E21: Digital System Design Lectures, Fall 2000

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

F00 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
- By the end you should have the knowledge necessary to begin designing computer peripherals and stand-alone digital devices.

Final project

- The last 4-5 weeks you will be doing your own final project, start thinking about it now
- Past years:
  - keyed security system, and it worked
  - evolvable hardware project
  - parallel port communication & control of external devices

    - lap counter
- 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 will 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 (strongly recommended): 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

Electrical implementations of truth tables

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

Table 1: Truth Tables

Input A	Input B	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Example question: So if I wanted to add two 1-bit numbers, what would be appropriate logic gates to generate their sum and carry? [sum = XOR, carry = AND]

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.

Note: mainframes and supercomputers used to be built out of SSI components. Cray designed the Cray-1 from PCBs with nothing but NAND gates on them because they were fast.

F00 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)
  - Each position in a binary system is a power of 2 (starting with 2^0 on the right)
  - Conversion from decimal to binary: keep dividing by 2, put the remainders right to left
    - Example: 26 -> binary = 11010
  - Conversion from binary to decimal: sum up the value of each place containing a one
    - Example 101010 = 2 + 8 + 32 = 42
- 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
- Hexadecimal values are (0-9, A-F)
- Convert 61453 (decimal) to hex -> F00D
  - 61453 / 2 = 30726 (1)
  - 30726 / 2 = 15363 (01)
  - 15363 / 2 = 7681 (101)
  - 7681 / 2 = 3840 (1101)
  - 3840 / 2 = 1920 (0 1101)
  - 1920 / 2 = 960 (00 1101)
  - 960 / 2 = 480 (000 1101)
  - 480 / 2 = 240 (0000 1101)
  - 240 / 2 = 120 (0 0000 1101)
  - 120 / 2 = 60 (00 0000 1101)
  - 60 / 2 = 30 (000 0000 1101)
  - 30 / 2 = 15 (0000 0000 1101)
  - 15 / 2 = 7 (1 0000 0000 1101)
  - 7 / 2 = 3 (11 0000 0000 1101)
  - 3 / 2 = 1 (111 0000 0000 1101)
  - 1 / 2 = 0 (1111 0000 0000 1101) = (F 0 0 D)

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

F00 E21 Lecture #3

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, 7, 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

Example:

1-information bit => 2 code bits

- Possible data values are 0/1
- Code bits must make even parity, so 000 / 111 are the legal codes
- A single bit flip can be corrected

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'

Standard Representations

SOP representation: all of the 1 combinations

POS representation: all of the 0 combinations, but invert variables and use product of sums

Example: F = Sum(1, 2, 4, 6, 7)

- SOP = $\overline{A}\ \overline{B}\ C + \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + A\ B\ \overline{C} + A\ B\ C$
- POS = $(A+B+C)(A+\overline{B}+\overline{C})(\overline{A}+B+\overline{C})$

Question: can we do it better?

Minimization using Boolean logic...

- SOP expression: Combine the last two terms into AB, combine 2nd and 4th to get B$\overline{C}$, combine 4th and 5th term to get A$\overline{C}$.

Have everyone do F = Sum(0, 2, 6, 7)

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

F00 E21 Lecture #4

Designing Simple Logic Circuitsi

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
- Static Hazards
  - Can occur when you go between disconnected groups in a K-map
  - One solution is to have overlapping groups (more terms than you really need)
  - Other solution is to use sequential logic that only samples during stable periods

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

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, maybe 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: F = Sum(2, 3, 5, 6, 7)]
  - Follow the procedure to create groupings at the highest level possible

- Each non-covered term is a prime implicant
- Create a second table with the prime implicants
- Pick the terms you need from the table (minimal cover generation)

Group ID	Subcube Minterms	Subcube Value	Subcube Covered
G1	(2)	010	yes
G2	(3, 5, 6)	011 101 110	yes yes yes
G3	(7)	111	yes
G1	(2, 3) (2, 6)	01X X10	yes yes
G2	(3, 7) (5, 7) (6, 7)	X11 1X1 11X	yes no yes
G1	(2, 3, 6, 7)	X1X	no

Prime Implicant	Expression	Minterms	2	3	5	6	7
P1	AC	(5, 7)			**X**		X
P2	B	(2, 3, 6, 7)	**X**	**X**		**X**	X

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

F00 E21 Lecture #5

Designing Simple Logic Circuits

Minimal Cover Generation with Quine-McClusky

1. Find the Essential Prime Implicants [EPI]
   - 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

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

Group ID	Subcube Minterms	Subcube Value	Subcube Covered?
G0	(0)	000	yes
G1	(2)	0010	yes
	(4)	0100	yes
G2	(3)	0011	yes
	(6)	0110	yes
	(9)	1001	yes
G3	(7)	0111	yes
	(11)	1011	yes
	(13)	1101	yes
G4	(15)	1111	yes
G0	(0, 2)	00X0	yes
	(0, 4)	0X00	yes
G1	(2, 3)	001X	yes
	(2, 6)	0X10	yes
	(4, 6)	01X0	yes
G2	(3, 7)	0X11	yes
	(3, 11)	X011	yes
	(6, 7)	011X	yes
	(9, 11)	10X1	yes
	(9, 13)	1X01	yes
G3	(7, 15)	X111	yes
	(11, 15)	1X11	yes
	(13, 15)	11X1	yes

Group ID	Subcube Minterms	Subcube Value	Subcube Covered?
G0	(0, 2, 4, 6)	0XX0	no
G1	(2, 3, 6, 7)	0X1X	no
G2	(3, 7, 11, 15)	XX11 1XX1	no no

Prime	Expression	Minterms	0	2	3	4	6	7	9	11	13	15
P1	$\overline{A}\,\overline{D}$	(0, 2, 4, 6)	**X**	X		**X**	X					
P2	$\overline{A}$ C	(2, 3, 6, 7)		X	X		X	X				
P3	CD	(3,7,11,15)			X			X		X		X
P4	AD	(9,11,13,15)							**X**	X	**X**	X

- P1 and P4 have to be in the expression (essential prime implicants)
- Shaded columns are the only ones not covered by P1 or P4
- Can pick either P2 or P3 to complete the expression

Example of SOP method

Consider the function: F = Sum(2, 6, 7, 8, 9, 13, 15)

- Prime implicants are: $\overline{A}$ C $\overline{D}$, $\overline{B}\,\overline{C}$ D, $\overline{A}$ B C, A$\overline{B}$D, BCD, ACD^{-}

Prime	Expression	Minterms	2	6	7	8	9	13	15
P1	$\overline{A}$ C $\overline{D}$	(2, 6)	**X**	X					
P2	$\overline{B}\,\overline{C}$ D	(8, 9)				**X**	X		
P3	$\overline{A}$ B C	(6, 7)		X	X				
P4	A$\overline{B}$D	(9, 13)					X	X	
P5	BCD	(7, 15)			X				X
P6	ACD	(13, 15)						X	X

- Build the POS expression for each column not covered by an EPI (shaded columns)
- F = (P3 + P5)(P4 + P6)(P5 + P6)
- F = (P3 + P5)(P4P5 + P5P6 + P4P6 + P6): P6 covers P5P6 and P4P6
- F = (P3 + P5)(P4P5 + P6)
- F = P3P4P5 + P3P6 + P4P5 + P5P6: P4P5 covers P3P4P5
- F = P3P6 + P4P5 + P5P6
- Choose any of these three combinations to minimally cover the circuit

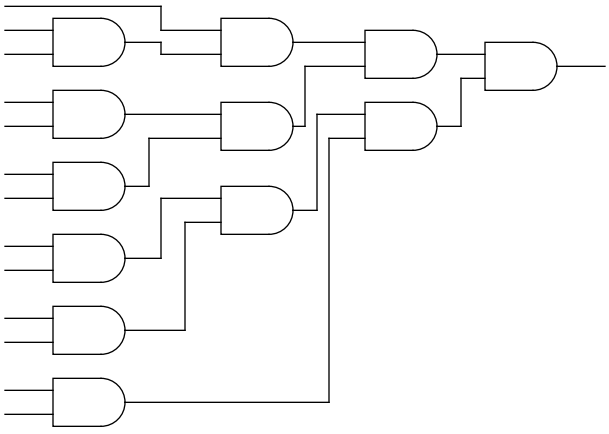
Technology Mapping

- 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
  - Rule 4: x+y = ((x + y)')': OR gate is a NOR gate with an inverter on the output
- 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 (sum)
  - 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
  - Example: Convert a 13 input AND gate to 2-input AND gates

Level	Inputs	#gates
1	13	6
2	7	3
3	4	2
4	2	1



- Timing optimization
  - There are often multiple different decompositions you can design
  - Pick the one that optimizes timing
    - minimize the longest path through the circuit
  - Pick the one that reduces the difference between the shortest and longest paths
    - minimizes the likelihood of hazards

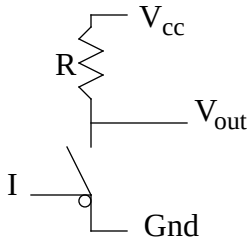
F00 E21 Lecture #6

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
- 1970's: Development of the single-chip microprocessor
- 1980's: CMOS development, CMOS replaces TTL logic
- 1990's: 1-10 million transistor ICs
- 2000's: 100-200 million transistor ICs

The Basic Inverter



To create an inverter you need a switch (any switch will do)

- The incoming signal controls the switch
- The outgoing signal is determined by how the switch is connected
  - Pull-up network: circuit is usually low, closing the switch pulls the output high
  - Pull-down network: circuit is usually high, closing the switch pulls the output low

Better inverters are faster because they pull the circuit high or low more quickly

Better inverters use less power because they have higher resistance to the disconnected voltage

The ideal inverter would have an instantaneous switch on boths sides:

- low source resistance to the connected power level
- high resistance to the disconnected power level

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

Low input voltage for TTL is 0-0.8V, high input voltage is 2.0-5.0V, 0.8-2.0 is undefined

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)

Basic AND and OR gates

- OR gate: put two transistors in parallel to ground with each input connected to one transistor
- AND gate: put two tarnsistors in series to ground with each input connected to one transistor

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

CMOS: Complementary metal oxide semiconductor

- CMOS logic levels: traditionally 0-1.5V = low, 3.5-5V = high, 1.5-3.5 = undefined
- Other power levels are used as well: 3.3V, 2.7V, or lower for large ICs
- Typical static power dissapation for an SSI gate is 0.0125mW (12.5 uW)
  - Typical dynamic power dissapation is 0.4mW/MHz (4mW at 10MHz)
  - Power consumption is dependent upon voltage and frequency $P = C_{PD}V_{CC}^2f$
  - C_{PD} = Power dissapation capacitance, a characteristic of the transistors
  - V = Power supply voltage, f = switching frequency

- Typical delays for an SSI gate are 5-10ns
- 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

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

F00 E21 Lecture #7

Logic Families

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

Fanout

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

- Estimate: maximum LOW-state output current = 20uA, max input current = +/- 1uA

How much current are you going to draw from the circuit?

- Look at an inverter and treat the NMOS and PMOS transistors as appropriate resistances

- Model the load as a resistive load
  - Thevinin equivalent of a 2k/1k voltage divider is a 667 ohm resistor and a 3.3 V source
  - Case one (PMOS off, NMOS = 100 ohms): low output, .43 V (sinking current)
  - Case two (PMOS = 100 ohms, NMOS off): high output, 4.78 V (sourcing current)
- Note: non-ideal inputs can cause problems because the transistor resistances change
- 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

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
  - AND/NAND gates: tie the free input high
  - OR/NOR gates: tie the free input low

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)

- 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

ECL: Emitter Coupled Logic

Emitter coupled logic has always been the fastest logic available

- Take two bipolar transistors and connect their emitters
- Put the inputs to the transitor bases
- Each collector has its own resistor to a source voltage
- Basically a differential amplifier circuit

Emitter coupled logic works by changing currents

- The voltage change is very small (0.8V difference between high and low)
- They switch between two current flow states, off and on
- By switching the current, they switch the voltage through the collector resistors

Typical logic valuse are: 4.2 (low) and 5.0 (high)

ECL logic has always been the fastest available

- Current ECL family (ECLinPS) offers maximum delays under 0.5ns, including the signal delay getting on and off the chip (2GHz)
- Useful for communications gear, fiber-optic interfaces, and gigabit ethernet

ECL uses a lot of power

- Current is always flowing
- 26mW per gate, plus from 10-150mW per gate for termination

F00 E21 Lecture #8

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 combinational logic using asynchronous assignment statements

Say we want to implement our own XOR gate. In VHDL we just describe the function of the combination circuit just like we were writing a Boolean expression

```
entity myxor is
      port(A, B: in std_logic;
            F: out std_logic);
end myxor

architecture dataflow1 of myxor is
      -- this implements an xor gate
      F <= (A and (not B)) or ((not A) and B)
end dataflow1;

architecture dataflow2 of myxor is
      ar1, ar2: std_logic;

      ar1 <= A and (not B);
      ar2 <= (not A) and B;
      F = ar1 or ar2;
end dataflow2;
```

Designing combinational circuits using a process statement

In a process statement we can write expressions like if statements, which makes coding some combinational logic easer.

```
entity FourToTwo is
      port(inLines: in std_logic_vector(0 to 3); -- eight bits of input
            outLines: out std_logic_vector(0 to 1); -- three output bits
```

```
            inactive: out std_logic);
end eightToThree;

architecture behavioral of FourToTwo is

      process(inLines) begin
            if inLines(0) then
                  outLines <= “00”;
                  inactive <= “0”;
            elsif inLines(1) then
                  outLines <= “01”;
                  inactive <= “0’:
            elsif inLines(2) then
                  outLines <= “10”;
                  inactive <= “0”;
            elsif inLines(3) then
                  outLines <= “11”;
                  inactive <= “0”;
            else
                  outLines <= “000”;
                  inactive <= “1”;
            end if;
      end process;
end behavioral;
```

F00 E21 Lecture #9

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)
- Conceptually, the range labels the bits left to right

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.

MSI design components are functions that have prove to be useful in the design of digital circuits

- Basic building blocks for interface circuits, control logic, and medium complexity circuits

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 is an additional input to each AND gate
- Netlist representation
  - Entity and port statement

- component declarations in the declaration section of the architecture
  - note: you have to write the definitions for inv and and3 as other VHDL files
- intermediate signal declarations
- port map statementes to instantiate each gate

```
library IEEE
use ieee.std_logic_1164.all

entity V2to4dec is
      port ( I0, I1, EN; in std_logic;
             Y0, Y1, Y2, Y3: out std_logic);
end V2to4dec;

architecture structural of V2to4dec is
      signal invI0, invI0: std_logic;
      component inv port(I: in std_logic; F: out std_logic);
            end component;
      component and3 port(a, b, c: in std_logic; F: out std_logic);
            end component;
begin
      U1: inv port map (I0, invI0);
      U2: inv port map (I1, invI1);
      U3: and3 port map (invI0, invI1, EN, Y0);
      U4: and3 port map (   I0, invI1, EN, Y0);
      U5: and3 port map (invI0,    I1, EN, Y0);
      U6: and3 port map (   I0,    I1, EN, Y0);
end V2to4dec;
```

- Dataflow representation 1
  - Use the concurrent **with** statement to determine the outputs
  - Change the port statement to use a std_logic_vector on the inputs and outputs
  - Note the use of **when others** to guarantee there are only 4 cases.

```
entity V2to4dec is
      port ( EN: in std_logic;
             I: in std_logic_vector(0 to 1);
             Y: out std_logic_vector(0 to 3));
end V2to4dec;

architectures dataflow of V2to4dec is
      signal Yi: std_logic_vector(0 to 3);
begin
      with I select Yi <=
            “1000” when “00”,
            “0100” when “01”,
            “0010” when “10”,
            “0001” when others; -- simplifies the logic by using only 4 choices
      Y <= Yi when EN = ’1’ else “0000”;
end dataflow;
```

- Dataflow representation 2 (hierarchical)
  - Say you want an easy way to control whether the output logic is active high or active low

    - create a V2to4dec entity which uses active high logic (as above)
    - Use a port map statement to insert it into an entity whose inputs or outputs are active low
    - Control the logic level switching at the higher level.
- Behavioral style representation 1
    - Use a process, with a case statement and an if statement
    - Note the use of when others to limit the number of cases to 4
    - Note the inclusion of Yi in the sensitivity arguments to the process statement
        - The variables inside a process don't get their assigments until the process completes
        - Therefore, it takes two passes through the process to get Yi assigned to Y, and we have to make sure and put Yi in the sensitivity list so that the process gets called twice
        - Could make the if statement a concurrent conditional assignment and accomplish the same thing without having to include Yi in the process sensitivity list.

```
architecture behavioral1 of V2to4dec is
      signal Yi: std_logic_vector(0 to 3);
begin
      process (EN, I, Yi) begin
            case I is
                  when “00” => Yi <= “1000”;
                  when “01” => Yi <= “0100”;
                  when “10” => Yi <= “0010”;
                  when others => Yi <= “0001”;
            end case
            if EN = ‘1’ then
                  Y <= Yi;
            else
                  Y <= “0000”;
            end if;
      end process;
end behavioral1;
```

F00 E21 Lecture #10

Combinational Logic Elements

Continue examination of how to design and work with MSI elements

Decoder

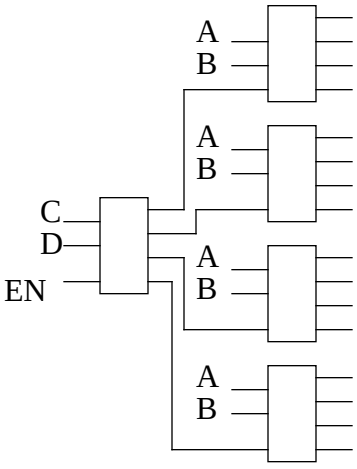
Given the entity declaration:

```
entity V2to4dec is
      port ( EN: in std_logic;
             I: in std_logic_vector(0 to 1);
             Y: out std_logic_vector(0 to 3));
end V2to4dec;
```

We can now complete our examination of the ways to represent the function

- Behavioral style representation 2
  - Go all out and write an algorithm
  - Note that each bit is only written once within the process (Altera requirement)

```
architecture behavioral2 of V2to4dec is
begin
      process (EN, I)
            variable i: integer range 0 to 7;
      begin
            if EN = '1' then
                  for i in 0 to 7 loop
                        if i = conv_integer(A) then
                              Y(i) <= '1';
                        else
                              Y(i) <= '0';
                        end if;
                  end loop;
            else
                  Y <= "0000";
            end if;
      end process;
end behavioral2;
```



What if we want a larger decoder?

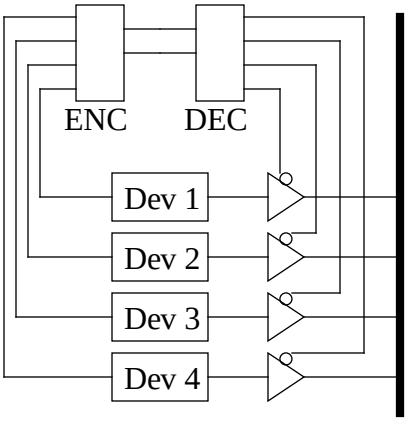
- Cascade them using enable inputs
  - If you have an N-output decoder, use the enable inputs to get a 2N-output decoder
  - If you have an N-output decoder, use a 2-4 decoder to get a 4N-output decoder

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

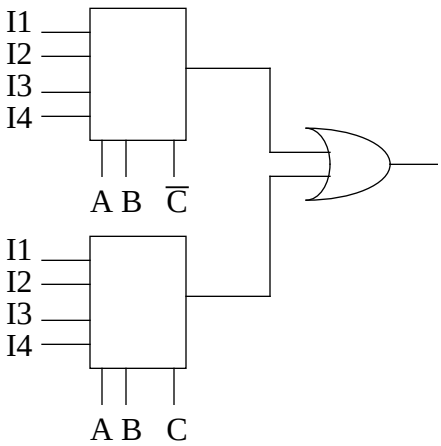
Note: with an encoder, decoder, and tri-state buffer we can start to make a bus and bus controller



- A bus is a set of wires that a number of devices may want to use
- Let the encoder handle all requests for the bus
- Send the output to a decoder that controls a set of tri-state buffers
- The tri-state buffers control what goes on the bus

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 big MUXs by cascading them using an 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
  - It is generally implemented as such when synthesized to hardware
  - This is why you want to make your case statements powers of 2, if possible
- 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

This is an alternative bus arrangement

- MUX controls what signal gets on the bus wire
- DEMUX controls where the signal goes at the other end
- Somehow have to decide who gets to send the signal and who gets to receive it

F00 E21 Lecture #11

Combinational Logic Elements

More complex combinational logic elements

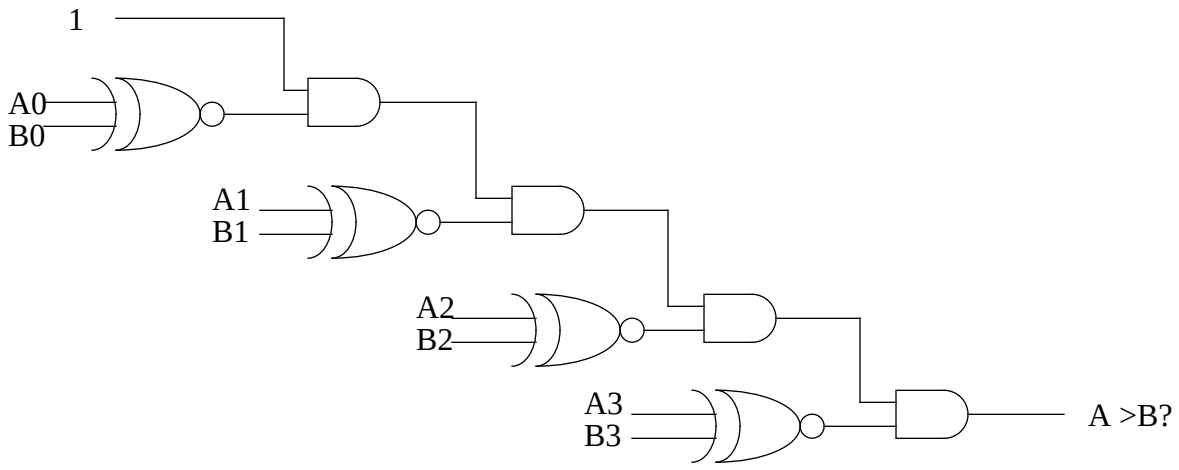
Comparators

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

- To compare multiple bits, we could cascade XOR gates and AND gates (see below)
  - This is the way Altera would implement a for loop: unfolding the logic
- 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

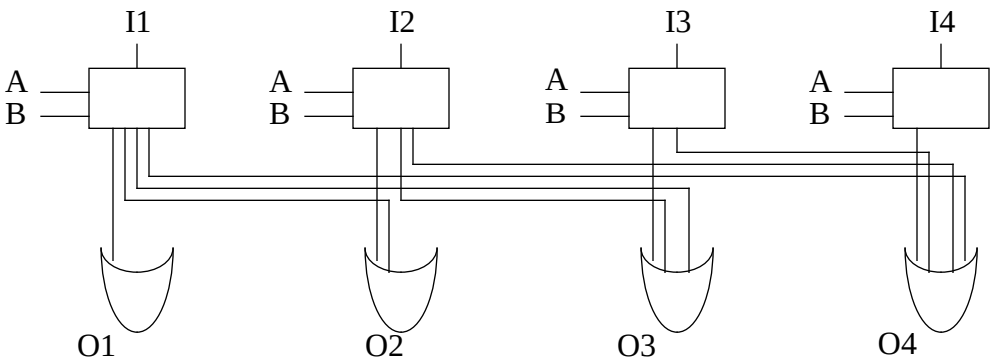
- Greater-than: start with the MSB and work down
- Less-than: start with MSB and work down
- Real implementations use one AND gate for each case and then an OR gate to see if any of the potential greater than/less than situations are true



Shifters/Rotators

We can use DEMUXs on the inputs to generate a shifter/rotator circuit (right shifter shown below)

- 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
- S = A XOR B XOR Cin
- Cout = AB + BCin + ACin

Carry Lookahead Adders

Ripple-adders are slow: signal has to propagate

- Difficulty is the carry bit: use a circuit to precalculate the carry bits
  - A particular full adder will generate a carry if its two addend bits are 1 (A,B)
  - A particular full adder will propagate a carry if either of its two addend bits are 1
- At each level $C_{i+1} = G_i + P_i * C_i$
  - $G_i = X_i * Y_i$: generate term
  - $P_i = X_i + Y_i$: propagate term
- We can iteratively expand this to get a two-level logic expression for the ith carry
  - C1 = G0 + P0 * C0
  - C2 = G1 + P1 * C1 = G1 + P1 * G0 + P1 * P0 * C0
  - C3 = G2 + P2 * C2
    C3 = G2 + P2 * (G1 + P1 * G0 + P1 * P0 * C0)
    C3 = G2 + P2 * G1 + P2 * P1 * G0 + P2 * P1 * P0 * C0
  - C4 = G3 + P3 * C3
    C4 = G3 + P3 * (G2 + P2 * G1 + P2 * P1 * G0 + P2 * P1 * P0 * C0)
    C4 = G3 + P3 * G2 + P3 * P2 * G1 + P3 * P2 * P1 * G0 + P3 * P2 * P1 * P0 * C0
- Each 4-bit adder circuit only takes 3 levels of logic to calculate the last carry
- Full binary adder is on pp. 437 of Wakerly
  - Note the use of INVERT-OR-AND logic, which is the dual of AND-OR-INVERT
  - This logic can be implemented in two layers of transistors, so it is as fast as one gate
  - Carry equations are slightly different: Ci+1 = Pi * Gi + Pi * Ci
  - Same equation, though, because Pi is always 1 if Gi is 1
  - Allows factoring: Ci+1 = Pi * (Gi + Ci) (this is source of the INVERT-OR-AND logic)

When you build large (i.e. 32-bit) 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

ALUs generally have 8, 16, or 32 functions they can execute

- They will have carry in and carry out lines so that multiple ALU's can be cascaded to handle more bits of data

F00 E21 Lecture #12

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 (no fuses, fixed number of product terms)
- Outputs can be inputs of one block can be inputs to another block
- AND gates can connect to any input or the output of another block
- Output pins are bi-directional: one AND gate controls the direction of the pin
  - Pin has a tri-state buffer at the output of the OR gate
  - If the tri-state buffer is enable, the pin is an output
  - Otherwise, it is an input since the OR gate does not connect to the wire
- Diagram on pp. 341 of Wakerly

Generic Array Logic: GAL

- EEPROM controls the fuses
- Output polarity can be controlled by an XOR gate on the output of each main OR gate
  - fuse on the second input to the XOR gate which connects to ground
  - if the fuse is blown, the XOR gate inverts the OR gate output
- Can be set up to do some sequntial logic as well by encoding a flip-flop out of NOR gates

Large Scale PLDs, of Complex PLD [CPLD]

- Altera MAX7000 family: thousands of logic gates in blocks, or macrocells
  - The chip number indicates how many macrocells there are (e.g. 7128 has 128 cells)
- Electrically programmable
- Electrically erasable
- A “fitter” puts the designs on the chip

MAX 7000 family is built from a set of macrocells that each connect to I/O pins and to a programmable interconnect array [PIA].

- Each macrocell is a PAL device with an output flip-flop that can be used or bypassed
- A macrocell can send product terms to other macrocells, so that a single OR gate can have up to 20 product terms in it
- The output of the OR gate goes through an XOR gate so that the output can be inverted

There are several fixed inputs to a MAX7000 that are globally distributed

- Two clock inputs
- One reset input

Field Programmable Gate Arrays [FPGA]

These are more complex, general programmable logic devices than CPLDs, and are designed so that state machines and arithmetic logic units are easy to configure.

FLEX10k family

- From 10,000 to 250,000 gate equivalents
- Consist of Logic Array Blocks [LAB] and Embedded Array Blocks [EAB]
  - LABs consist of eight four-input lookup tables, each with a flip-flop on the output
  - LABs are cascaded to they can be joined together to form full-adders, etc.
  - EABs consist of 2048 bits of memory that can be configured in different ways
  - EAB inputs and outputs can come from or go to a flip-flop
- All of the LABs and EABs are connected by row/column buses
- Each LAB/EAB can route information between its adjacent rows and columns

Xilinx XC4000 family

- From 2000 to 250,000 gate equivalents
- XC4000 is a 2-D array of configurable logic blocks [CLB]
  - Each CLB contains 2 four-input lookup tables
  - Each CLB has a flip-flop on the output
  - Each CLB also has a 3-input LUT connected to the 2 four-input LUTs, which allows functions of 5 or more variables
  - Each CLB can also serve as a 32x1 memory block (using both 4-input LUTs)
- CLBs are inteconnected using “routing channels”
  - routing channels have different wires of various lengths

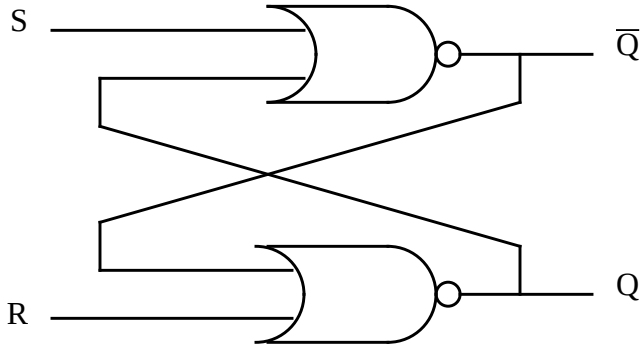
Sequential Logic

Sequential logic circuits have memory

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

F00 E21 Lecture #13

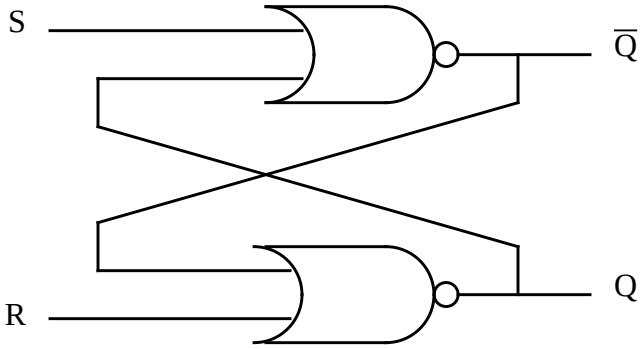
Sequential Logic

Sequential logic circuits have memory

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



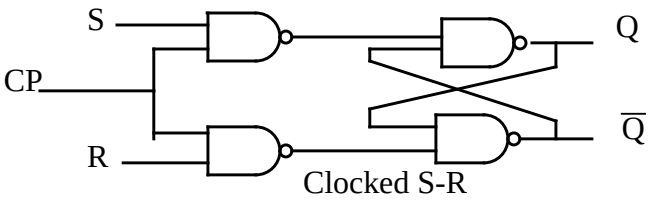
Basic S-R

Useful Flip-flops

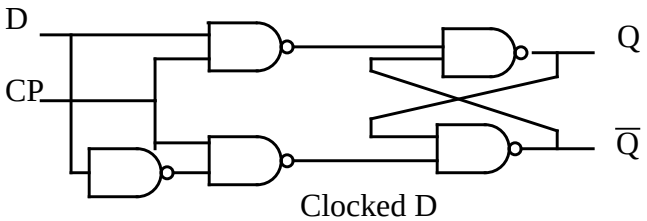
- Clocked S-R
- D Flip-flop
- JK Flip-flop
- T Flip-flop

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

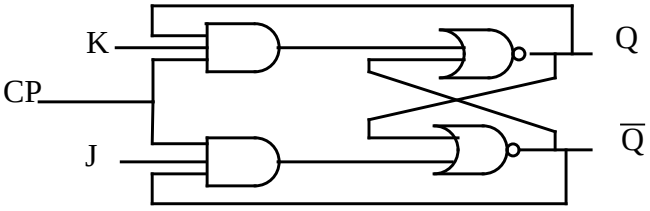
- Master-Slave Flip-flop
  - Two-flip-flops, with the clock pulse inverted on the second
  - Ouput only changes on the falling edge
- D edge-triggered Flip-flop
  - pp. 215, [Mano 1991]
  - Also, pp. 543 [Wakerly, 2000]
  - Output only changes on the rising edge



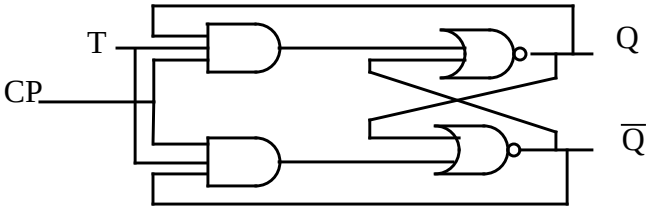
Clocked S-R



Clocked D



Clocked J-K

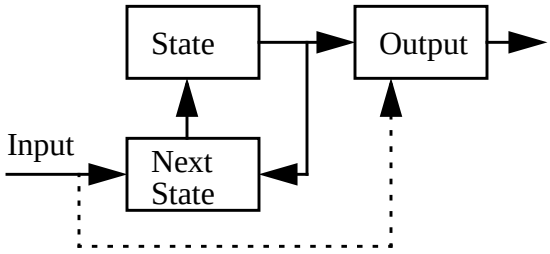


Clocked T

F00 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

Designing a Sequential Circuit

Define the task using words.

Determine, if possible from the task description, whether your outputs should be a function of only the state (Moore state machine), or a function of both the state and the inputs (Mealy state machine).

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
- State values
- Next state values
- 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

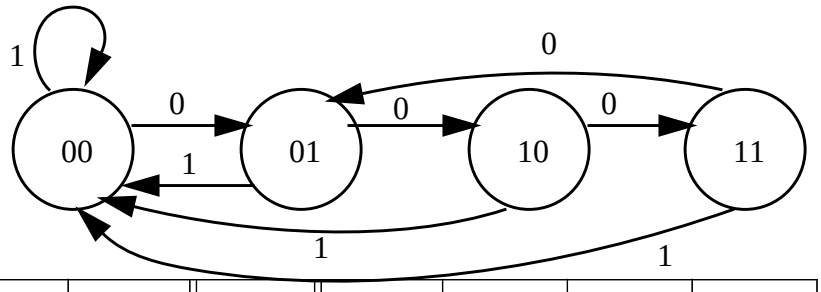
Design the combinational circuits

- For the flip-flop inputs
- For the output

Draw the logic diagram and build the circuit.

Example: 1.Design a state machine that holds a 1 value on the output for 1 clock cycle any time three 0's are received on a bit stream, with no overalap between sequences of 3.

1. Has to be a Moore machine to hold the output
2. Create the state diagram and assign values to the states
3. Create the state table and minimize the number of states (already done)
4. Select the kind of flip-flops to use: S-R.



Input	A	B	A'	B'	Output	S_A	R_A	S_B	R_B
0	0	0	0	1	0	0	d	1	0
1	0	0	0	0	0	0	d	0	d
0	0	1	1	0	0	1	0	0	1
1	0	1	0	0	0	0	d	0	1
0	1	0	1	1	1	d	0	1	0
1	1	0	0	0	0	0	1	0	d
0	1	1	0	1	0	0	1	d	0
1	1	1	0	0	0	0	1	0	1

5. Calculate the logic equations for the output and flip-flop inputs
   - The output is a function of only A and B
     - $O = AB$
   - The logic equations for the inputs to each flip-flop are functions of A, B, and I
     - $S_A = \bar{I}\,\bar{A}\,B$, $R_A = IA + AB$
     - $S_B = \bar{I}\,\bar{B}$, $R_B = IB + \bar{A}\,B$
6. Build the circuit based on the logic equations

F00 E21 Lecture #15

Sequential Logic

Characteristic Excitation Tables

Revisit the characteristic excitation tables

Q	Q'	S	R	Q	Q'	D	Q	Q'	T	Q	Q'	J	K
0	0	0	d	0	0	0	0	0	0	0	0	0	d
0	1	1	0	0	1	1	0	1	1	0	1	1	d
1	0	0	1	1	0	0	1	0	1	1	0	d	1
1	1	d	0	1	1	1	1	1	0	1	1	d	0

Use these tables to fill in the flip-flop excitation values in the state transition table.

Example: Timer circuit

Build a timer circuit that counts from 0 to 15 and outputs a high value on the 0, 10, and 14.

1. Has to be a Moore machine (only input is a clock)
2. State diagram is 16 states from 0 to 15, moving from one to the next.
3. Build state table: states cannot be minimized

A	B	C	D	A'	B'	C'	D'	O	J_A	K_A	J_B	K_B	J_C	K_C	J_D	K_C
0	0	0	0	0	0	0	1	1	0	d	0	d	0	d	1	d
0	0	0	1	0	0	1	0	0	0	d	0	d	1	d	d	1
0	0	1	0	0	0	1	1	0	0	d	0	d	d	0	1	d
0	0	1	1	0	1	0	0	0	0	d	1	d	d	1	d	1
0	1	0	0	0	1	0	1	0	0	d	d	0	0	d	1	d
0	1	0	1	0	1	1	0	0	0	d	d	0	1	d	d	1
0	1	1	0	0	1	1	1	0	0	d	d	0	d	0	1	d
0	1	1	1	1	0	0	0	0	1	d	d	1	d	1	d	1
1	0	0	0	1	0	0	1	0	d	0	0	d	0	d	1	d
1	0	0	1	1	0	1	0	0	d	0	0	d	1	d	d	1
1	0	1	0	1	0	1	1	1	d	0	0	d	d	0	1	d
1	0	1	1	1	1	0	0	0	d	0	1	d	d	1	d	1
1	1	0	0	1	1	0	1	0	d	0	d	0	0	d	1	d
1	1	0	1	1	1	1	0	0	d	0	d	0	1	d	d	1
1	1	1	0	1	1	1	1	1	d	0	d	0	d	0	1	d
1	1	1	1	0	0	0	0	0	d	1	d	1	d	1	d	1

4. Select J-K flip-flops for this circuit.
5. Calculate the logic equations:
   - $J_D = K_D = 1$
   - $J_C = D$, $K_C = D$
   - $J_B = CD$, $K_B = CD$
   - $J_A = \overline{BCD}$, $K_A = BCD$
   - $O = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + A\,C\,\overline{D}$
6. Build the circuit

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

F00 E21 Lecture #16

Useful Sequential Circuits

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 select lines:

- Clear is buffered
- Clock is inverted

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 $C_{i+1} = \overline{D}Q_iC_i + D\overline{Q_i}C_i$
- 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
- Commonly used in computer systems to control different stages of operation

Duty cycle

- The duty cycle of a timer is the percentage of time it is at a high voltage
- 50% duty cycle is equal time on and off
- 25% duty cycle means a pulse of duration 1 within a period of duration 4

Example: Serial to Parallel Converter

Build a circuit that takes a clock and a bit stream as input. In the idle state, the bit stream input is low, and the circuit waits for input. Upon receiving a high bit, the circuit then collects the next eight bits (data) into a register and an optional ninth bit (parity) depending upon a parity flag. The clock signal after receiving the last bit (8th or 9th), the circuit should output a high signal indicating data is ready.

F00 E21 Lecture #17

Useful Sequential Circuits

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
- Commonly used in computer systems to control different stages of operation

Duty cycle

- The duty cycle of a timer is the percentage of time it is at a high voltage
- 50% duty cycle is equal time on and off
- 25% duty cycle means a pulse of duration 1 within a period of duration 4

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 on computer systems

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

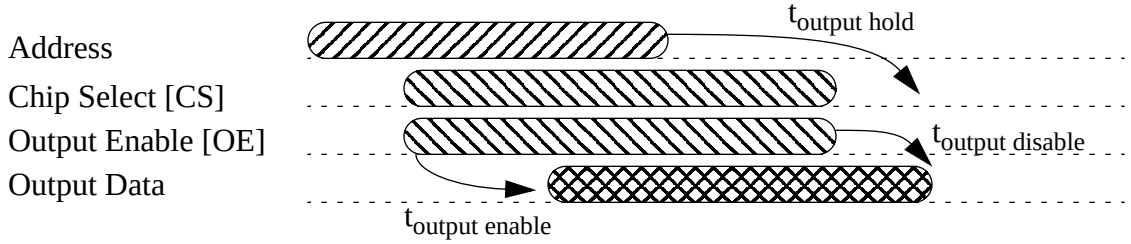
Like in the ROM, a decoder selects the word (N inputs, 2^N output lines)

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

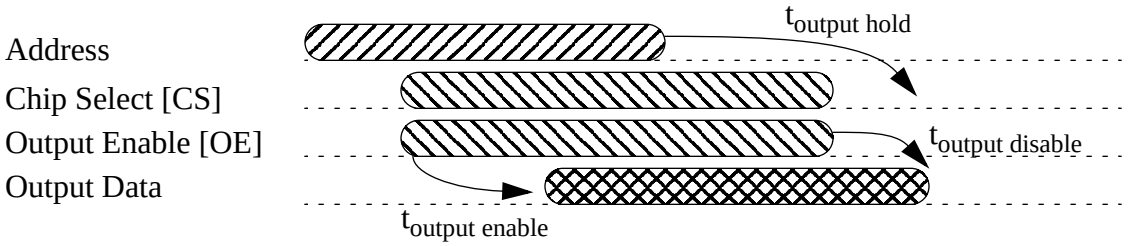


F00 E21 Lecture #18

RAM

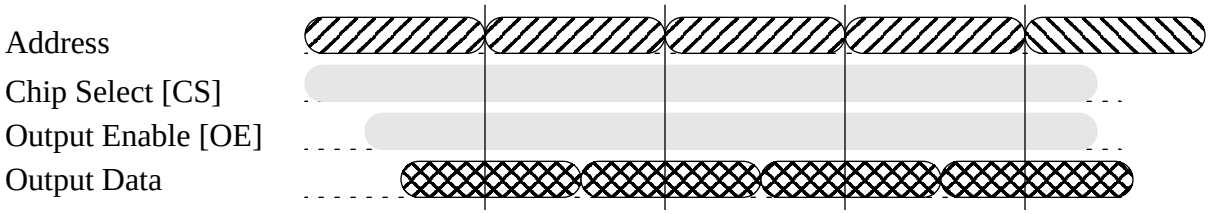
Timing pattern for reading: governed by CS and OE inputs

- 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



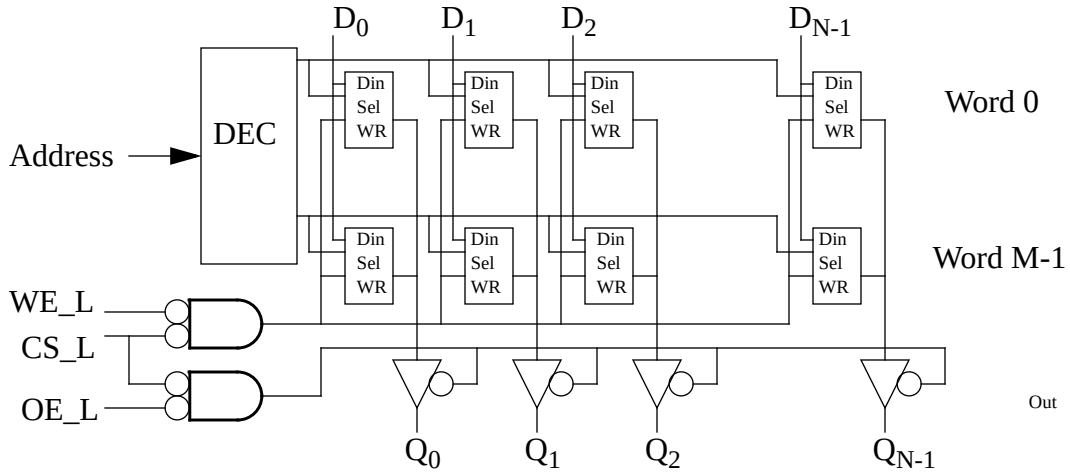
Timing parameters for read operations

- 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 (enables the decoder signal)
- 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.



The diagram above maximizes throughput. Data is read at the sampling points shown. This shows a single memory chip providing four spearate memory locations. At the end of the burst, a new memory chip is selected, and the address is provided to it before the last chunk of data is read from the current memory chip. Note that the output enable timing controls when the chip is putting information on the bus.

SRAM Diagram: Wakerly pp. 857



- WR = WE_L * CS_L
- IOE = 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

Why falling edges in RAM chips?

- RAM chips use latches rather than edge-triggered flip flops
  - Edge triggering requires, in essence, two flip-flops.
- This means the data to be written must be stable for a time period before the falling edge
  - Level-triggered flip-flop
  - Level controlling the flip-flop is a combination of the chip-select and the write signal

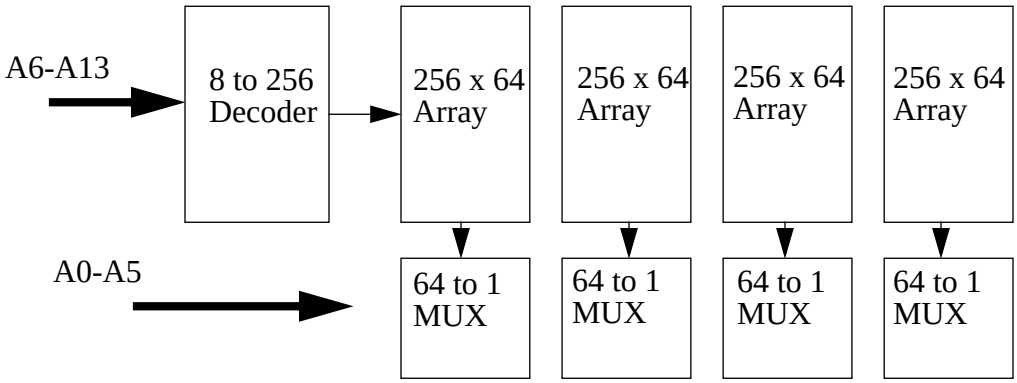
Timing parameters for the write-cycle

- 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). The example below shows a possible layout for a 16K x 4 memory chip.



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

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

F00 E21 Lecture #19

Synchronous Circuits: RAM

SRAM Performance

Asynchronous SRAM: 1999

- Large SRAMs have 4 or 9 Mbits
  - 4M = 256K x 16, 512K x 8, or 1M x 4 bits (7.5-10ns access times)
  - 9M = 512K x 18 (7.5 - 10ns access times)
- Fastest SRAMs have 4Kbits
  - 1K x 4 bits
  - 2.7ns access time
- Example: from IDT (Integrated Device Technology) (www.idt.com)
  - 64K x 16 (1M) asynchronous static RAM comes in 10, 12, 15, and 20ns timings
  - 32K x 32 (1M) pipelined burst synchronous static RAM comes in 5, 6, 7ns timings
  - 512K x 18 (9M) Synchronous flow-through SRAM comes in 7.5, 8.5 and 9.5ns times
- Example: from Cypress (www.cypress.com)
  - 1024k x 32 (4M) asynchronous static RAM has a 12ns 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 200MHz
  - Assume you can access 32 bits/cycle
  - 800 MB/s throughput
  - This is still a factor of 4-8 slower than the processor

DRAM

A transistor and a capacitor

- High densities: factor of 64 x SRAM densities
- Values need to be refreshed every 60ms or so, refresh cycle takes 80ns or so
- Chip is still available for R/W 99% of the time

DRAM timing somewhat different than 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 generally take the row and column addresses sequentially
  - Row address activates a “bank”
  - Column address grabs the appropriate value(s)
- DRAMs take much longer to read than SRAMS
  - 50ns - 100ns is typical for a single read cycle
  - 20ns Row-column delay
  - 20-30ns column-output delay

- SDRAMs help because the operations are pipelined and you can run in “burst” mode
  - Burst modes are approaching 166MHz, or a 6ns clock
- Examples: http://www.mitsubishichips.com
  - Synchronous 64M DRAM: 46-54ns from row activation to getting the first data value

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

ROM: Read Only Memory

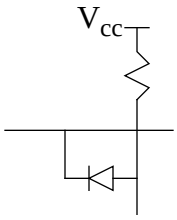
In structure ROMs are similar to RAM except they don’t need the flip-flops to store values

Basic ROM

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

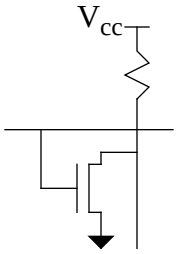
- Manufacturer uses a mask to control what bits are written into the ROM
- Design time: about 4 weeks

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 multiplexers that select a column based on the low Y address bits
- Output is a single bit for each multiplexer

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
  - ROM is often 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, which 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

F00 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 (information 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 indicates 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 just after the rising edge when the state begins
  - Outputs are a function of the state variables
- All register transfer operations occur just after the rising edge when the state ends.
  - Setup time is during the state, and transfer values 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 out of the black box 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
- FPGA synthesis algorithms like to use the “one-hot” method of state machine design
  - Scan the case statement for all conditions coming into a state
  - Create the appropriate AND-OR tree for each state F/F.
- This also helps the synthesis algorithm design the data unit
  - Since there is a signal line for each state, the data transfer expressions are easy to design

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.

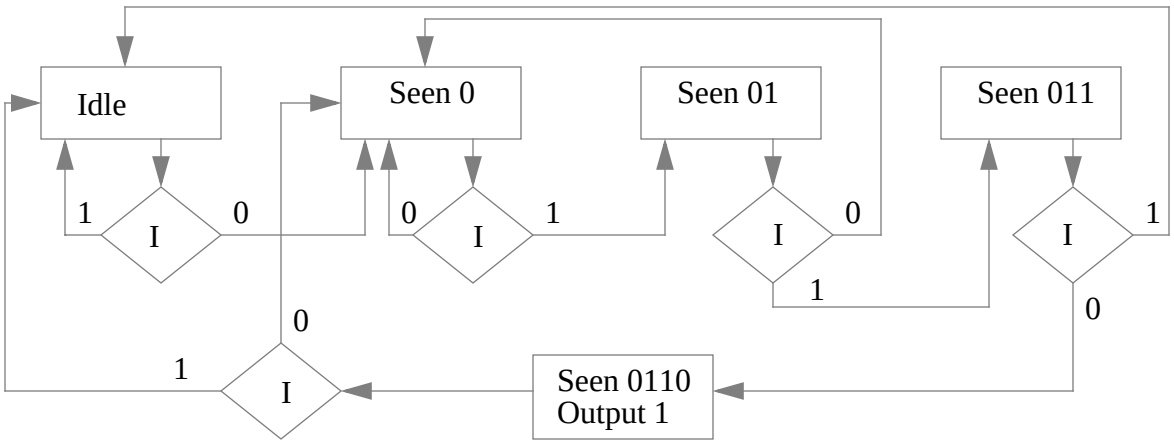
Design with multiplexers

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(s)
- The output of the MUX(s) is connected to the state register inputs
- The MUX(s) 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 MUX input for a particular state bit is the sum of the 1 terms leaving a state

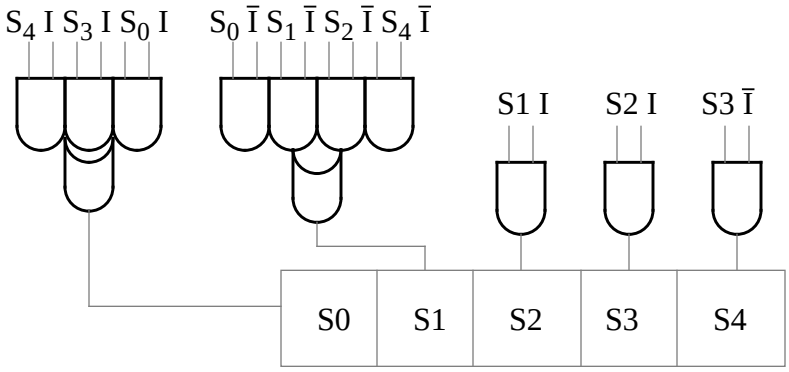
Example

Design a circuit to recognize the non-overlapping string 0110 in a bit sequence.



Flip-flop per state design

Put a decoder on the output of the state variables so there is one control line associated with each state. Then for each state AND together the incoming transition conditions with the appropriate state controls signals.

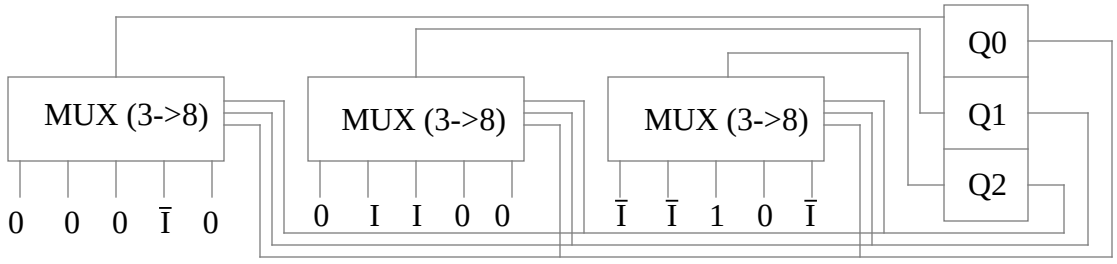


Mux design

Use binary labels for each state and feed the state information to a MUX address.

Table 1: Example MUX Design Table for 0110 bit string recognition problem

Q0	Q1	Q2	Q0’	Q1’	Q2’	Inputs	Mux Q0	Mux Q1	Mux Q2
0	0	0	0	0	0	I			
0	0	0	0	0	1	$\bar{I}$	0	0	$\bar{I}$
0	0	1	0	0	1	$\bar{I}$			
0	0	1	0	1	0	I	0	I	$\bar{I}$
0	1	0	0	0	1	$\bar{I}$			
0	1	0	0	1	1	I	0	I	I+$\bar{I}$ = 1
0	1	1	0	0	0	I			
0	1	1	1	0	0	$\bar{I}$	$\bar{I}$	0	0
1	0	0	0	0	0	I			
1	0	0	0	0	1	$\bar{I}$	0	0	$\bar{I}$



F00 E21 Lecture #21

VHDL State Machine Design Techniques

The standard structure of a state machine is:

- Declare a state variable/register in the architecture
- If reset then put the state machine into the reset state
- Else if it is a rising edge
  - Execute a case statement based on the current value of the state variable
  - Within each case, test the input conditions and update the state and output variables

The standard structure requires you to specify the binary state labels. You can also give the synthesizer the task of assigning state labels by defining an enumerated type.

- type STATE_TYPE is (Initial, seen0, seen01, seen011, seen0110);
- In the case statement, use the state names rather than binary values

A more radical departure in VHDL design is to use records (aggregates) to explicitly store the state table in a VHDL program. The following is a state machine that recognizes two consecutive 0's or 1's in a bit string. This state machine is a Mealy machine implemented using three states.

```
entity two_consecutive is
      port(clk, r, x: in bit; Z: out bit);
end two_consecutive;

architecture fsm of two_consecutive is
      type state is (s0, s1, s2);
      signal fsm_state: state := s0;

      type transition is record
            output: bit;
            next_state: state;
      end record;
      type transition_matrix is array(state, bit) of transition;

      constant state_trans: transition_matrix :=
            (s0 => (‘0’ => (‘0’, s1), ‘1’ => (‘0’, s2)),
            (s1 => (‘0’ => (‘1’, s1), ‘1’ => (‘0’, s2)),
            (s2 => (‘0’ => (‘0’, s1), ‘1’ => (‘1’, s2)));
begin
      process(r, x, clk, fsm_state)
      begin
            if r = ‘0’ then -- reset
                  fsm_state <= s0;
            elsif clk’event and clk=’1’ then -- clock event
                  fsm_state <= state_trans(fsm_state, x).next_state;
            end if;

            if fsm_state’event or x’event then -- output function
                  Z <= state_trans(fsm_state, x).output;
            end if;
      end process;
end fsm;
```

This program is our 0110 recognizer implemented as a 5-state Moore machine

```
entity recognizer is
      port(clk, r, x: in bit; Z: out bit);
end recognizer;

architecture fsm of recognizer is
     type state is (s0, s1, s2, s3, s4);
     signal fsm_state: state := s0;

     type transition is record
          output: bit;
          next_state: state;
     end record;

     type transition_matrix is array(state, bit) of transition;

     constant state_trans: transition_matrix :=
          (s0 => (‘0’ => (‘0’, s1), ‘1’ => (‘0’, s0)),
          (s1 => (‘0’ => (‘0’, s1), ‘1’ => (‘0’, s2)),
          (s2 => (‘0’ => (‘0’, s1), ‘1’ => (‘0’, s3)),
          (s3 => (‘0’ => (‘0’, s4), ‘1’ => (‘0’, s0)),
          (s4 => (‘0’ => (‘1’, s1), ‘1’ => (‘1’, s0)));
begin
     process(r, clk)
     begin
          if r = ‘0’ then -- reset
               fsm_state <= s0;
          elsif clk’event and clk=’1’ then -- clock event
               fsm_state <= state_trans(fsm_state, x).next_state;
               Z <= state_trans(state_trans(fsm_state, x).next_state, x).output;
          end if;
     end process;
end fsm;
```

Communication between devices

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

On the computer end of things, each I/O port has its own protocol for communication

F00 E21 Lecture #22

Communication between devices

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*

On the computer end of things, each I/O port 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 or vitual files (Linux)
- 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

- State machines are a simple method of controlling communication.

Parallel Communication

Centronics mode

- One-way communication controlled by the host
- Uses the 8 data lines and a strobe
- Peripheral has a busy line and an ack line, but they may not be acknowledged by host
- Process
    - Host places data on the data lines
    - Host checks if busy line is low (peripheral ready?)
    - Host asserts nStrobe (active low)
    - Host de-asserts nStrobe
    - Peripheral may assert busy/ack lines while reading the data

Nibble mode

- One-way communication from the peripheral to host
- Sends 4 bits at a time through the status register input lines
- Process
    - Host signals readiness by setting HostBusy low
    - Peripheral puts the first nibble on the four input lines
    - Peripheral signals valid data by asserting PtrClk low
    - Host sets HostBusy high until it is ready for the next nibble
    - When host sets HostBusy low again, peripheral repeats process for second nibble

EPP: Enhanced Parallel Port mode

- Two-way communication using bi-directional data lines
- Achieves 500kB to 2MB data rates
- From the hosts point of view, it just involves a write to the EPP data port
- Process for a data write cycle
    - Program writes to the EPP data port (port 4)
    - The nWrite line is asserted and the data is output to the parallel port
    - The data strobe is asserted (low) since nWAIT is asserted low
    - The port waits for the acknowledge from the peripheral (nWAIT de-asserted)
    - The data strobe is de-asserted and the EPP cycle ends
    - nWAIT is de-asserted by the peripheral to indicate that the next cycle may begin
- Process for an address/command write is the same

ECP: Extended Capability Port

- Standard includes run-length encoding compression
- Also a bi-directional communication protocol

Negotiation

- Devices that are compliant with the IEEE 1284 standard (which includes EPP/ECP) must be able to negotiate what mode to use and what ID a device might have

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
  - Need 3 wires to have a control line
- 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
- Asynchronous communication
  - Transmitter and receiver may have different rate clocks, but an agreed upon data rate
  - Information is only broadcast when there is information to send
  - Format of the message indicates when there is information arriving

F00 E21 Lecture #23

Serial Communication

Methods of serial communication

- *Simplex mode: information flows one direction only (2 wire communication)*
- *Half duplex mode: information flows both directions, but only in one direction at a time*
- *Full-duplex mode: information flows both directions simultaneously*
- *Synchronous communication*
- *Asynchronous communication*

Asynchronous communication protocol

- Start bit (0) (opposite of idle line)
- Character bits (7 or 8 of them)
- Parity bit (1)
- Stop bit (1-2 of them) (equal to idle line)

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

Example asychronous receiver

Receiver is based on a fast clock

- Process initiates on a start bit assertion, sample time is set to be in the middle of a bit
- Character bits get read in the middle
- Parity bit is read
- Receiver indicates to host that data is ready and resets back to the idle state

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, ticks
- 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
- Resolution (spectral): how precisely can you measure frequency?
- Dynamic Range: range within which the sensor operates
- Nyquist Criterion: you have to sample at least twice the frequency you want to see
    - Remember: aliasing creates signals that don't actually exist

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
        - Direct RF/IF processing (don't have to do it in analog circuitry
        - High-speed data acquisition
        - Digital Oscilloscopes
        - Radar/ECM systems

F00 E21 Lecture #24

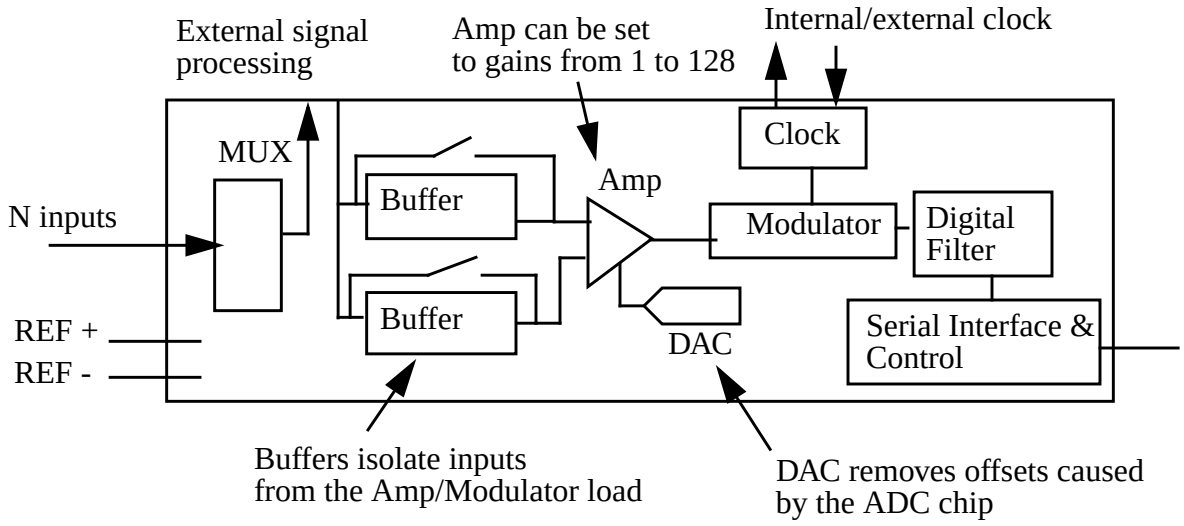
Sensing & A/D interfaces

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 (slow sampling rate)
- 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

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 (hi-pass filter)

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

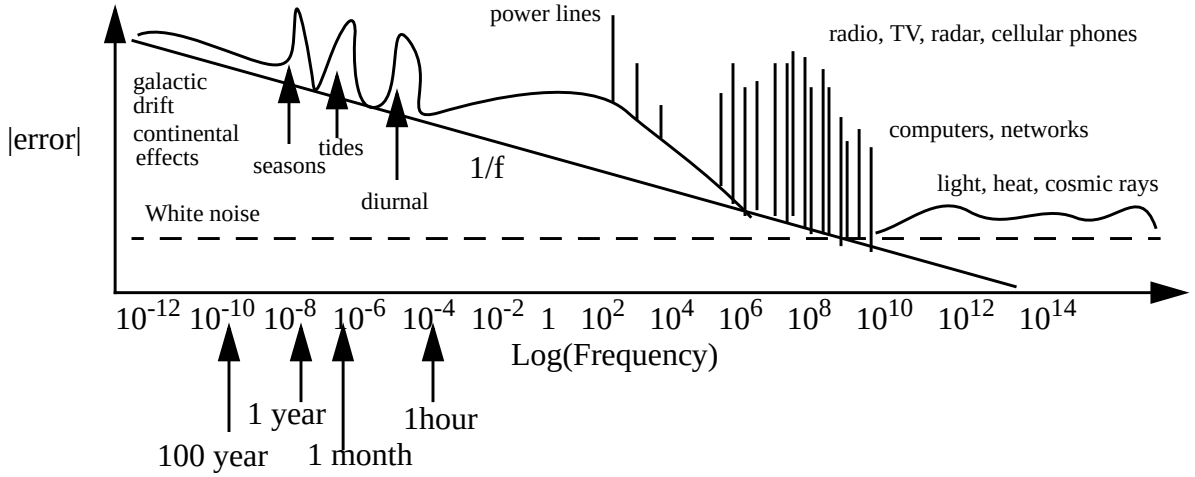
Shot Noise: quantization of charge

- 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. It is always possiblet o do much worse.



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

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

F00 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
- We can apply evolutionary techniques to algorithms?
- Can we apply evolutionary techniques 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 μand 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?)

Example

Maximize $(x-20)^2$ over the range [0, 31].

Representation: 5 bits, interpreted as a binary number

Population size: 15 (class size)

Mutation rate: 1 in 12

Number of generations: 10

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

F00 E21 Lecture #26

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

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
  - The 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 allowed 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 (next to a light bulb)
- 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

F00 E21 Lecture #27

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 in many cases)
- 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

In fair use cases, the four criteria are supposed to be weighted equally. In practice, factors #1 and #4 have received the most attention since it is usually commercial ventures that sue, and they can throw up numbers in support of their case.

F00 E21 Lecture #28

Administrivia

1) When to have the exam

2) Project presentations

Intellectual Property Rights: Patents

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
  - Gene sequences and chemical compounds that have specific utility
  - Software with a specific, limited utility
- 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. These are the most common.

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 (prosecutes the patent)

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

You have to pay a small initial fee to apply for and receive the patent. Then you have to pay maintainance fees at 3.5, 7.5, and 11.5 years. The fees go up each time, but they are pretty small ($850, $1950, and $2990 for large organizations, half that for small).

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. In the 90s, attention focused 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 a practical application of a mathematical algorithm, formula, or calculation, because it produces a useful, concrete and tangible result - a final share price”.
    - This removed 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).

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.

F00 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 (12Mbits/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/Address 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
- Four types of 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
  - 16-bit CRC code
- Handshake packets
  - ACK: ok
  - NAK: error detected
  - STALL: device or computer is busy
  - no CRC field

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

Device Classes

- Since many devices are similar in function, there are a set of standard classes

- Common class, Communications, Hub, Printer, Image, Monitor, Mass Storage, Audio, HID [Human Interface Device], PID [Physical Interface Device], and power.
- Some devices can belong to more than one class
- Each class has a standard software driver from which you can develop specialized drivers, or mini-drivers.

Building a USB device

There are several USB protocol chip-sets on the market.

- Many are simple microcontrollers with example firmware that you can modify
  - This means you may not need any other computing on the peripheral
- Some have a serial interface with another microcontroller
- Some have a parallel interface with another microcontroller

You can also purchase prototype USB devcies

- Some of them automatically download the “firmware” from the USB interface on startup
  - This makes for very simple prototype development

If you want to build your own USB chip

- Much more complicated than a standard UART (serial interface)
- Using a microcontroller makes sense in this situation
  - Write the USB code in C rather than trying to design a digital circuit to handle it

F00 E21 Lecture #30

I/O Interfaces: PCI Bus Protocol

If you need more bandwidth than provided by the USB specification then your options are:

- Firewire: external fast serial protocol (25 to 400 Mbits/s)
- PCI: internal fast massively parallel (32/64 data bits) 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

- 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
- JTAG/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. The bus master can change each interaction

- 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 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

F00 E21 Lecture #31

I/O Interfaces: PCI Bus Protocol: A State Machine Representation

Table 1: State table for a simple I/O device (Read cycle)

Current	Next	Condition	Action
Idle	Idle	$\overline{\text{Frame}}$	DEVSEL, TRDY deasserted
Idle	Check Address	Frame	1) latch command lines 2) latch address lines
Check Address	Idle	$\overline{\text{My address}}$	Disregard and return to idle state
Check Address	Get & Place Data	My address & Read	1) latch byte enable lines 2) Assert DEVSEL 3) Assert TRDY 4) Place data as specified by C/BE 5) Increment internal address
Get & Place Data	Get & Place Data	$\overline{\text{IRDY}}$	1) Maintain data 2) Maintain 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) latch byte enable lines 2) Assert TRDY 3) Place data as specified by C/BE 4) Increment internal address

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
  - 1-40MHz
- On-chip peripherals
  - A/D converters

    - USART: Universal sychronous/asynchronous receiver/transmitter
    - USB interface
- 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 used to be the largest manufacturer and user of microcontrollers? GM

- Motorola is the largest today (and they have the best development tools)

What does a microcontroller cost?

- Between $2 and $20 (Motorola 68HC908JB8 with USB 1.1 built-in is $2.50)

F00 E21 Lecture #32

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

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

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

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

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

F00 E21 Lecture #33

Microcontroller Example: PIC Microcontroller Architecture

Let's look at the major design characteristics of the PIC microcontroller (167XXX series)

Bus architecture

PIC is a one-bus architecture

- Everything hangs off of one 8-bit data bus except the program memory
- All binary operations use the W register as one operand

Address modes

- Immediate: operand comes from the instruction (literal instructions)
- Direct: address comes from the instruction (file register address)
- Indirect: address comes from the FSR register (File Select Register)
  - Indirect addressing treats the FSR register as an index into an array of registers

Harvard v. Von Neumann Architecture (Memory design)

Where do 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 occur 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

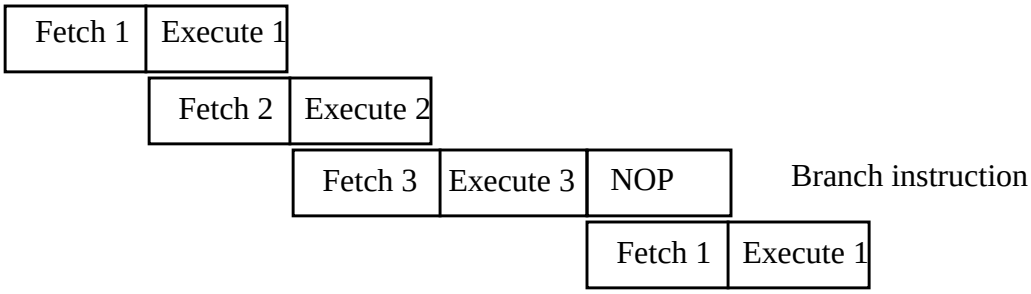
Pipelining

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



The Fetch Cycle

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

Example: PIC

- Clock is divided into four states: Q1-Q4
- Q1: increment PC
- Q2, Q3: hold address for program memory
- Q4: latch instruction register
- In the case of a branch the new address is loaded into the PC on Q4 and the control unit suspends operation until the following Q4 so that the correct next instruction gets loaded

Execute Cycle

On some microcontrollers the length of the execute cycle depends upon the instruction

- Motorola 6811: different instructions take different amounts of time
- This means the control unit is usually microcoded, which means each instruction is executed by a simple program in the control unit microcode memory

On other microcotnrollers the length of the execute cycle is constant

- This often means you don't have complex instructions (like multiply or divide)
- All instructions follow the same format

Example of a constant length execution cycle (PIC):

- 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
- Since the PIC is pipelined, the instruction fetch cycle takes place simultaneously

F00 E21 Lecture #34

Example Instructions

The PIC instruction set is rather small: 35 instructions

Typical examples are

- MOVLW: move a literal to the W register
  - Q1: IR[literal] -> ALU Mux input
  - Q2: ALU latches the value
  - Q3: Value appears on ALU output
  - Q4: Value latched into the W register
- MOVF: move the contents of a file register either to the W register or back to itself
  - Q1: Apply address from IR or FSR to file register
  - Q2: ALU latches the value on the data bus
  - Q3: Value appears on ALU output
  - Q4: Value latched either into W or the file register (d = 0, or d = 1, respectively)
  - The case d = 1 is useful for testing the value of a bit in the register
- ADDWF: add the contents of f and W and put the result in either W or f
  - Q1: Apply address from IR or FSR to file register
  - Q2: ALU latches the value on the data bus
  - Q3: Value appears on ALU output
  - Q4: Value latched either into W or the file register (d = 0, or d = 1, respectively)

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 (indicates a logical operation & flags with the ALU)
- [10]: Call or goto (indicates stack push action)
- [11]: Operations with literals (controls ALU input MUX)

- 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

F00 E21 Lecture #35

Example: Subroutines on a PIC

CALL, RETURN, and RETLW allow you to implement subroutines

Why do you want subroutines?

- It makes the code more compact when you can separate out repeatedly used chunks
- You have to be able to jump to the starting address of the routine
- You have to remember where you jumped from
- You have to be able to jump back to where you were

The PIC handles subroutines return addresses through a stack mechanism

CALL

- The address of the subroutine is in the instruction (11 bits, plus 2 bits in PCLATH)
- The value of the incremented PC (Q1) is pushed onto the return address stack (Q2/Q3)
- The new address is loaded into the PC (Q4)
- There is a NOP while the address is applied to the program memory (Q1-Q3)
- The IR loads the first instruction from the subroutines (Q4)

RETURN

- The PC gets the address on the top of the stack (Q2/Q3)
- There is a NOP while the address is applied to the program memory (Q1-Q3)
- The IR loads the instruction just after the subroutine was called (Q4)

RETLW

- The PC gets the address on the top of the stack (Q2/Q3)
- A literal value k in the instruction goes through the ALU into the W register (Q2-Q4)
- There is a NOP while the address is applied to the program memory (Q1-Q3)
- The IR loads the instruction just after the subroutine was called (Q4)
- You can use the RETLW instruction to create a lookup table in program memory
    - CALL the lookup table subroutine
    - Calculate the offset into the lookup table and add that offset to the PC
    - Each entry in the lookup table is a RETLW instruction that returns from the subroutine while putting the appropriate value in the W register.

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
- I/O and A/D ports have their own state machines to control their actions

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

PIC 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

Also note, W_TEMP must exit in both “banks” of memory since the current value of the RP0 bit could be either 0 or 1 in STATUS.