Introduction to Embedded Systems

CS/ECE 6780/5780

Al Davis

Today's topics:

- some logistics updates
- a brief view of processor history
- 6812 Architecture
- introduction to Lab1

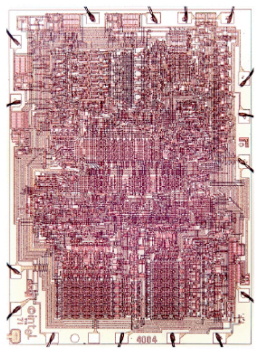
Logistics

- Acronyms
  - it's a disease and I have it
    - » you will too if you stay in the sport
  - don't hesitate to ask what the heck I'm talking about
    - » e.g. DMA question after class last Thursday
    - » lectures are intended to be interactive – don't be shy
- You should be on the mailing list ALREADY
  - if you aren't get it done TODAY or drop the class
    - » it's your choice
- Teams and lab sections
  - should be signed up by Thursday (2 days from now)
  - check out your lab kits
  - labs will start next week
    - » TA's debugging the lab write up now
      - it will be posted on the web by Thursday
        - so don't delay – it will be very hard to catch up after a slow start

My Records Indicate

- Just an fyi
  - registered but not on the mailing list
    - » Min, Najar, Sreedharan, Tateoka, Wiser, Worley
  - on mailing list but not registered
    - » Behera
- See me after class and let me know
  - if my records are wrong
  - and if not what your plans are
- So far out of a possible 42 students
  - 26 have teams
    - » 2 teams haven't given me a lab section choice yet
    - » unteamed so far
      - 6780: 3
      - 5780: 13
    - » might have the odd number problem
      - so get teams formed and let me know if you're going to drop, etc.

Intel 4004 – first single chip computer?

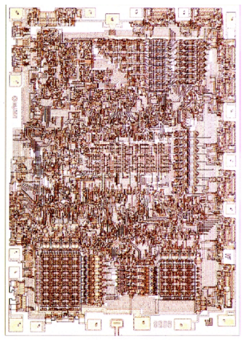


1970 – Burroughs D machine and an IBM microProc showed up around the same time

4-bit BCD

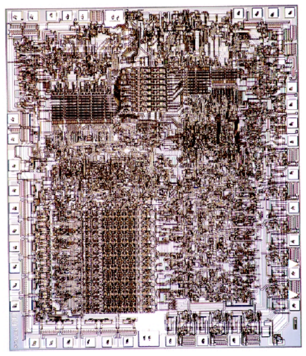
92 kHz

Intel 8008 (1972)



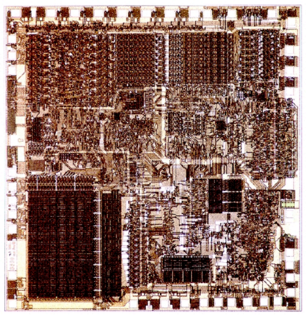
8-bit
500 kHz

Intel 8080 (1974)



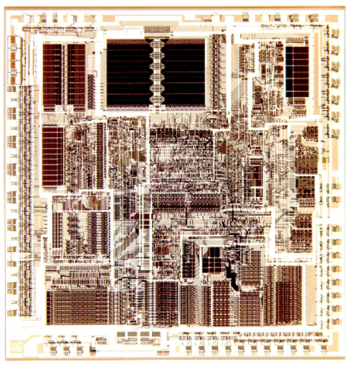
2 MHz
Considered to be the first truly usable microprocessor

Intel 8086-8088 (1978)

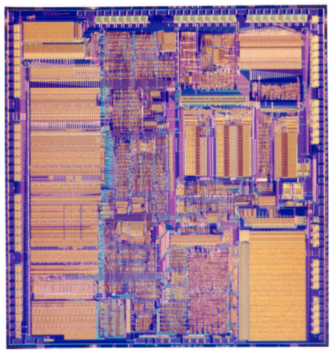


Notice anything different?

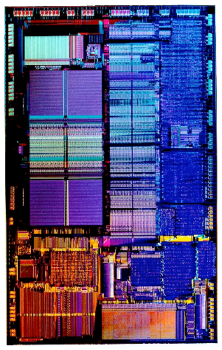
Intel 286 (1982)



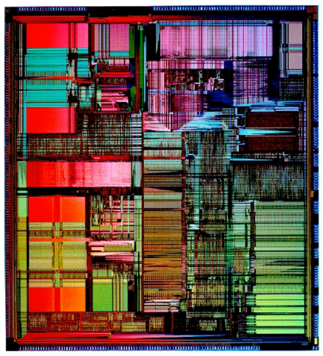
Intel 386 (1985)



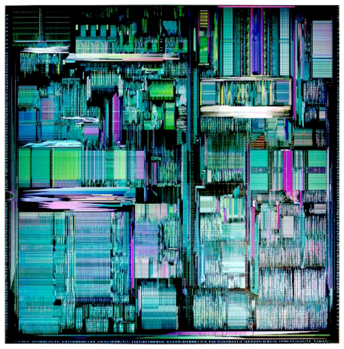
Intel486 DX (1989)



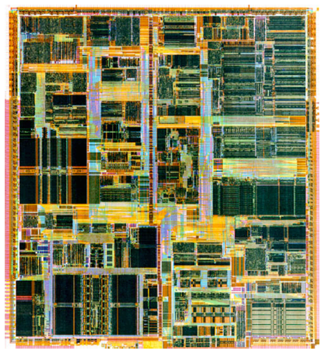
Intel Pentium (1993)



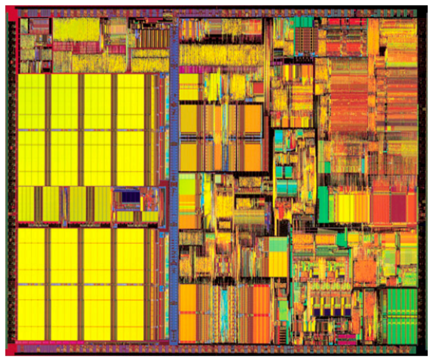
Intel Pentium Pro (1995)



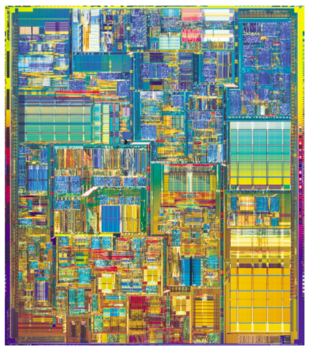
Intel Pentium II (1997)



Intel Pentium III (1999)



Intel Pentium 4 (2000)



What's the Point?

- Never ending progression
  - added architectural features
    - » simple accumulator machine
      - no such thing as virtual memory
    - » add caches
    - » add virtual memory ➔ cache translation == TLB
      - physical vs. virtual page mapping
      - segmentation also an option
    - » dynamic issue
    - » pipelining
    - » super-scalar
    - » multi-threading
    - » multiple cores
      - deeper cache hierarchy
      - coherence choices
  - added cost and power too
    - » not suitable for ES's (4004 wasn't a computer – Nehalem isn't an ES choice)

Enter Microcontrollers

- 2 ways to think about it
  - dumbed down microprocessor
  - get just what you need
    - » and not a bunch of power hungry crap that you don't
- Realization circa 1980 that ES's were necessary
  - and microprocessors were more than you needed
    - » ES's don't need the same generality
  - Intel produces the 8051 microcontroller
  - Motorola: 6805, 6808, 6811, 6812
    - » 1999 – they shipped their 2B'th MC68HC05
    - » 2004 – spins off microcontroller division
      - call it Freescale Semiconductor
      - still owned by Motorola but operates as an autonomous business unit
        - – well sort of

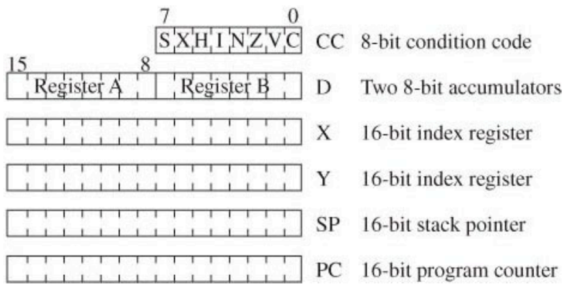
6812 Architecture

- Target
  - 16-bit data path
  - low-power → low voltage but keep bus speed high
  - single wire background debug
    - » allow in-circuit "minimally intrusive" program and debug
  - support for level language programming
    - » in C?? What a hoot?
- Lots of variants
  - biggest difference is the I/O
    - » key aspect of ES controllers
      - support for multiple standard interfaces*
  - hence pin count varies from ~60 to ~120 pins
  - amount of memory varies
  - why?
    - » target the various market segments we talked about last time
      - automotive, medical, ...

Generic 6812

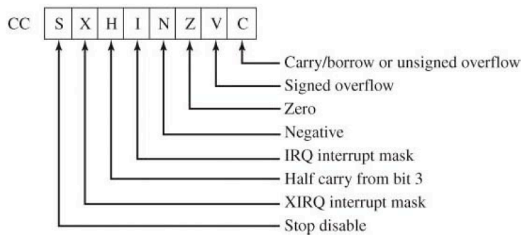
- Registers
  - 2 8-bit accumulators (called A&B)
    - » combined to form a 16-bit accumulator (D)
    - » 2 16-bit index registers (X, Y)
    - » 8-bit condition code register
    - » stack pointer and PC
  - 8-bit condition code register
  - ISA
    - » powerful bit manipulation instructions
      - not typically found in mainstream μP's
    - » arithmetic instructions
      - 16 bit +/-
      - 32 x 16 signed/unsigned divide (32? how?)
      - 16 x 16 fractional divide
      - 16 x 16 multiply
      - 32 + (16 x 16) MAC
    - » stack manipulation
      - stack pointer points to top element and grows downward

Registers



Condition Code Register

Stores critical state – important to understand how each instruction may influence this state.



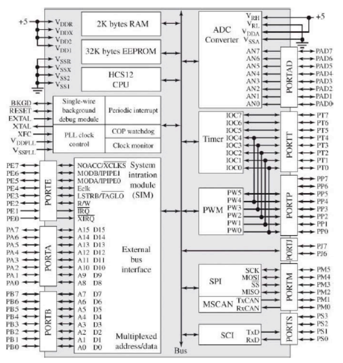
Address Map for CSM12C32

Address (hex)	Size	Device	Contents
$0000 to $03FF	1K	I/O	
$3800 to $3FFF	2K	RAM	Variables and stack
$4000 to $7FFF	16K	EEPROM	Program and constants
$C000 to $FFFF	16K	EEPROM	Program and constants

External I/O Ports

Port	48-pin	Shared Functions
Port A	PA0	Address/Data Bus
Port B	PB4	Address/Data Bus
Port E	PE7, PE4, PE1, PE0	System Integration Module
Port J	–	Key wakeup
Port M	PM5-PM0	SPI, CAN
Port P	PP5	Key wakeup, PWM
Port S	PS1-PS0	SCI
Port T	PT7-PT0	Timer, PWM
Port AD	PAD7-PAD0	Analog-to-Digital Converter

CSM12C32 Block Diagram



Numbers and Addresses

- **Byte-addressable**
  - **typical tradition (some of which is stupid in mainstream)**
- **Numbers**
  - **typical 2's complement for signed**
    - » **+/- uses same HW, divide, mult, shift are different**
    - » **everybody know this stuff?**
  - **unsigned gives greater range – assumed positive**
  - **byte = 2 hex digits in C**
    - » **10110101 = $B5 = C's version 0xB5**
    - » **also can represent a 7-bit ASCII code**
  - **programmer must keep track of signed vs. unsigned**

b7	b6	b5	b4	b3	b2	b1	b0

$$N = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \text{ (unsigned)}$$
$$N = -128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \text{ (signed)}$$

16-bit Values & Lilliputian Wars

Search "Lilliputian Wars" for an extended discussion of this problem.

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

Endian comparison for the 16-bit number $03E8:

Address	Contents
$0050	$03
$0051	$E8

Big Endian

Address	Contents
$0050	$E8
$0051	$03

Little Endian

Freescale microcontrollers use the *big endian* approach.

Fixed-Point Numbers

- **Often used in ES due to memory efficiency**
  - **FxPnum = xxx.yyyy (implicit decimal point)**
    - » **base can be decimal, binary, or whatever**
    - » **note the HW does binary**
      - **if you want something else it will have to happen in SW**
        - – **translation back and forth will be required**
  - **problem**
    - » **you bet**
    - » **obscure code is not your friend**
    - » **implied decimal point and base**
      - **often appears in the code as a comment**
      - **YIKES**

Precision, Resolution, and Range

- **Precision - # of distinguishable values**
- **Resolution – smallest representable difference**
- **Range – representable set between min and max values**
- **Example**
  - **10-bit ADC with a range of 0 to +5V**
    - » **precision of 2^{10} = 1024 values**
      - **note binary decade hack (useful if you don't already know it)**
      - **10 bits = Kilo**
      - **20 bits = Mega**
      - **30 bits = Giga**
      - **40 bits = Tera**
      - **50 bits = Peta**
      - **60 bits = Exa**
    - » **resolution of 5V/1024 = ~5mV**
    - » **representation**
      - **16-bit fixed point number**
      - **with base of 0.001V**

Overflow and Drop-Out

- **Overflow**
  - **calculated value is outside the range**
- **Drop-out**
  - **intermediate result can't be represented**
- **Example**
  - **M = (53*N)/100 vs. M = 53*(N/100)**
    - » **given fixed number of bits normal arithmetic rules change**
      - **e.g. order matters**
    - » **promotion to a higher precision avoids overflow**
    - » **dividing last avoids drop-out**

Notation

w is 8-bit signed (-128 to +127) or unsigned (0 to 255)
n is 8-bit signed (-128 to +127)
u is 8-bit unsigned (0 to 255)
W is 16-bit signed (-32787 to +32767) or unsigned (0 to 65535)
N is 16-bit signed (-32787 to +32767)
U is 16-bit unsigned (0 to 65535)
$=[addr]$ specifies an 8-bit read from address
$=\{addr\}$ specifies a 16-bit read from address (big endian)
$=< addr >$ specifies a 32-bit read from address (big endian)
$[addr]=$ specifies an 8-bit write to address
$\{addr\}=$ specifies a 16-bit write to address (big endian)
$< addr >=$ specifies a 32-bit write to address (big endian)

Assembly Language

Assembly language instructions have four fields:

Label	Opcode	Operand(s)	Comment
here	ldaa	$0000	RegA = [$0000]
	staa	$3800	[$3800] = RegA
	ldx	$3802	RegX = {$3802}
	stx	$3804	{$3804} = RegX

Assembly instructions are translated into machine code:

Object code	Instruction	Comment
$96 $00	ldaa $0000	RegA = [$0000]

Addressing Modes

An *addressing mode* is a way for an instruction to locate its operand(s)

About 80% of understanding assembly language is understanding the addressing modes

Some simple addressing modes:

Inherent addressing mode (INH)
Immediate addressing mode (IMM)
Direct page addressing mode (DIR)
Extended addressing mode (EXT)
PC relative addressing mode (REL)

Inherent Mode

Uses no operand field.

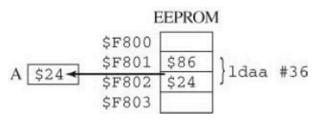
Obj code	Op	Comment
$3F	`swi`	`Software interrupt`
$87	`clra`	`RegA = 0`
$32	`pula`	`RegA = [RegSP]; RegSP=RegSP+1`

Immediate Mode

Uses a fixed constant.

Data is included in the machine code.

Obj code	Op	Operand	Comment
$8624	`ldaa`	#36	`RegA = 36`

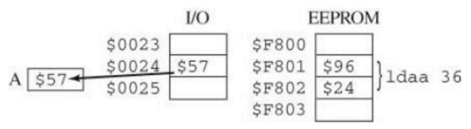


What is the difference between `ldaa #36` and `ldaa #$24`?

Direct Page Mode

Uses an 8-bit address to access from addresses $0000 to $00FF.

Obj code	Op	Operand	Comment
$9624	`ldaa`	36	`RegA = [$0024]`

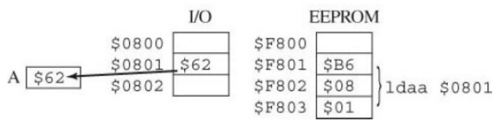


What is the difference between `ldaa #36` and `ldaa 36`?

Extended Addressing Mode

Uses a 16-bit address to access all memory and I/O devices.

Obj code	Op	Operand	Comment
$B60801	`ldaa`	$0801	`RegA = [$0801]`



PC Relative Addressing Mode

Used for branch and branch-to-subroutine instructions.
Stores 8-bit signed relative offset from current PC rather than absolute address to branch to.

rr = (destination address) − (location of branch) − (size of the branch)

Assume branch located at \$F880.

Obj code	Op	Operand	Comment
\$20BE	bra	\$F840	\$F840 − \$F880 − 2 = −\$42 = \$BE
\$2046	bra	\$F8C8	\$F8C8 − \$F880 − 2 = \$46

Lab1Example.c Requirements

- **SW1 and PB2 light up LED1 (MCU board) and LED1 and LED2 (project board) when pressed**
- **SW2 and PB1 light up LED2 (MCU) and LED3/LED4 (project) when pressed**
  - **MCU board switches and LEDs**

Application Module Student Learning Kit Users Guide (APS12C32SLKUG.pdf) contains the necessary information.

User jumpers table states that jumpers User1-4 must be on to enable the switches and LEDs (pg. 11).

Switches are active low (pg. 11).

SW1 and SW2 provide input on PORTE0 (PE0) and PORTP5 (PP5) respectively (pg. 11).

LEDs are active low (pg. 12).

LED1 and LED2 are driven by PORTA0 (PA0) and PORTB4 (PB4) respectively (pg. 12).

Project Board

MCU Project Board Student Learning Kit User Guide (PBMCUSLKUG.pdf) contains the necessary information.

Push button switches are active low (pg. 17).

PB1 and PB2 are connected to the MCU via ports 9 and 11 respectively (pg. 20).

Push buttons are enabled by a '0' on port 36 (pg. 21).

LEDs are active high (pg. 18).

LED1-LED4 are connected to the MCU via ports 33, 35, 37, and 39 respectively (pg. 20).

LEDs are enabled by a '0' on port 34 (pg. 21).

MCU Port Mappings

Board port	MCU Port	Function
9	PP5	PB1
11	PE0	PB2
33	PAD4	LED1
35	PAD5	LED2
37	PAD6	LED3
39	PAD7	LED4
34	PT4	LED_EN
36	PT5	PB_EN

Mapping found in Application Module Student Learning Kit Users Guide (APS12C32SLKUG.pdf) (pg. 11).

MCU Port Configurations

MCU Port	Direction	Config Register	Value	Function
PORTE0	Input	DDRE0 (pg. 140)	0	SW1
PORTP5	Input	DDRP5 (pg. 94)	0	SW2
PORTA0	Output	DDRA0 (pg. 136)	1	LED1
PORTB4	Output	DDRB0 (pg. 137)	1	LED2
PORTP5	Input	DDRP5 (pg. 94)	0	PB1
PORTE0	Input	DDRE0 (pg. 140)	0	PB2
PORTAD4	Output	DDRAD4 (pg. 102)	1	LED1
PORTAD5	Output	DDRAD5 (pg. 102)	1	LED2
PORTAD6	Output	DDRAD6 (pg. 102)	1	LED3
PORTAD7	Output	DDRAD7 (pg. 102)	1	LED4
PORTT4	Output	DDRT4 (pg. 82)	1	LED_EN
PORTT5	Output	DDRT5 (pg. 82)	1	PB_EN

Reference: MC9S12C Family Reference Manual (MC9S12C128V1.pdf).

Lab1Example.c Code

```
void main(void) {
  //Set the direction of ports A,B,E, and P.
  DDRA = 0xFF;
  DDRB = 0xFF;
  DDRE = 0x00;
  DDRP = 0x00;
  //Set the direction of ports T and AD
  DDRT = PTT_PTT4_MASK|PTT_PTT5_MASK;
  DDRAD = PTAD_PTAD7_MASK|PTAD_PTAD6_MASK|PTAD_PTAD5_MASK
          |PTAD_PTAD4_MASK;
  //Enable project board push buttons and LEDs
  PTT = ~(PTT_PTT4_MASK|PTT_PTT5_MASK);
}
```

Macro definitions are found in `mc9s12c32.h`.

Alternatively

```
void main(void) {
  //Set the direction of ports A,B,T,AD,E, and P.
  DDRA = 0xFF;
  DDRB = 0xFF;
  DDRE = 0x00;
  DDRP = 0x00;
  DDRT = 0xFF;
  DDRAD = 0xFF;
  //Enable project board push buttons and LEDs
  PTT = 0x00;
}
```

Lab1

- **Test your kit**
  - **same test used at end of term to demo to the TA that your kit works**
    - » **if you break it in between you're liable**
    - » **note ESD precautions**
  - **you'll load predefined code**
    - » **push buttons and switches and make sure the proper LED's do the right thing**
- **Write a simple piece of C code**
  - **4 bit Gray counter**
    - » **4 LED's indicate value**
    - » **push button or switch to increment**
    - » **typical wrap-around**
    - » **anybody not know Gray code?**

Hamming Distance 1

- **Gray code**
  - **successive values achieved by a single bit flip**
- **Karnaugh maps are your friend**

ab \ cd	00	01	11	10
00	0	1	2	3
01	7	6	5	4
11	8	9	10	11
10	15	14	13	12

N – abcd

recursive reflection
0
1
flip and add new bit
00
01
11
10
repeat as needed

MCU Programming Summary

- **Basic programming issues**
  - **simple or no data structures**
  - **simple control structures (no objects, indirect jumps, …)**
  - **lots of bit-twiddling**
- **C and assembly are almost the same**
  - **good in a way – transparent compilation**
    - » **there are some gotcha's to be covered later**
- **Key skills**
  - **debugging w/ very little feedback**
  - **low level details must be a focus**
  - **getting the right info from diverse documentation**