D converter

-> computer port -> software processing -> sensor output/decision-making

Transducer: converts one form of energy (information) to another (usually electrical)

- piezoelectric material: converts pressure to electricity
- silicon: temperature sensitive, EM sensitive, pressure sensitive
- spring: converts force to a distance

Amplifier circuit:

- May include some processing before amplication (low/hi-pass filters)
- Amplifiers are low-pass circuits: higher the amplication the lower the cutoff frequency

Analog processing

- You can do quite a bit in analog hardware
- Can significantly reduce the load on a computer (microcontroller or PIC)
- Has to include a low-pass filter that meets the Nyquist criterion ($< 2f_s$)
- May include a dc cutoff capacitor

A/D converter

- Precision is determined by the number of bits
- Frequency is determined by the signal you want to measure
- Often needs to be much higher than just $2f_s$

Computer Port

- Shielding issues
- Need a computer interface circuit
- Bandwidth issues

Software procesing

- Correctness
- Are you getting the right signal?
- Speed

Decision-making

- Control cycle, how do you make a decision once you have a measurement?

Sensing and Error

Measurement error

- Theoretical error
  - Transducers are non-linear but you assume linearity
  - Equations used to produce the sensor output are approximations of the actual values
- Static Error
  - Interpolation errors: reading a meter stick
  - Environmental error
  - Manufacturing or design errors (offsets, flaws in the meter stick)
- Dynamic Errors
  - The measurand is moving while you're trying to measure it
  - Inertia of mechanical devices
  - Frequency limitations
  - Hysteresis
- Instrument Insertion Error
  - The environment is altered by the measurement process

Nature imposed noise

Johnson Noise: thermodynamic origin

- A resistor has instantaneous current in it (which over time will create power)
- White noise phenomenon
- Constant in magnitude across the spectrum

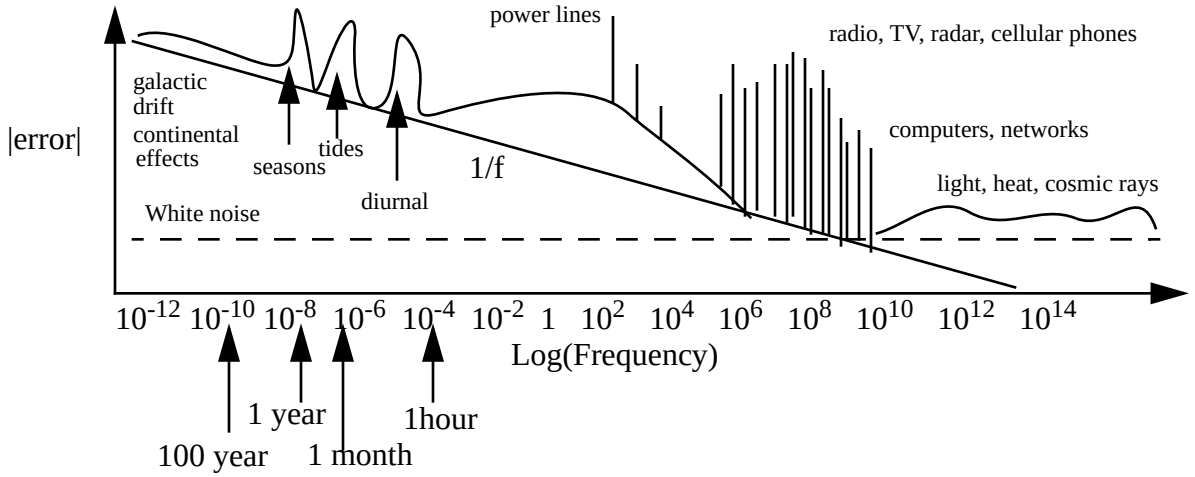
Shot Noise: quantization of charge

- Also a white noise phenomenon

Flicker (1/f noise): origin is quantum mechanical

- Related to the uncertainty principle
- Increases in magnitude with decreasing frequency
- Engineers can always hang the "blame" on some particular device

These are nature imposed. They are lower bounds on how well you can do.



Human imposed noise

- Rotating machinery
- Power Lines
- Traffic
- Radio
- TV
- Microwaves
- Radar

Other noise sources

- Galactic effects
- Continental drift
- Seasons
- Tides
- Diurnal effects
- Weather
- light, heat, and cosmic rays

Solutions to measurement error

- Use high-quality instruments and good designs
- Use the instrument that least disturbs what is being measured
- Use several different instruments to measure the quantity
- Use multiple measurements and average the results
  - Will remove zero-mean Gaussian noise
  - Will not remove salt & pepper noise
- Use multiple measurements and take the median value
  - Will remove salt & pepper noise
  - Will probably be close to the true value unless the measurements are biased

F99 E21 Lecture #23

Finish up Sensing

Solutions to noise

- Lock-in Amplifier
  - Turn a slowly varying signal (but not the background) on and off at a relatively high but arbitrary frequency
  - This creates an AC signal that is amplitude modulated by the information of interestt
  - Rectify the resulting signal
- Measuring differences directly
  - Don't measure two large quantities and then subtract them, this magnifies errors
  - Example: measuring the distance between a door and a door frame
- Use noise to our advantage
  - When digitizing a signal add some white noise
  - Take multiple measurements of the signal
  - The average will be more accurate when you add the noise
  - What you are doing is equivalent to rounding to the nearest significant figure (LSB)
- Watch out for "antennas, magnetic pickup loops, and ground loops

Example: Building a light sensor for a laser

- Building a sensor with a photoresistor
  - Resistance decreases with increasing light levels
  - Need to convert ohms to voltage/current
  - Use a simple inverting/non-inverting op-amp circuit
  - Look at the gain (megaohms to 100 ohms) using a fixed feedback resistor
  - Set the power supply to an appropriate value
- How do you deal with noise in the system?
  - Is there sufficient difference between the lights on/lights off and the laser that you don't need to worry about noise?
  - Do you need to shade the sensor?
  - Do you need to chop the laser at a high frequency and then integrate/low-pass the rectified AC signal?
- How do you make a decision with the signal?
  - Analog logic that sets a TTL level: set the output to the integrated output, a low signal represents when the laser has been broken.
    - Let the op-amp saturate at 5V max signal
    - Make sure the op-amp does not saturate and has a low value when the laser is off
  - A/D convert and use software processing to make a decision: look at the signal over time and make a decision using a rule-based framework.

Example: motion sensor using ultrasonic transducers

- Get a transmitter and a receiver
  - Pulse the transmitter with a big pulse (square wave, short duration, high voltage)

- Sensing voltage across the receiver (connect one side to ground, the other to an op-amp)
- Amplify the signal to a reasonable level
- Integrate the signal so that the output of the system is a big peak whose maximum is directly related to the return time by some offset.
- A/D convert the signal and take the maximum value as the return time
- Do the calculations
  - Sound = ~300m/s
  - We want to resolve 5cm (theoretical limit is 0.5 cm)
  - Therefore, we need to be able to resolve the difference between peaks that are .05m / 300 m/s = .16ms apart
  - Therefore, we need to sample at least twice that frequency (~12kHz)
  - Then we come up with an equation D = (time + offset) * speed
  - offset is found using a calibration procedure
- To use as a motion sensor, just look for significant changes in the distance. When the change is above a certain number, then activate the alarm/turn on the lights/whatever...

F99 E21 Lecture #24

Intellectual Property Rights

What is it?

- Ideas
- Manifestations of ideas: books, art, catchy phrases
- Processes and Inventions
  - internal combustion engine
  - process for manufacturing integrated circuits
  - person detector
    - combination of sensors in a new and unique configuration
    - digital logic to tie it all together and implement it
- Software (only possible through computers)
  - Algorithm
  - Source code
  - Object code
  - Look & feel

Why do we have intellectual property laws?

In order to promote art and science.

- Reward people for their innovation
- Ensure that knowlege is distributed

Four useful categories

- Trademark: phrase, name, logo
- Trade Secret: something you don't want to let anyone else know about
- Copyright: manifestations of ideas
- Patent: processes, inventions

Trade Secret

Trade secret laws allow companies to keep secrets.

A trade secret must:

1. have novelty
2. represent an economic investment to the claimant,
3. have involved some effort in development, and
4. the company must show that it made some effort to keep the information a secret.

Most well-known example: Coca-Cola

In the technical world: Industrial Light & Magic

Problems:

- Laws are not uniform throughout the U.S. (and definitely not beyond)
- The laws were not written with computer technology in mind

    - There is a risk if you go to court in untested territory
    - court could decide that trade secret laws don't apply
- The company must reveal the secret to sell the software
- Once the trade secret is known, the trade secrecy laws do not apply

Copyright

- Copyright office has been accepting computer program source code since 1964.
- In 1980, congress amended the copyright law to explicitly include programs under the category of literary works that are automatically covered
- Definition: "a set of statements or instructions used directly or indirectly in a computer in order to bring about a certain result."
- Weak form of protection
    - Only protects the expression of the idea
    - Can write the algorithm in another language
    - Can develop the code independently (not copying)
    - Burden of proof is on the copyright holder
        - Have to show access
        - Have to show that you copied the material
- Does copyright extend to the look and feel of a program?
    - The structure, sequence, and organization of a program falls under copyright protection
    - However, external forces may require similar structure among programs
    - *Is the look and feel of a program the idea, and the source and object code the expression?*
        - Look and feel could not be copyrighted, then.
- Recent changes in copyright law affect digital transmission and encoding of information.
    - You can now be prosecuted for building devices to break encryptions designed to maintain copyrights.
    - Certain encryption & digital technologies allowed under the new laws threaten fair use
        - Pay per use encryption methods means libraries would have to charge for each use by a borrower
        - You would have to pay a price if you wanted to let your friend borrow a document
- Some copyright stakeholders are lobbying for copyright to databases that just contain facts (like NBA statistics).
    - Current law is that a database is not copyrightable unless it involves a creative process.
- Fair use laws
    - Copying is permitted under certain restricted situations
    - Archival copy of software
    - Copying for personal scholarly use (but not for the design of commercial products)
    - Copying for classroom use in certain situations (only the first time)
- Whether a use is fair use depends upon four factors
    - The purpose and character of the use, including whether such use is of a commerical nature or is for nonprofit educational purposes.
    - the nature of the copyrighted work

    - the amount and substatiality of the portion used in relation to the copyrighted work as a whole; and
    - the effect of the use upon the potential market for or value of the copyrighted work

Patent

A patent is the strongest form of protection for intellectual property because it gives the developer a monopoly over the use of that idea for a period of 14 or 20 years.

- *Main purpose of a patent?*
    - To encourage the advancement of useful arts and sciences
    - Foster invention
    - Promote disclosure of inventions
    - To ensure the idea is in the public domain
- What can be patented?
    - Process
    - Machine
    - Article of manufature
    - Composition of matter
    - Improvement of any of the above
    - Ornamental design of an article of manufacture
    - Asexually reproduced plant varieties by design and plant patents
- What cannot be patented
    - abstract ideas,
    - laws of nature,
    - physical phenomena,
    - literaray, dramatic, musucal and artistic works
    - inventions which are not useful or offensive to public morality

To give someone the right to these things would inhibit scientific thought and advancement.

A patent claim must:

1. fall within the category of permissible subject matter
2. have utility
3. have novelty
4. must be nonobvious
5. be adequately described or enabled (for one of ordinary skill in the art to make and use the invention)
6. claimed by the inventor in clear and definite terms

What types of patents are there?

Utility patents may be granted to anyone who invents or discovers any new, useful, and nonobvious process, machine, article of manufacture, or composition of matter, or any new and useful improvement thereof.

Design patents may be granted to anyone who invents a new, original, and ornamental design for an article of manufacture.

Plant patents may be granted to anyone who invents or discovers AND asexually reproduces any distinct and new variety of plant.

Provisional v. Non-provisonal patents

The non-provisional application establishes the filing date AND initiates the examination process.

The provisional application only establishes the filing date and automatically becomes abandoned after one year. You may file a provisional application when you are not ready to enter your application into the regular examination process. A provisional application establishes a filing date at a lower cost for a first patent application filing in the United States and allows the term "Patent Pending" to be applied to the invention. Claims are not required in a provisional application. The PTO does not examine a provisional application and such an application cannot become a patent. You must submit the non-provisional application within one year of submitting your provisional application in order to possibly receive the benefit of the provisional application's filing date. You do not have to file a provisional application before filing a non-provisional application.

Process

1. Generate a great idea, process, or invention
2. Figure out if your idea is patentable
3. Figure out if it infringes on any other patents
4. Start the process with a provisonal application (optional)
5. Develop your non-provisional application (usually with a lawyer)
    - Title
    - Background of the invention
    - Description of the prior art
    - Summary of the invention
    - Brief description of drawings in the application
    - Detailed description of the invention
    - Claim(s)
        - Claims define the invention and are what are legally enforceable. Therefore, they are extremely important. Whether a patent will be granted is determined, in large measure, by the wording of the claims. Claims continue to be important once a patent is granted, because questions of validity and infringement are judged by the courts on the basis of the claims.
    - Abstract
    - Oath or declaration
    - Sequence listing, if there is one
6. Go through the challenge process with a patent examiner
    - You try to make as broad claims as you can
    - The patent examiner tries to minimize your claims
7. If you patent is granted, then you can claim a patent on your invention
    - You now have the right to license or sell your patent
    - You now have the right to sue someone for infringement

Software Patents

Computer software faces its greatest challenge with number 1.

For the purpose of a patent, software is generally considered a process.

70s and 80s, the argument was the software could be duplicated by a mental process, which is inappropriate subject matter.

90s, attention focuses on how computer algorithms are different from mathematical algorithms (if at all)

- Freeman-Walter-Abele test: mathematical algorithms are abstract ideas and unpatentable unless there is a practical application of the algorithm. In previous cases a practical application of an algorithm consisted of data being transformed through mathematical calculations to produce a smooth waveform display on a mointor. A practical application required physical elements or steps.
- In State Street Bank & Trust Col v. Signature Financial Group (1998), Inc., the Court of Customs and Patent Appeals held that “the transformation of data, representing discrete dollar amounts, by a machine trough a series of mathematical calculations into a final share price, constitutes ap ractical application of a mathematical agorithm, formula, or calculation, because it produces a useful, concrete and tangible result - a final share price”.
    - This removes the “physical element” aspect of patentable software
    - This requirement now becomes that a computer program, method, or process must produce a useful, concrete, and tangible result.

The concern now is that there are too many software patents

- Before you sell a piece of software you have to do a patent search
- The organization of software by the patent office is poor at best, and is continuously being reorganized by the USPTO
- This makes software development a risky business, and creates barriers to entry

Digital System Patents

You no longer have to provide source code something like a robot controller that relies on software to run (you used to).

You do have to say that it relies on software and give an explanation of the function of that software. There has to be sufficient explanation that a person “conversant in the field” could recreate it. (Robotic Vision Systems, Inc. v. View Engineering, Inc., 1997).

This also shows that you must describe the “best mode” for controlling a device in your patent. If the best mode is software, you must disclose the function of the software.

You might want to provide the source code to the patent office to help keep future patent seekers from infringing on your patent or from trying to patent something you already created (Translogic Corporation v. Tele Engineering, Inc, 1997). This works in two ways

- It will make it easier for a patent seeker to do a patent search.
- It will keep others from trying to patent a piece of software that you have already created.

F99 E21 Lecture #25

Genetic Algorithms and Evolutionary Programming

The traditional design of systems has evolved a set of useful design tools

- Mathematical models
- Lumped parameter systems
- Analog system simulators based on the above [SPICE]
- Digital components and design methods
- Discrete-event simulators
- Programming languages
- Modular design

These design tools allow us to design circuits that behave well over a range of temperatures, noise levels, and other external influences

These design tools also allow us to understand, simulate, and trouble-shoot our designs

However, they also limit our ability to create systems

- We abstract away from the physical world to simplify the task
- We stay within a particular domain--digital logic / analog design / functional programming--to keep the system simple, intuitively understandable, and modelable
- This limits the space of possible designs we consider for a particular system

What if we could let the system explore the space of possible designs on its own?

- Biologically motivated
- Evolution is the exploration of design spaces by a self-modifying system
- Can we apply evolution to algorithms?
- Can we apply evolution to physical systems?

Genetic algorithms [GA]

GAs were originally designed as a massively parallel optimization/search method

Basic genetic algorithm

- Represent a solution to a problem as a bit string
- Generate a population (N) of random bit strings (population size)
- Evaluate all of the bit strings in the population
  - A “Fitness Function” evaluates each individual
  - A better solution should be assigned a higher fitness
- Stochastically select N strings from the population to become the next generation
  - Strings with a higher fitness are more likely to be selected
- Stochastically select M <= N strings to participate in crossover (crossover rate)
  - Stochastically select a location on the bit string
  - Swap the halves so that two new strings are created from the two parents
- Stochasticaly mutate some of the bits on strings in the population (mutation rate)
  - Set a mutation rate μ and test each bit in the population to see if it flips
- Repeat the fitness evaluation and new generation selection (# generations)
  - Terminate when the solution is good enough, or the population has largely stabilized

Variations

- Elitism: guarantee that at least one copy of each of the best B solutions appears in the next generation
- Evolution strategies: use the actual numbers instead of bit strings, and co-evolve standard deviations the determine how the numbers get modified each generation
- Population-based Incremental Learning: ditch the population and just keep the statistics, generate new populations and then move the statistics towards the best individual

Issues with GAs

- Good fitness functions are essential
  - The GA will learn to optimize the fitness function
  - The fitness function should be a continuous function (although it doesn't have to be)
  - If you want robustness to temperature, etc., put that in the fitness function
- The mutation rate is important
  - No mutation limits the search space
  - Too much mutation makes the search random
- The population size is important
  - A larger population size gives you a better sampling of the search space
  - A smaller population makes the process go faster (but does it go to the right place?)

Evolvable Hardware

Hardware circuits form an interesting system

- Subject to physics
- Situated in the real world
- Time-varying components
- Current designs are limited by our tools

Just like any other system, we can evolve hardware circutis

- Set up a physical system
- Develop a bit string representation for it
- Develop a method of evaluating the system's fitness
- Run the GA

F99 E21 Lecture #26

Evolvable Hardware

Example: Tone Discriminator

Task: discriminate between a 1kHz signal and a 10kHz signal (both square waves)

Physical circuit: Xilinx 6200 programmable logic chip

- Use only one quadrant of the FPGA: 100 blocks in a 10x10 array
- Each block has four inputs and four outputs (N, S, E, W)
- Blocks on the interior edges have either three, or two inputs/outputs
- Blocks on the exterior edges connect one I/O pin (select a direction)
- Each block as the following structure
  - Three multiplexors that select among the four input signals
  - One multiplexor that selects a function F
- Each output is a MUX that selects among three inputs or F as the output

GA: population = 50, crossover rate = 0.7, mutation rate = 2.7 per generation

Fitness function: maximizes the difference between the average output voltages for a 1kHz signal and a 10 kHz signal

Result: after 5000 generations the circuit works perfectly

- Always changes correctly on the falling edge of an input waveform
  - At the end of 200ns the circuit has made a decision
- Only uses a small portion of the 10x10 grid (21 blocks)
- Forms three components (A, B, C)
  - Parts A and B go inactive during the high part of a pulse
  - Part C remains static during the high part of a pulse
  - 200ns after a falling edge, the circuit assumes the correct state
  - They designers can’t figure out how it keeps time
- The output is temperature sensitive

Example: Robot controller

Task: avoid walls and keeping moving

Physical setup: two sonars, two motors

Electrical system: “Dynamic” state machine

- 1k by 8bits RAM
  - 10 address input (6 inputs fixed)
  - 8 data outpus
  - Only 32 bits of information, since 6 address lines are fixed
- Optional latches on the RAM addresses (signal can be latched by a clock, or passed through asynchronously) (4 bits)
- Optional latches on two of the RAM outputs going to the motors (2 bits)
- The 2 RAM motor outputs go back to the 10 address inputs
- The clock frequency is evolved (16 bit gray code: range 2Hz to several kHz)
- 54 bits to represent a particular DSM

GA: population = 30, crossover rate = 0.7, mutation rate = 1/generation

Fitness function: integrates a value proportional to distance from the wall, and subtracts a value if the robot is stopped (penalizes a stopped robot).

- Fitness of each bit string (configuration) evaluated for 30s
- Towards the end of the training, fitness was evaluated for 90s
- Training took place in a robot "virtual reality" with the wheels off the ground and simulated sonar inputs
  - Noise was added to both in order to simulate reality

Results: after 35 generations the robot shows good performance

- Robot used 32 bits of RAM
- 3 flip-flops
- Clock generation

Analysis

- evolved to use a 9Hz clock (~twice the sonar frequency, hmmmm)
- sonar inputs to the RAM were asynchronous (fairly slow pulse signals)
- both motor outputs were clocked
- internal state variable for the left motor was asychronous
- internal state variable for the right motor was clocked

This is neat: since the motors are clocked, we can think of the state as being sampled by the motors on a 9Hz schedule

- The clock allows the robot to move with a slight waggle, causing the sonars to scan the walls and minimize erroneous sonar readings (hmmm)

Training for robustness I: robot controller with bit errors

Task: same as before, but one of the RAM bits gets incorrectly stuck to a 1 or a 0

Problem: can't feasibly test each individual for 32 different faults

Solution:

- Start by training a population as above for N generations (85)
- After N generations, test the consensus string for all 32 faults, pick the worst
  - consensus = string with each bit being the most likely for its position in the population
- Iterate the following
  - Create the next generation, evaluating the fitness function using the "current fault"
  - Test the consensus individual on all 32 faults and pick the worst as the "current fault"

By generation 204, the consensus individual displayed complete fault tolerance

Training for robustness II: the Evolvatron

How do you make a circuit robust to temperature?

- Put one circuit in the freezer
- Put one circuit in a heated environment
- Put one circuit on the table
- Use FPGAs from different batches, and with different cases
- etc..
- Evaluate the fitness function on all of the different FPGAs in all of the situations

Custom hardware for evolution of buildable circuits: the Evolvable Motherboard

Make a programmable crosspoint architecture that can connect a variety of elements

The crosspoint architecture also allows you to monitor each signal in the system

Hook it up to a computer using a bus-based connection (ISA, PCI, etc.)

Each switch on the crosspoint architecture has some resistance

Example: Inverting amplifier

Task: Evolve an inverting amplifier with a gain of -10

Physical setup:

- EM connecting the bases, emitters, and collectors of transistors
- Switches act as resistors in the circuit

Fitness function: have the output match the amplified input signal

Driving function: 1kHz sine wave, 2mV peak to peak amplitude, offset at 1.4Vdc

GA Characteristics: population = 50, elitism, crossover rate = 1, mutation rate = .01 per bit

- 48 rows/columns
- Each column could have up to 4 connections to a row
- 1056 bits in each string

Results: it worked, using two amplifiers and lots of switches

Example: Evolving in simulation

Task: generate a transistor amplifier

- Have to require the simulator to use standard values for capacitances and resistances
- Have to put in penalties for high collector and emitter currents
- Have to match component parameters to the real world such as saturation current and gain

With these constraints, the GAs trained more slowly, but had a higher probability of working when built

Without these constraints, the GAs trained more quickly, but didn’t create buildable circuits

F99 E21 Lecture #27

Quick Review & Administrivia

Exam Monday: someone will be there to hand out the exam and pick it up

Cheat sheet: 1 side of an 8.5” x 11” piece of paper

Topics

- Anything on combinational logic
- Sequential circuits
- Sequential circuit design
- State machine design methods
  - Standard state table design
  - MUX design
  - ROM & Register design
- RAM design & configuration
- Sensors & sensing
- A/D conversion topics
- Evolvable hardware

ROM: Read Only Memory

We havent really looked at the structure of a ROM, or the types that are available on the market

Basic ROM

Also called a Mask ROM because it is manufactured with the information on it

- Manufacturer uses a mask to control what bits are written into the ROM
- Design time: about 4 weeks

Diode-based ROM design (bipolar technology)

- N bit Address, N = X + Y, with X and Y as equal as possible
- Decoder selects a row based on the X highest address bits and pulls it low
- Diodes connect selected columns to each row
- Each column is connected to Vcc through a resistor
- If a diode connects a column and a row that is pulled low, then the column is pulled low
- The columns feed into a multiplexer that is selected based on the low Y address bits
- Output is a single bit.
- < 100ns access time

Transistor-based ROM (MOS technology)

- An NMOS transistor connects each column to ground
- The gate of the transistor is connected to a row
- Decoder selects a row and sets it active-high
- A high transistor connects the column to ground, pulling the column low
- 10-200ns access time

PROM

Programmable ROM

- Bipolar technology
- The circuit is built with each column connected to each row through a diode
- A fuse connects the diode to the column
- The user can blow fuses (just once) to program the chip
- < 100ns access times
- Design time: 10-50 us/byte

EPROM

Eraseable programmable ROM

- Floating gate technology
  - A transistor connects each column (drain) to ground (source)
  - A capacitor sits between the row and the gate
- To program a low bit, apply a high voltage, which stores charge in the gate capacitor
- To erase it, expose the chip to UV light, which breaks down the capacitor material (dielectric) and lets the charge escape
- EPROMs store 70% of their charge for at least 10 years
- (Note: most PROMs are EPROMs without any lid for erasing them)

EEPROM

Electrically eraseable programmable ROM

- Uses thinner material as the dielectric for the gate capacitor
- Gate can be erased by applying a negative charging voltage to the gate
- Limited number of erases: ~10,000 per bit
- Flash EPROMs: erasing only happens in large blocks
- Use: configuration memories for computers

ROM configuration & timing

Configurations

- Inputs:
  - Output enable [OE] line (3-state buffers on each data output)
  - Chip select [CS] line
  - Pair are ANDed together to control the 3-state buffers
- Connection to microprocessor
  - RAM is usually set up as a high address space
  - When the high N address lines are all asserted, then access the ROM
  - Feed the rest of the address lines to the ROM
  - Feed the microprocessor READ output to the OE input of the ROM
  - Have a MUX select and control the CS line if you have multiple ROM chips
  - CS line de-asserted also tends to set the chip into a power-down mode
- Timing Parameters
  - t_{AA}:Access time from address
  - t_{ACS}: access time from chip-select
  - t_{OE}: access time from output enable

    - t_{OZ}: output disable time
    - t_{OH}: output hold time
- Process
    - Processor asserts the address
    - This sets the chip select and starts the access process
    - Time passes by
    - Processor asserts the READ line (OE)
    - Data appears on the output lines
    - Processor de-asserts READ, address

F99 E21 Lecture #28

I/O Interfaces

I/O Configuration for a computer

- Memory bus (Proprietary)
- Main peripheral bus (PCI / SCSI)
- Secondary peripheral bus (ISA)
- Parallel port
- Serial port
- USB
- Firewire
- Ethernet

Each of these I/O connections has its own protocol for communication

An I/O port will generally have the following information

- Data registers
  - Might have buffers associated with them
  - Often mapped to memory locations
- Control registers
  - Specify the configuration of the I/O unit
  - Transmission speed
  - Communication protocol
- Status registers
  - Specify the state of the I/O unit
  - Data ready
  - Data sent
  - Errors

General models of communication

Strobing

- One unit controls the interaction
- Sending information
  - Strobe active indicates unit has information to send
  - Strobe and data go inactive after some period of time
  - Has to wait as long as the slowest device before removing the data
- Receiving information
  - Strobe active indicates unit wants some information
  - Data is read after some time period
  - Strobe goes inactive after the data has been read
  - Has to wait as long as the slowest device before reading the data
- Problem: you have no idea what the other device is doing

Handshaking

- Both units communicate their readiness
- Sending information

    - Sending unit sets strobe/request active to indicate the data is ready
    - Receiving unit sets reply high to acknowledge it has received the request
    - Sending unit sets strobe/request inactive to indicate it is about to remove the data
    - Receiving unit sets reply inactive to indicate it is ready for another transmission
- Receiving informaiton
    - Receiving unit sets strobe/request active to indicate it wants some data
    - Sending unit sets reply active after it has put the data on the data lines
    - Receving unit sets strobe/request inactive after it has read the data
    - Sending unit sets reply inactive when it is ready to receive another request

Handshaking between state machines

- States are: 00, 01, 11, 10 (gray code!)
- Two 4-state state machines talking to one another
- Can be simulated as two state-machines in VHDL (but Altera doesn't like it)

Serial Communication

Common method of communication

- telephones
- motor control systems
- peripheral communication
- bandwidth is limited

Methods of serial communication

- Simplex mode: information flows one direction only (2 wire communication)
    - Radio/TV are examples
- Half_duplex mode: information flows both directions, but only in one direction at a time
    - Old-modems
    - Requires some turnaround time to switch directions
    - Still a 2-wire mode
- Full-duplex mode: information flows both directions simultaneously
    - Requires either 3 wires, or non-overlapping frequency bands
- Synchronous communication
    - Both the transmitter and receive have clocks that are at the same speed
    - A regular timing signal is used to keep the clocks synchronized
- Asynchrnous communication
    - Transmitter and receiver may have different rate clocks
    - Information is only broadcast when there is information to send
    - Format of the message indicates when there is information arriving

Asynchronous communication protocol

- Start bit (0)
- Character bits (7 or 8 of them)
- Parity bit (1)
- Stop bit (1-2 of them)

Synchronous transmission does not use the start and stop bits, but must use timing signals to keep the bits synchronized to the clock

F99 E21 Lecture #29

I/O Interfaces: USB Protocol

The original method of adding devices to a computer was to give each device its own I/O interface

- Add an ISA or PCI card to the computer
- Connect to the serial or parallel port

In the mid-90's, seven companies (Compaq, DEC, IBM, Intel, Microsoft, NEC, and Northern Telecom) got together to make a new standrard with the following goals:

- Users must not have to set switches or jumpers
- Users must not have to open the case to install new devices
- There should be only one kind of cable
- I/O devices should get their power from the cable
- Up to 127 devices should be attachable to a single computer
- The system should support real-time devices (sound, telephone)
- Devices should be installable while the computer is running
- No reboot should be needed after installing a new device
- The new bus and its I/O devices should be inexpensive to manufacture

The USB [Unversal Serial Bus] protocol meets all of these requirements

Bandwidth: 1.5MB/sec

Root hub exists inside the computer

- Additional devices and hubs can be attached to the root hub
- Topology is a tree

Cable

- 4 wires: 2 data, 1 power, 1 ground

Bit representation

- 0 = voltage transition
- 1 = no voltage transition
- A 0 is defined as D- > D+, and 1 is D+ > D-, in both cases by a margin of 200mV
- String of 0's is a string of pulses
- Definition: Non-Return to Zero Inverted
- Note: a 0 is "stuffed" in long bit strings of 1's so that no more than 6 1's appear in a row

Plugging in new devices

- Root hub detects the new device
- Device is queried as to how much bandwidth it needs
- If sufficient bandwidth exists, the root hub assigns an address to the device and downloads the address and other configuration information to the I/O device
- Uninitialized devices start out with address 0

Data travel

- View the setup as a set of bit pipes between the root hub and the I/O devices
- I/O devices do not communicate with each other

Communication protocol

- Every 1ms (+/- 0.05ms) the root hub broadcasts a new frame
- A frame consists of one or more packets
- A frame addresses a single device
  - Each device gets polled according to bandwidth needs
  - A keyboard may be polled once every 50 frames (50ms)
  - A scanner may be polled often when it has data, but not at all otherwise
  - A speaker may be polled (sent data) once every 10 frames (10ms)
- There are four kinds of frames
  - Control: sending control information to a device
  - Isochronous: sending regularly timed packets to a device (no re-transmission)
  - Bulk: sending one-time information to or from a device
  - Interrupt: used for regular polling of devices

Packets

- General format
  - SYNC field: 00000001 (pulse stream)
  - PID field: packet type, 4 bits identifying the packet type, 4 bits which are the complement of the packet type for error detection
  - Data field
  - CRC field: Cyclic Redundancy Code
  - EOP field: end of packet field, both voltages are set low for a time, and then in Idle for 1 bit's time
- Token packets
  - SOF: start of frame
  - IN: command to send data from the device
  - OUT: command to receive data from the computer
  - SETUP: used for configuring a device
- Data packets
  - Can contain up to 1024 bytes of data
- Handshake packets
  - ACK: ok
  - NACK: error detected
  - STALL: device or computer is busy

Example formats

- Out: [Sync][Type = 10010110][Device address 7 bits][Port Address 4 bits][CRC][EOP]
- Data: [Sync][Type = 11000011][Data, up to 1024 bytes][CRC 16 bits][EOP]
- Handshake: [Sync][Type = 01001011][EOP]

Overall

- The computer must have a substantial amount of support software to handle USB
- The devices can be fairly simple, and may not have to do more than send ACK signals
- Each device will have a microcontroller or programmable chip to handle the USB protocol

F99 E21 Lecture #30

I/O Interfaces: PCI Bus Protocol

High-bandwidth, processor-independent bus architecture that can function as a mezzanine or peripheral bus (often there are two PCI buses, one for each purpose).

Current specification is a width of 32 or 64 bits and a clock speed of 33MHz or 66MHz, for a max raw transfer speed of 528MB/s.

Designed to be economical and requires few chips to build.

Developed by Intel in 1990, specifications and patents all released to the public domain

Current version is PCI 2.1 (1995)

Typical PCI bus struture for a desktop and a server (pp. 77)

- PCI bus is connected to a bridge/DRAM controller in the desktop system
- Mutliple PCI buses connected to a fast system bus via host bridges
- Can support different speed processors and other expansion buses

Bus structure

Required 49 pins

- System pins: includes clock and reset
- Address & Data pins: 32 lines that are time-multiplexed for addresses and data. Also includes lines used to interpret and validate the address & data lines (i.e. parity)
- Interface Control pins: control timing of interactions and provides coordination among initiators and targets
- Arbitration pins: these are not shared lines, but a pair of dedicated lines between each module and the PCI arbiter.
- Error reporting pins: used to report parity errors and other errors

Optional 51 pins

- Interrupt pins: these are not shared lines, but each PCI device has its own interrupt line to an interrupt controller
- Cache support pins: supports a memory on the PCI bus
- 64-bit extension pins: 32 additional lines that are time-multiplexed for addresses and data. Includes two lines that allow devices to agree whether to use 64 pins or 32
- ITAG/Boundary Scan Pines: allow testing procedures defined in an IEEE standard

PCI Commands

Bus activity occurs as interactions between a bus master and a target. The bus master determines the type of interaction that is going to occur.

- Interrupt acknowledge: tells the interrupt controller to place the id of the interrupt device on the data lines and specify the size of the id number
- Special cycle: used to broadcast a message to multiple modules simultaneously

- I/O read: read data from an I/O device
- I/O write: write data to an I/O device
- memory read: read data from memory
  - with cache, bursting one-half or less of a cache line
  - without cache, 2 data transfer cycles or less
- memory read line:
  - with cache, bursting more than one-half a cache line to three cache lines
  - without cache, bursting 3 to 12 data transfers
- memory read multiple
  - with cache, bursting more than 3 cache lines
  - without cache, bursting more than 12 data transfers
- memory write: transfer data in one or more cycles to memory
- memory write and invalidate: transfer data in one or more cycles to memory
  - guarantees that at least one line of cache is written
- configuration read: read device configuration parameters
  - each device can have up to 256 internal configuration registers
- configuration write: enable and update configuration parameters for devices
- dual address cycle: using 64-bit addressing (which means 2 cycles of address at 32 bits)

Data Transfers

Read transaction (bus master wants to read data from a target)

1. When the bus master has been given control of the bus, it asserts FRAME. This line remains asserted until the initiator is ready to complete the last data phase. The initiator also puts the start address on the address bus and the read command on the C/BE lines
2. At the start of clock 2, the target device will recognize its address on the AD lines
3. The initiator ceases driving the AD bus. A turnaround cycle is required on all multiply driven lines to let the bus drop. The initiator changes the information on the C/BE lines to designate which AD lines are used for transfer of the currently addressed data. The initiator also asserts IRDY to indicate it is ready for the first data item
4. After one empty clock cycle, the target asserts DEVSEL to say it has recognized its address and will respond. It places the data on the AD lines and asserts TRDY to say that valid data is present on the bus.
5. The intiator reads the data and changes the byte enable lines as needed in prepraration for the next read.
6. In this example, the target needs time to prepare the second block of data for transmission. Therefore it deasserts TRDY to signal the initiator that there will not be new data in the fifth clock cycle. It reasserts TRDY in clock cycle 5 and the data is read at the beginning of clock cycle 6.
7. At clock cycle 6, the target places the third piece of data on the AD lines, but the initiator is not ready and deasserts IRDY. Thus, the target maintains the data for another clock cycle.
8. The initiator knows this data transfer is the last, and deasserts FRAME to tell the target this is the last transfer. It then asserts IRDY to indicate it has read the third piece of data.
9. The initiator deasserts IRDY, returning the bus to an idle state, and the target deasserts TRDY and DEVSEL

A State Machine Representation

Table 1: State table for a simple I/O device

Current	Next	Condition	Action
Idle	Idle	$\overline{\text{Frame}}$	DEVSEL, TRDY deasserted
Idle	Check Address	Frame	1) check if it's my address 2) grab byte enable lines 3) Start data setup
Check Address	Idle	$\overline{\text{My address}}$	Disregard and return to idle state
Check Address	Get & Place Data	My address	1) Assert DEVSEL to acknolwedge 2) Put data on data lines 3) Assert TRDY
Get & Place Data	Hold Data	$\overline{\text{IRDY}}$	1) Increment internal address 2) Put new data on the line 3) Assert TRDY
Get & Place Data	Idle	$\overline{\text{Frame}}$	1) Deassert DEVSEL 2) Deassert TRDY 3) Deassert Data
Get & Place Data	Get & Place Data	Frame & IRDY	1) Increment internal address 2) Put new data on the line 3) Assert TRDY
Hold Data	Hold Data	$\overline{\text{IRDY}}$	1) Maintain data and TRDY
Hold Data	Place Data	IRDY	1) Maintain data and TRDY

F99 E21 Lecture #31

PCI Bus: Arbitration

PCI uses a centralized arbitration scheme with a PCI arbiter.

- Device asks for use of the bus by sending a REQ signal
- PCI controller tells a device it can use the bus by sending a GNT signal
- PCI specification does not specify an arbitration algorithm
  - first-come first served
  - round robin
  - priority queue

Example:

1. A asserts REQ signal, the arbiter samples the signal at clock cycle 1
2. During clock cycle 1, B requests the bus.
3. At the same time, the arbiter grants A's request by asserting GNT-A.
4. Bus-master A samples GNT-A at the beginning of cycle 2 and starts a transfer process by asserting FRAME. It also continues to assert REQ-A because it has multiple transfers to make.
5. The arbiter samples the REQ lines in clock cycle 3 and decides to grant the next transfer to B. It then deasserts GNT-A and asserts GNT-B. B will not be able to use the bus, however, until after FRAME and IRDY have been deasserted by A.
6. A deasserts FRAME, but still asserts IRDY to tell the target to read the data
7. When IRDY and FRAME are deasserted, B is able to take control of the bus by asserting FRAME. It also deassertes its REQ-B line since it has only one transaction to make.

This process is referred to as hidden arbitration because it takes place while other transfers are happening.

Microcontrollers

Computers were first developed in the 40's, but for 30 years they were developed using SSI and MSI components.

- 1971 an engineer at Intel (Marcian Hoff) was working on a chip for a specific application
- He developed a general purpose chip that could execute a program so that reprogramming it was easy to do
- Intel bought back the rights for the chip from the company for which they did the work
- It was later sold as the 4004, the world's first microprocessor
- It had 4096 4-bit memory locations
- It could execute 45 instructions

Later in 1971, Intel released the 8008, and 8-bit version of the 4004 with 48 instructions that could access 16kBytes of memory.

In 1973, the 8080 was released, which was 10 times faster than the 8008 and had more instructions and a larger address space.

In 1978, the 8086 and the 8088 were released, which were the basis of the ensuing PC boom.

In 1978 these were considered state of the art, high performance microprocessors. Today they would be considered slow microcontrollers.

Microcontrollers v. Microprocessors

What is a microcontroller?

- A simple computer: memory, control unit, ALU, Registers, I/O
- Generally have a single fixed program in ROM/ PROM/ EPROM/ EEPROM
- Intended to be robust, cheap solutions where some simple processing power will work

How do they differ from microprocessors?

- Limited memory
  - Usually on the chip
- Limited program size
  - Usually on the chip
- Robustness
  - reset themselves on startup and on brown-outs
- Speed
  - not slow, but usually 2 orders of magnitude slower than a microprocessor
  - 1-16MHz
- On-chip peripherals
  - A/D converters
  - USART: Universal sychronous/asynchronous receiver/transmitter
- Easy to program
  - Don’t really have an operating system, per se
- Flexible I/O pins
  - Same inputs can be A/D, input pins, output pins, or can drive a bus directly

Who is the largest manufacturer and user of microcontrollers? GM

Building a Microcontroller

Move away from a special purpose digital circuit

- Circuit should execute a fixed number of general-purpose instructions
  - How do we design a general purpose digital circuit?
  - How do we decide what instructions to allow?
  - How do we implement it?

A computer must contain six components that all work together

- Control Unit
  - State machine that specifies how the CPU is to carry out specific instructions
  - Execution is controlled by a set of control registers and the program instructions
    - Can have microcoded instructions where the control unit is it’s own state machine
    - Can use a hardwired design where the instruction contains all of the commands
  - Four important questions the control unit must answer
    - What are the operands for this instruction?
    - What is the operation that is supposed to happen?
    - Where does the operand go when its finished?

    - Where does the next instruction come from?
- ALU
    - Executes addition, logical operations, increment and decrement
    - Combinational circuit controlled by the CU
- External memory
    - If some kind of ROM, then it only holds the program to be executed
    - If some kind of RAM, then it can also hold temporary results
- Internal Memory / register file
    - register file that allows temporary data storage
    - Sequential register file controlled by the CU
- I/O interface
    - How the microprocessor/microcontroller communicates with the external world
    - Sequential/combinational circuit controlled by CU
- Internal bus
    - How data is moved around internally between registers, ALU, I/O channels
    - A set of wires connected through buffers and latches, controlled by the CU

Control-Unit: Fetch-Execute Cycle

The control unit is basically a simple state machine that moves through a set of states

- Fetch the instruction
    - Increment the program counter / load from the stack / do nothing
    - Send the instruction's address to memory so that it appears on the memory output lines
    - At the beginning of the execute cycle, latch the address into the instruction register
- Decode the instruction
    - Set control signal lines that specify bus control and ALU operations
- Read the data/operand
    - Data moves across the bus to the ALU
- Process the data
    - Data moves through the ALU
- Write the data
    - Data moves across the bus to its destination

F99 E21 Lecture #32

Microcontrollers

Control-unit follows a fetch-execute cycle

Datapath does what the control unit says

- Internal communication structure (bus)
- ALU (data processing)
- Internal registers for holding temporary data
- Memory for holding longer term data
- I/O interface

Datapath construction for a microcontroller

The datapath has to support the fetch-execute cycle

- Single bus architecture
  - All components hang off a single bus
  - Each execute cycle can move one piece of data across the bus
  - ALU has an accumulator associated with it
  - All binary operations involve the accumulator and data from the bus
  - ALU operations require multiple cycles
- Two bus architecture
  - One bus is an input bus to the ALU
  - One bus comes from the ALU
  - ALU still has a dedicated accumulator
  - All binary operations involve the accumulator and data from the bus
  - Can complete an ALU operation in a single cycle
- Three bus architecture
  - Two buses as input to the ALU
  - One bus comes from the ALU
  - ALU does not need a dedicated accumulator
  - Binary operations involve two operands from the internal register file
  - Can complete an ALU operation in a single cycle

Register Transfer Language

A method of describing motion through the data path

- Definitions
  - Registers have names [PC, IR, D1, D2, etc.]
  - Motion between registers is indicated by an arrow ->
  - Operations like +, -, shl, shr, etc. are indicated by operators
  - In an RTL, the conditions for the event happening are given before the operation
    - e.g. K1: D1 -> BusA

A register transfer language helps us to understand what is going on at a dataflow level and relate that to control signals.

Example: PIC Microcontroller Architecture

- One-bus architecture
  - ALU has a dedicated register, called Wreg
  - All binary operations involve Wreg
  - The other argument can come from the instruction or from the register file
  - Arithmetic results can go to either Wreg, or back to the location of the other operand
  - I/O devices hang on the same bus
    - I/O registers are memory mapped: they are addresses just like the internal register file
- Address mode: direct or indirect
  - Address can come from the instruction: direct addressing
  - Address can come from the FSR register (File Select Register): indirect addressing
    - Indirect addressing treats the FSR register as an index into an array of registers
- Execute Cycle
  - Clock is divided into four signals Q1 - Q4
  - Q1: instruction is latched into the instruction register
    - Address is applied to the RAM or I/O units, if necessary
  - Q2: operands are latched into the ALU
  - Q3: ALU processes the data and the result appears on the bus
  - Q4: Data is latched into its destination
- Fetch Cycle
  - Fetch cycle is executed in parallel with the Execute cycle

F99 E21 Lecture #33

Microcontrollers

Review: Datapath

- Number of buses
- ALU complexity
- Address modes
  - Immediate: operand in the instruction
  - Direct: address of operand in the instruction
  - Indirect: address of address of operand in the instruction

The Fetch Cycle

The control unit follows a simple set of states

- In a modern microprocessor, the control unit is extremely complex
  - Different instructions can take different numbers of states
  - The CPU can execute multiple instructions simultaneously
  - The CPU may execute instructions that it never needs to use
- In a microcontroller, the control unit is generally quite simple
  - The execution may always take the same number of states
  - The fetch may always take the same number of states
  - This simplicity is largely due to the simplicity of the architecture

The control unit has a set of registers that allow it to evaluate instructions and fetch the next instruction

- Instruction Register [IR]: this register holds the current instruction
- Program Counter [PC]: this register holds the address of the next instruction
- Status Register [STATUS]: this register holds various flags necesary for operation, including flags describing the result of the last ALU operation
- Stack: Set of registers (8 on the PIC) that hold PC values in order to facilitate branches to subroutines and returns from subroutines

The fetch cycle is quite simple

- The PC holds the address of the next instruction
- At the appropriate time, this address accesses the program memory and the next instruction is loaded into the IR
- The IR is then decoded and controls the actions of the execute cycle
- There may be other state variables
  - On the PIC, the other state variables are realized by the Q1-Q4 signals

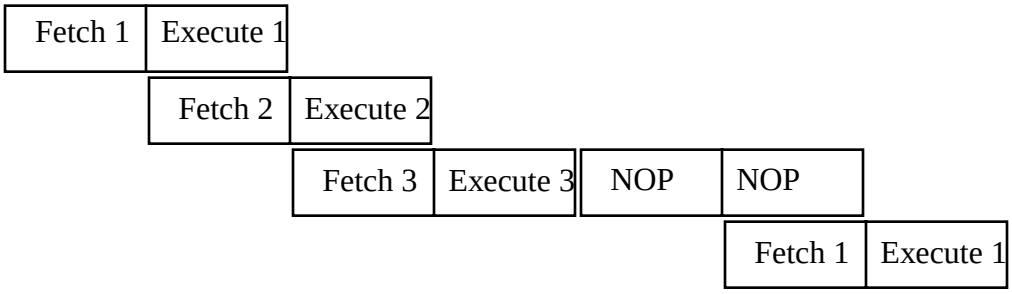
Pipelining

Pipelining is, in effect, an assembly line inside the computer

- Split the fetch-execute cycle into smaller subdivisions
- Develop a datapath that can execute the smaller subdivisions in parallel
- Once the pipeline is full, you are executing more instructions per second, even though each instruction takes the same amount of time as it did before

Example: PIC architecture

- PIC is an example of simple pipelining
- Fetch cycle for the next instruction is executed in parallel with the current execute phase
- The exception is a branching statement, which may take two cycles to complete
- Conditional branch only takes 1 cycle if it doesn't branch
- Unconditional branch always takes 2 cycles



Harvard v. Von Neumann Architecture

A final issue in designing a microcontroller is where you put the program and where you put the data. Do you put them in the same place?

Von Neumann architecture:

- Program and data are located in the same memory space (physical and logical)
- Generally this is RAM memory
  - For a microcontroller, this means the RAM must be loaded from a ROM at startup
- Von Neumann architecture is extremely general
- Programs can be self-modifying (which can be interesting, and leads to viruses, etc..)
- This puts constraints on the size of your instruction
  - Must be some multiple of the basic data element, which is generally 8 bits (1 byte)
- More difficult to pipeline the fetch and execution cycles, but it is possible

Harvard architecture

- Program and data are located in different memory spaces (physical and logical)
- Program memory does not have to be RAM
  - If the program memory is ROM, then everything can be on one chip
- The instruction size is not constrained by the basic data unit
- The fetch and execute cycles can be executed in parallel, because there are separate buses

Examples: 6811, and PIC

6811: Memory and data spaces are the same

- Not a pipelined architecture
- Requires off-chip ROM and RAM in order to function

PIC: Harvard architecture

- Program memory is EPROM
- Doesn't require any off-chip modules to function
- Pipelined architecture

Example: PIC Microcontroller Complete Cycle

Fetch and execute cycles are pipelined

Fetch cycle

- PC is incremented in Q1
- Instruction register is latched in Q4

Execute cycle (standard instruction)

- Instruction is decoded, and control lines are set in Q1
- Operands are fetched in Q2
- ALU execution takes place in Q3
- Operands are latched on Q4

Branch instructions

- Skip if X
  - The condition X is tested in Q2
  - If the condition is true, then a NOP is forced in the next cycle
  - The PC gets incremented a second time
  - Instruction then resumes as usual
- Unconditional branch
  - The PC is loaded with a new value in Q2/Q3
  - The instruction fetched in Q4 is ignored
  - The following execute cycle is a NOP cycle
  - The instruction from the branch address is loaded in the following Q4
  - How slow can your EPROM memory be? at 20MHz, 2 cycles = 100ns

F99 E21 Lecture #34

Microcontrollers

Review:

- Control Unit
  - 13-bit PC
  - 14-bit instructions on 16CXXX series
- Pipelining
- Harvard v. Von Neumann Architecture

Instruction Formats

Somehow you need to indicate to the computer what it is supposed to do

- Operation
- Operands
- Destination

Horizontal format

- There is one bit in the instruction for each control line in the computer
  - This makes the control unit fast and easy
  - It makes the instructions very long

Vertical format

- Encode the control lines in appropriate groups
  - Each group will have a decoder associated with it
  - The control unit is more complex, but the instruction is more compact

How many addresses?

- In part depends upon how many buses you have
  - More buses => more possible operands and destinations
- In part depends upon how long your instructions can be
  - Longer instruction => more potential addresses
- In part depends upon how many instructions you have
  - More instructions => more possible addressing modes and numbers of addresses

The **architecture** of the computer is the set of instructions it can execute and the RTL describing what each instruction does.

Knowing an **architecture** means you can write a program for the computer

The **architecture** is independent of the **organization,** which is how the instructions are actually implemented.

- Different PIC chips have the same architecture
- Therefore, most programs will run on several different kinds of PICs

Example: PIC Microcontroller Instruction Set

14-bit instructions

- Byte-oriented file register operations
  - [opcode (6 bits)][destination bit (0,1)][File # (7 bits]

- Bit-oriented file register operations
  - [opcode (4 bits)][bit # (3 bits)][File # (7 bits)]
- Literal and control operators
  - [opcode (6 bits)][literal (8 bits)]
- Call and GOTO operations
  - [opcode (3 bits)][literal address (11 bits)]
  - High 2 bits for the address come from the PCLATH register
  - This means to make some jumps you have to use two instructions
    - Move high two bits to PCLATH
    - GOTO/CALL operation

These four types of operations are differentiated by the first two bits

- [00]: byte register operations or one of six special ops
- [01]: bit oriented register operations
- [10]: Call or goto
- [11]: Operations with literals

- All 0's in the first 6 bits indicates one of these six special instructions
  - NOP (all 0's, but don't care for bits 5:6)
  - MOVWF: move Wreg to register F (no ALU op)
  - RETFIE: return from interrupt
  - CLRWDT: clear watchdog timer
  - RETURN: return from subroutine
  - SLEEP: sleep

Designing Flexible I/O Devices

A/D and input/output port

- PortA: RA5, RA3:RA0 pins
  - configurable as inputs or outputs
  - configurable as digital or analog inputs
  - open wire to the A/D converter
  - AND gate to input latch, latch is input enabled on reads only
  - Logic connects output latch and register control latch to ouput CMOS drivers
- PortA: RA4 pin
  - configurable as digital input or digital output
  - Schmitt Trigger input buffer isolates input and peripheral module from the pin
  - Open drain means the output can be connected in a wired-OR fashion (negative logic)
  - Bus floats high unless something is pulling it down
- PortB: RB3:RB0
  - Has a weak pull-up feature, which means if the output pin is not connected to ground it will float high (if configured as an output
  - Output goes through a tri-state latch (TTL driver)
- PortB: RB7:RB4
  - Also has a weak pull-up
  - Allows interrupt on change feature (can use it to wake up the processor when someone hits a key, for example)

    - Old value of the input from the last read cycle is compared with the current value to see if there has been any change.
    - This only works if the port is configured as a read port
- PortC: RC7:RC0
    - Can act as inputs or outputs
    - Can select between a peripheral input/output or the port C input/output
    - Peripheral input goes through a Schmitt trigger to isolate it from the pin
    - CMOS output driver
- PortD: RD7:RD0
    - Simple configurable TTL input/output pins

F99 E21 Lecture #35

Microcontrollers

Peripheral Control

Each peripheral is a state machine that is set by moving information to and from registers

Peripherals can communicate asynchronously with the processor through the use of interrupts

Interrupts

What are they?

- A method that an asynchronous process can communicate with the processor
- I/O and A/D processes are slow, for example, compared to the processor speed

How are they implemented?

- An interrupt bit is set when a device signals for an interrupt
- At the beginning of each instruction cycle, the internal logic tests the interrupt bit
- The processor saves its current instruction on the stack and disables interrupts
- The processor then enters an interrupt service routine
- The processor polls the interrupt flags to find the device that caused the interrupt
- The processor takes care of the situation
- The processor then returns to the instruction just after the interrupt occurred
- Interrupts are re-enabled on the return

On the PIC the interrupts all come into a single line

- OR gate tree with two levels
- The GIE (Global Interrupt Enable) bit is ANDed with the Interrupt input line
- The interrupt service routine must poll the interrupt flags to see what caused the interrupt
- More than one interrupt could happen at the same time, so the polling order is important

Process specifics

- INTF flag is sampled every Q1
- On the Q1 in which it is high, the instruction currently being executed is allowed to finish
  - The instruction being fetched is not executed on the following cycle, which is a NOP
  - The current value of the PC is pushed onto the stack
- On the second cycle after the INTF flag is sampled high 0x0004 is loaded into the PC
- On the third cycle after the INTF flag is sampled high, the first instruction of the interrupt service routine is executed
  - LOOP: BCF INTCON, GIE --Disable the GIE flag
  - BTFSC INTCON, GIE --test if it is still disabled
  - GOTO LOOP --loop back to BCF if not disabled
  - MOVWF W_TEMP --save W register
  - MOVF STATUS, 0 --swap STATUS and W registers
  - BCF STATUS, RP0 --change to memory bank 0
  - MOVWF STATUS_TEMP --save STATUS register
  - ...Interrupt service routine goes here

- MOVF STATUS_TEMP, 0 --get old value of STATUS register
- MOVWF STATUS --put it back in the STATUS register
- MOVF W_TEMP --get the old W register value

Note, the handout is slightly strange in its use of the SWAPF call

- This may be assembly language, but it's bad usage

Also note, W_TEMP must be defined in both "banks" of memory since the current value of the RP0 bit could be either 0 or 1 in STATUS

A/D conversion using Interrupts

- Converter uses three registers to control operation, plus the A/D interrupt flag and the A/D interrupt enable bit
  - ADCON0: controls the operation of the converter
    - Bits 7:6 control the sampling frequency by specifying a divider on the internal clock
    - Bits 5:3 specify which line to read
    - Bit 2 is how you tell the A/D converter to begin, it resets the bit when it is finished
    - Bit 1 is reserved
    - Bit 0 turns the A/D converter on (1) or shuts it off so it doesn't draw current (0)
  - ADCON1: specifies the functionality of the A/D pins
    - Bits 7:3 not used
    - Bits 2:0 specify which pins are Analog, which pins are Digital, and whether to use an external reference instead of the internal reference voltage source
  - ADRES: Holds the result of the conversion
- Process
  - Configure the A/D module
    - Configure the converter pins (ADCON1)
    - Select the input channel (ADCON0)
    - Select the input clock (ADCON0)
    - Turn on the A/D module (ADCON0)
  - Configure the A/D interrupt
    - Clear the ADIF bit
    - Set the ADIE bit (enables or disables interrupts)
    - Set the GIE bit (enables or disables all interrupts)
  - Wait the required acquisition time so the A/D capacitor can charge
  - Start the conversion (ADCON0)
  - Wait for the conversion to finish
    - Poll the GO/$\overline{\text{DONE}}$ bit
    - Wait for the interrupt
  - Read the A/D result register
    - Clear the ADIF bit, if required

Wrap-up

Course evaluation

Homeworks and labs back on Monday/Tuesday

Hope you enjoyed the class