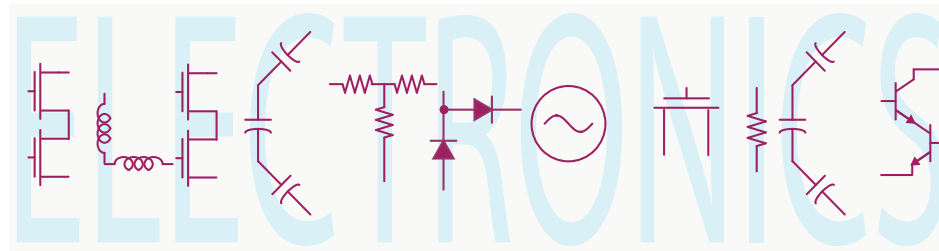


# Announcements

- OH F ~4-5pm  
Check Piazza
- HW DEADLINE: 8PM FRI
- REVIEW: NEXT TUE
- EVALS TODAY

*EE 42/100*

*Lecture 25: Binary Signals / Microcontrollers*



Rev C 4/26/2012 (9:07 AM)

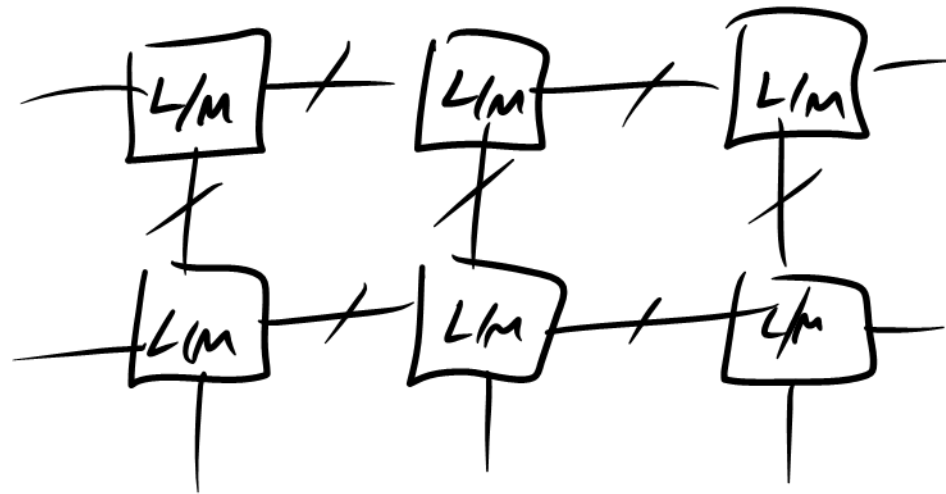
Prof. Ali M. Niknejad

University of California, Berkeley

Copyright © 2010 by Ali M. Niknejad

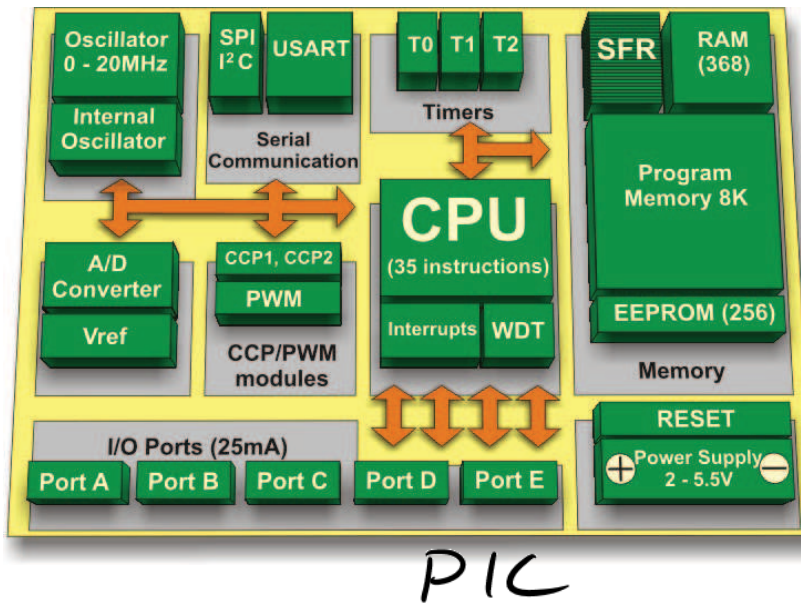
# FPGA

FIELD PROGRAMMABLE GATE ARRAY



L/M = LOGIC + MEMORY

# Microcontrollers – Introduction



UART	Universal Asy. Rec./Trans.
SFR	Special Function Registers
WDT	Watchdog Timer
PWM	Pulse Width Modulation
SPI	Series Peripheral Interface
I <sup>2</sup> C	Inter-Integrated Circuit

- Microcontrollers are stripped down versions of microprocessors with additional functionality specific to control applications.
- To save power, they run at much lower speeds than microprocessors (MHz clock rates), work with less memory, and run much simpler software. Often there is no operating system and no more than a few kBytes of RAM.
- Speed is usually a secondary concern in microcontroller applications but latency is very important. In other words, we want the system to be very responsive to changes. Running a complex operating system is not a good idea!
- The most important additional functionality is the ability to interface with external signals, both digital and analog, using several ports.
- Most importantly, microcontrollers are very low cost, less than a dollar in certain applications.

# *Microprocessors*

- Contrast this to today's microprocessors, which run at multi-GHz clock speeds, have multiple cores, and have access to Mbytes of on-chip cache and Gbytes of off chip memory.
- Today's microprocessors are designed to run complex applications, often in multi-tasking modes, and work with relatively complex multimedia signals (audio, video, 2D/3D graphics).
- The downside is that they consume several watts of power and have complex pinouts and cost tens to hundreds of dollars. In many control applications, we require a part that must run at much lower power levels and the required software is very simple.

# *Applications*

- Almost any electronic device today has a microcontroller in it. From a toaster oven to a dishwasher or battery operated toy, the microcontroller is used to “control” the device.
- In more complex systems, such as automobiles or aircrafts, tens to hundreds of microcontrollers are used in almost every aspect of the design.
- Good Example: Compression ratio of modern internal combustion engines.
- While custom circuitry can be used in certain applications, such as a toaster oven, it is usually faster to design a system based on a microcontroller. Most of the design time is the software, which can be changed and fine tuned, sometimes even after a product is shipped! (the much beloved software update)

## *Oven Block Diagram*

- In this simple system, we have heating elements, switches), a timer, and user input. This is simple enough that an EE42/100 student could easily design a custom circuit to make it all work. But as the complexity of the system increases, the pure hardware solution gets increasingly more complex. Software is easier to write (12-year olds make good programmers!) and the system is much more generic.
- More complex functionality is enabled by a microcontroller. For instance, feedback control can be used to work with less precise components by monitoring the oven temperature and adjusting the voltage accordingly.
- This also allows a richer interface (toaster oven can have settings for bread, potatoes, etc) to be introduced, giving the user a “push button” experience: Just toast my bread the way I like it! (customized settings)
- In the future, a radio in your toaster will be used in a “smart home”. Why? Repair, upgrades, recall, remote shutdown (using a phone app), integration with the heating/cooling system, ...

# Generic Digital Process Control

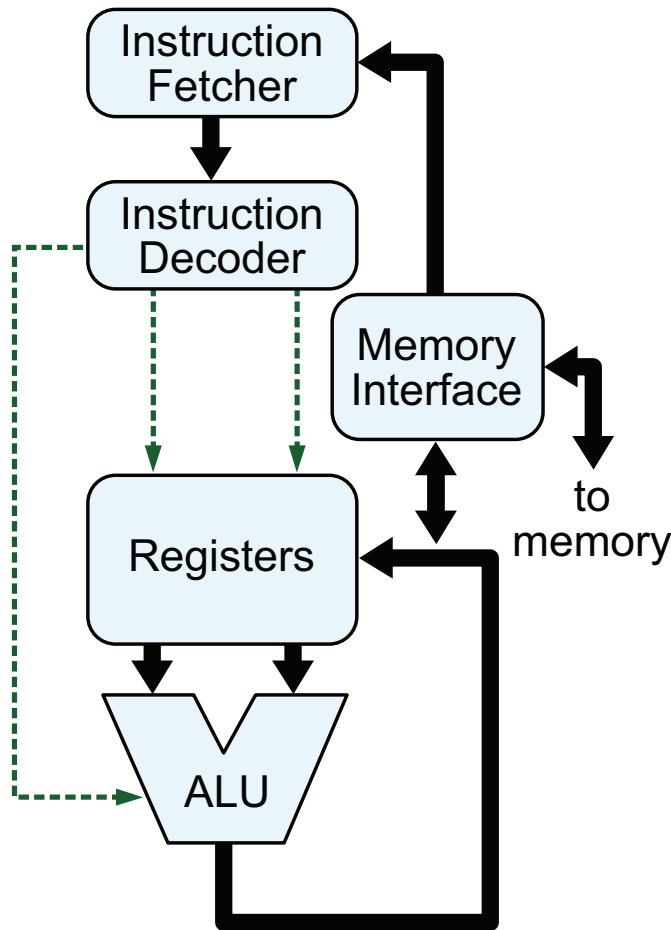
- Sensors: Sense a physical quantity to be measured. Can be digital or analog. A digital light sensor only detects if it's dark or light. An analog light sensor detects the amount of light impinging on the sensor. Other examples: temperature, pressure, humidity, ...
- Actuators: These devices take an electrical input (analog or digital) and actuate (cause into action). A good example is a solenoid switch, which responds to an electrical signal with a mechanical force that can be used to shut on/off a valve. Examples: Motors, steppers ...
- Transducers: Convert a non-electrical signal to electrical form. Piezoelectric device is a good example (electricity  $\leftrightarrow$  mechanical force), speakers (audio), resistor (electrical  $\leftrightarrow$  heat).
- Display: As simple as an LED or as complicated as a HD LCD screen. Printers.
- Operator Inputs or Settings: Some way to input a setting into a system. The input can come remotely from a radio signal, a dial, a keyboard, a touch screen, etc.



## *Microcontroller Block Diagram*

- The microcontroller has several key blocks including a central processing unit (CPU), several registers (local high speed memory), on-board RAM/ROM memory, an addressing and data bus (sometimes shared), and several ports (digital, analog) for input/output. There is also a timing system used to synchronize events.

# Central Processing Unit: CPU



- The CPU is the “brain” of the microcontroller, and it consists of an Arithmetic Logic Unit (ALU) and control circuitry.
- The CPU is fed with instructions and data from memory. Each instruction tells the CPU to perform some simple function (add two numbers, for instance).
- CPU instructions are in the form of op-code and then operand(s). The op-code is a numeric code that tells the CPU to perform a specific function, such as “addition”. The operands (the two numbers to add) may be supplied directly or they must be fetched from memory. The actual operation occurs in the ALU.

# *System Clock*

- The CPU runs at a certain clock speed. Each clock cycle the CPU performs a given operation. The clock speed is directly related to the power dissipation of the CPU.
- Simple instructions may complete in one cycle. More complex instructions may take several clock cycles. The key point is that the operations of the CPU are designed to occur in discrete units of time.
- For low power applications, an internal clock can be generated. This clock tends to drift with time due to temperature variations and noise in the circuit.
- For precision applications where timing accuracy is very important, a crystal reference is used to establish the clock frequency.

# ALU

- The ALU can perform simple operations (integer addition/subtraction, maybe integer multiplication). Often only integer operations are supported (no floating point!). For example, if division is not supported, the calculation must be done through software.
- In this class we'll design a simple ALU that can perform addition and subtraction.

# Registers

- The CPU has several important registers that are used as “scratch space” and to store the current location in memory for retrieving instructions.
- Fundamental to the operation of the CPU, the following registers are used
  - The instruction pointer (IP) or program counter(PC), points to the next instruction to be executed.
  - The stack pointer (SP), which points to the location of the “stack”
  - Accumulators or general purpose registers that are used for arithmetic operations.
  - Index registers used for addressing modes.
  - A condition register records any significant events that may have occurred in the previous calculation (overflow, carry, zero, etc).

## *Instruction Pointer (IP)*

- During each execution cycle, the data stored at location IP is read into the CPU and executed. Each instruction contains an op-code (instruction) and operands (ADD A B) which specify where to retrieve the data for the calculation (memory or other general purpose registers).
- The instruction pointer is incremented each cycle to point to the next instruction (the width of instructions varies based on the number and width of operands).
- The flow of instruction can jump to a new place (JSR – such as a jump to sub-routine) or jump back (to implement a loop) or forward (goto new location). Thus “jump” instructions just perform arithmetic on the value stored in the IP.

## *Stack Pointer (SP)*

- The “stack” is a memory location used for temporary storage. The structure is a First-In Last-Out (FILO) structure, much like a stack of plates.
- Data is pushed onto and off of the stack with a “PUSH” and “PULL” (or “POP”) instruction. The stack pointer is set to a high value of memory and the stack grows downward.
- The FILO structure is very convenient for implementing sub-routines, especially nested sub-routines or recursion.
- Suppose a program is executing and it is interrupted, which means it needs to respond to an external (or internal) event – new data is available from a sensor. The CPU needs to do a “context switch”, which means it has to save what it’s currently doing, save the current location, jump to a new memory location (the interrupt service routing (ISR)), and then come back and continue where it left off.
- The way this is done is that the current location (next instruction) is pushed onto the stack, and then all the registers are “pushed” onto the stack. The IP is updated to the ISR address, the ISR is executed, and then to return, the registers are also pulled off the stack and calculations resumes by pulling off the return address from the stack.

# *Sub-Routines*

- Subroutines make heavy use of the stack:
  - Caller pushes arguments onto the stack (caller stack frame)
  - Caller pushes a location for the return value on the stack (caller stack frame)
  - Callee accesses arguments in caller's stack frame
  - Callee pushes space for local variables (callee stack frame)
  - Callee may itself call other subroutines
  - When the callee computes the return value, it places it in the caller part of the stack. Remember that the caller reserved some section of the stack for the return value.
  - Callee restores stack pointer back to the way it found it just as it was being called. Thus, stack pointer now points to the return value.
  - Caller gets the return value, and eventually pops that off and the arguments off.



# Memory

- Memory is a critical component of a microcontroller. Both instructions (programs) and data are stored in memory.
- There are several kinds of memory in a microcontroller. Let's focus on the on-board (internal) memory.
- The main differences in memory types are the speed (how many clock cycles does it take to access the memory) and the volatility (does the information persist after the power is shut down?).
- The fastest memory is in the form of “static RAM”, which is a set of register files which reside close to the CPU and the cache. These consume the most power and take up the most area, and so only a few (3-4 or dozens of registers, or kbytes or Mbytes of cache) are available to the CPU.
- Many CPU instructions operate on these registers. Example: ABA → Add the contents of registers A and B and store the result in A.

## *The Address/Data Bus*

- When the CPU needs to access external memory, it does so through the “bus”. While communication can occur over a “serial” or “parallel” link. A parallel link is faster but less immune to interference and cross-talk (short range).
- The bus is a series of parallel wires that connect the CPU to external memory. The width of the bus depends strongly on the application. In the lowest cost applications, a small bus of 8-bits is used.
- More complex buses use 16-bits, 32-bits, or even 64-bits. The width of the memory has two important implications: the speed at which data is read off per clock cycle and the number of unique points in memory that can be addressed. A 16-bit address bus can only access a maximum of  $2^{16} = 65536$  points in memory, or 64K. To access more memory, early computers used various addressing modes (“paging”). For most microcontrollers, this is enough memory!
- The bus is bidirectional since information flow occurs in both directions. For instance, if the CPU wants to read from memory, it can write a certain address to the bus and then “assert the address”. The memory responds by writing the contents of the address back onto the bus and then it “asserts the data”.

## *Random Access Memory – RAM*

- RAM stands for random-access memory, meaning that any memory location can be accessed randomly. This terminology is outdated because most memory is like this, but the name stuck. A more accurate name would be RW memory, or read-write memory, since each memory cell can be easily read or written.
- RAM is volatile, meaning that the contents of the memory cells will be lost if power is not applied to the chip.
- RAM also comes in two flavors: static and dynamic. We'll learn how to design static RAM later in the course but the basic ingredient is a bistable circuit. You have already met a circuit that has two output states only – the Schmitt Trigger. In a bistable circuit, the input only needs to be applied momentarily to “flip” the state of the cell, and then the cell will store the state as long as power is supplied.
- In a dynamic RAM cell, tiny capacitors are used to store states. The presence or absence of charge stores a “1” or “0”. As we shall show, this state needs to be *refreshed* periodically otherwise the charge leaks away.
- Access time for static RAM is much faster, but this kind of RAM is much more expensive since it occupies larger area.

## Dynamic RAM

- The capacitors are so small that the charge quickly leaks away due to leakage currents (tunneling, leakage in diodes, very small conductance of insulators, etc).
- Suppose the capacitor is 10 fF and there is 100pA of leakage current (parasitic diodes associated with the switches that access the transistor). That means the switch will discharge from 1V to 0V in a time

$$\Delta t = \frac{Q}{I_0} = \frac{C\Delta V}{I_0} = \frac{10 \text{ fF} \cdot 1V}{100 \text{ pA}} = \frac{10 \times 10^{-15}}{100 \times 10^{-12}} = 0.1 \text{ ms}$$

- This means that the capacitor has to be refreshed at a rate of 10 kHz or more to keep the data persistent. If power is turned off, the information leaks away very quickly.

# ROM

- ROM – Read Only Memory. This kind of memory can only be read (cannot write to it). It's used to store small programs (boot up sequence for a CPU).
- Unlike RAM, ROM is non-volatile memory, meaning that it keeps its state regardless of the whether power is supplied to a cell.
- There are many ROM technologies. The simplest are made of resistor fuses. If a large current is run through a thin wire, it melts and we get an “open circuit” between two points. This corresponds to say a “1”; if the fuse is not blown, that represents a “0” state.

## *Flash*

- Flash: Store data on “floating gate” transistors – something we’ll learn about later.
- Invented in 1980, Flash memory has become very popular in the past 10 years – think of the iPod and the introduction of solid-state disk drives.
- Flash memory is like ROM in that it’s non-volatile, but it’s electrically programmable. This is very convenient since the “hardware code” in a device can be updated on the fly.
- The read times are slower than RAM, but still fast enough for many applications. The write times are much slower.

## *Ports*

- Input/Output ports on a microprocessor allow external signals to directly interface with the microprocessor.
- Typically a few or tens of signals are available. The microprocessor can write digital signals to each port (say an LED display) or it can read data from external inputs. The same pins are shared to save real estate.
- Some microcontrollers have built-in conversion capability from digital to analog form.

## *Digital I/O*

- Some sensors output digital signals naturally. These need to be conditioned to have the right voltage levels before they are applied to the microprocessor. Even if the signal is analog, we can restrict it to take digital levels (light detector example) using some simple analog signal processing (Schmitt trigger).
- The CPU can only do one thing at a time. In every given clock cycle, it is executing an instruction. If we want it to read/write a digital port, we usually read/write a particular memory address reserved for the port.



## *Interrupts versus Polling*

- We may read the digital ports either periodically (synchronous) by polling or in an event driven fashion (asynchronous) using interrupts.
- Say we have a microprocessor running at 1 MHz, or a clock period of 1  $\mu\text{s}$ . We can periodically examine the port signals (say 1 out of every 100 cycles), which means we can read signals changing as fast as 10 kHz, or 0.1 ms. This means that we need a mechanism to do this periodically.
- Most microcontrollers have timers that can be used to raise interrupts periodically. When an interrupt occurs, the microprocessor saves its state, stops the current instruction, and instead jumps to a particular memory address and executes the code say corresponding to reading the port.
- On the other hand, we can use the digital input to interrupt the microprocessor when a new value is ready. This is done by raising the Interrupt Request (IRQ) pin of the microprocessor.

## *Analog I/O*

- Some microcontrollers can read and write analog signal directly.
- Inside the microcontroller, all signals are digital. To read/write analog signals, therefore, requires signal conversion.
- This circuit building block is an Analog-to-Digital Converter (ADC) and Digital-to-Analog Convertors (DAC).

## *Analog-to-Digital Converter*

- The important specifications for an ADC are the sampling rate (clock rate) of the ADC and its resolution (number of bits).
- Analog signals need to be sampled at a rate of twice the signal bandwidth (audio bandwidth is roughly 5 kHz).
- The resolution determines the smallest discernible signal since the full-scale input voltage (say 5V) is divided by  $2^N$  where  $N$  is the number of bits. Any signal smaller than this level is lost in the “quantization noise” (round-off error).

## *Digital-to-Analog Converter*

- Likewise, some microcontrollers can write analog outputs directly using a Digital-to-Analog Converter (DAC).
- In class and homework we learned how to build some simple DACs using current summing op-amp circuits.
- A “poor person’s DAC” can always be realized by using a variable duty cycled signal to represent an analog output. The signal must be filtered and oversampled (clock frequency much higher than the signal bandwidth).

## Instruction Set

- The typical microcontroller includes several simple instructions. We'll explore some sample instructions to get a flavor for the machine language of the CPU (examples from the 68HC11 ubiquitous controller):

```
ABA (opr)    ; Add accum: A + B --> A
ADDA (opr)   ; Add memory to A: A + M --> A
ADDB (opr)   ; B + M --> B
ADDD (opr)
BCS (rel)    ; Branch if carry set
BEQ (rel)    ; Branch if zero
BLO (rel)    ; Branch if Lower
BNE (rel)    ; Branch if not equal
BRA (rel)    ; Always branch
CLRA        ; Clear acc A
CLRB        ; Clear acc B
COMA        ; Complement A: $FF - A --> A
INCA        ; Increment acc A
INCB        ; Increment acc B
JMP (opr)    ; Jump Address --> PC
JSR (opr)    ; Jump to subroutine (see RTS)
LDAA (opr)   ; Load Accumulator A: M-->A
LDAB
LDX
LDD
LDY
```

## *Instruction Set (cont)*

- To push and pull the contents of the stack, special instructions are used. Also, instructions to move the contents of the registers to memory are given.

```
MUL   ; Multiply registers A and B and store in D: A * B --> D
PSHA  ; Push onto stack
PSHB
PSHX
PULA  ; Pull off of stack
PULB
PULX
PULY
RTS   ; Return from sub-routine
STAA  ; Store the contents of register into memory: A--> M
STAB
STD
```

# *Assembly Language/Machine Language*

- Ultimately each instruction is represented by a number and the operands are also represented by numbers. So an entire program is a long string of binary numbers.
- Humans have difficulty reading/writing this kind of program, so we use mnemonic representations for the instructions (ADD versus \$F5 for instance).
- In addition, the registers are named and constants can be represented with symbols (not variables!).
- Certain points in memory can also be labeled to make it easy to write instructions such as (JMP END\_PROGRAM).
- The process of converting assembly code into machine code is done by the assembler. The program is loaded into memory by a loader. Some programs are written to only run in certain parts of memory but most good code should be able to run in any memory location. So the JMP instruction should be a relative offset!

## Example Programs

- Contrast writing a program in C (a high level language) with Assembly Language, which is closely related to Machine Language, the actual instructions fed into the computer.

```
int acc = 0;
for(i = 0; i < 100; i++)
{
acc += func(i);
}
```

- Versus:

```
LDA #99
CLRA ; clear contents of acc A
CLRB ; clear contents of acc B
STAB $RESULT ; clear memory address (our final result stored here)
BRANCH_POINT:
PSHA ; put argument on stack
PSHA ; this is the return value (just making room on the stack)
JSR CALC_SUB ; call the function
; return value is now on the stack
PULB ; put return value in register B
ADDB $RESULT
DECA ; increment X
BNE BRANCH_POINT ; if A is not zero, branch to start of loop
; result is now stored in address $RESULT
```



## *Higher Level Languages*

- Higher level languages such as C or C++ are translated into assembly language by a compiler. More sophisticated programs can be written this way because higher level programs abstract away all the small steps involved in a complicated calculation. For instance, in C++ you may write

```
object->Display()
```

- This involves thousands of lines of code but to understand the program, you only need to understand that the above code displays an object on the screen. You don't need to see all the small steps involved (break object into sub-objects, find x-y coordinates, translate coordinates to screen coordinates, draw lines, draw points, write to memory locations ...)

## Addressing Modes

- The operands of the instructions require the CPU to fetch data from memory. The memory location can be specified in different ways, which allows one to optimize the code for a specific application.

- Extended Addressing: The entire address is specified directly. The following instruction fetches the contents stored at \$FA0B

```
ADDA $FA0B ; fetch the value at $FA0B and add it to the value in r
```

- Direct Addressing: Only the least significant digits (hex) are specified and the most significant bits are assumed zero:

```
ADDA $F5 ; fetch the value at $00F5 and add it to the value in regi
```

- Inherent Addressing: Only access the registers

```
ABA ; add register B and A
```

## Addressing Modes (cont)

- Immediate Addressing: The operand is in the location immediately following the instruction (signified by # preceding operand)

```
ADD #$93 ; load 83 to the contents of A
```

- It's called immediate because of the way it's loaded in memory:

```
8B (op code for ADDA)
```

```
93 (the operand)
```

- Indexed Addressing: Effective address is the sum of an offset byte contained in the contents of a register (say X)

```
ADDA $03,X
```

- This loads the memory contents stored at  $X + \text{offset}$
- Relative Addressing: Used for branching. The program counter is used as the index and a relative offset is supplied.

```
BEQ -$05 ; branch if result equals zero
```

```
    ; if zero, increment PC by -5 bytes, which means we're branching back
```