

Lecture 5: Machine Language

CS442: Great Insights in Computer Science
Michael L. Littman, Spring 2006

Recap

- Using logic gates, we know how to do a bunch of things with bits:
 - add
 - test equality
 - if-then-else gate
 - select one bit from a set (universal gate)

What Can We Do?

- Lots: Any function of bits, we can specify with logic gates.
- But, creating dedicated circuitry for every new problem is daunting and inefficient.
- Would like a way of using a fixed set of circuits to act like any circuitry we might want.
- Trade gates for time...

List of Squares

i	x	y	z
0	0	1	2
1	1	3	2
2	4	5	2
3	9	7	2
4	16	9	2
5	25	11	2
6	36	13	2
7	49	15	2
8	64	17	2
9	81	19	2
10	100	21	2

Recipe for squares

x takes on the value of the first 11 squares

```
x = 0
```

```
y = 1
```

```
z = 2
```

```
for i in range(0,11):
```

```
    x = x + y
```

```
    y = y + z
```

"=" means "assignment"

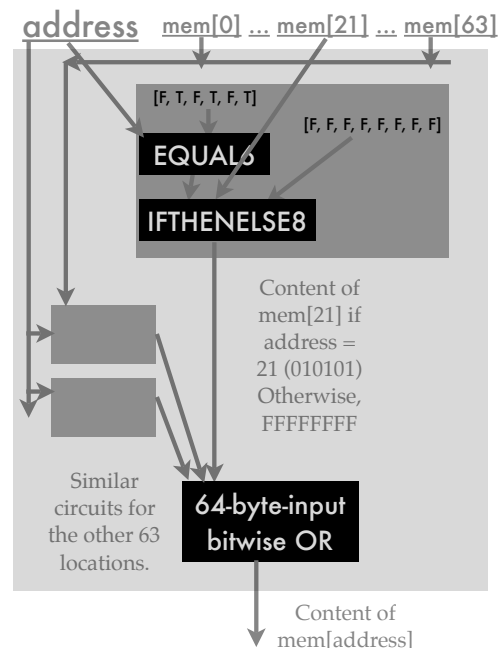
Memory

- Need a place to store the various quantities we're working with.
- Main memory is like a giant filing cabinet, where each drawer is numbered consecutively and can store one **byte**.
- Need to be able to store and retrieve values.



Memory Circuit

- Let's say there are 64 memory locations, 0-255.
- Each one has an 6-bit name called its "address".
- Memory circuit takes the contents of memory (64 x 8 bits) and an address, 518 bits in all, & outputs the byte stored at the corresponding address.

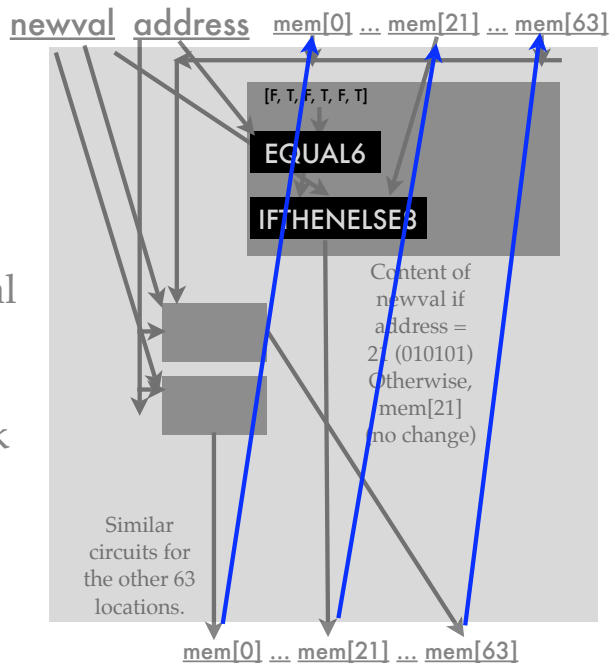


Memory Lookup

```
def memlookup(addr, mem):
    mem00 = mem[0]
    mem01 = ifthenelse8(equal8(intToByte( 1),add), mem[ 1], mem00)
    mem02 = ifthenelse8(equal8(intToByte( 2),add), mem[ 2], mem01)
    mem03 = ifthenelse8(equal8(intToByte( 3),add), mem[ 3], mem02)
    mem04 = ifthenelse8(equal8(intToByte( 4),add), mem[ 4], mem03)
    mem05 = ifthenelse8(equal8(intToByte( 5),add), mem[ 5], mem04)
    mem06 = ifthenelse8(equal8(intToByte( 6),add), mem[ 6], mem05)
    mem07 = ifthenelse8(equal8(intToByte( 7),add), mem[ 7], mem06)
    mem08 = ifthenelse8(equal8(intToByte( 8),add), mem[ 8], mem07)
    mem09 = ifthenelse8(equal8(intToByte( 9),add), mem[ 9], mem08)
    mem10 = ifthenelse8(equal8(intToByte(10),add), mem[10], mem09)
    mem11 = ifthenelse8(equal8(intToByte(11),add), mem[11], mem10)
    mem12 = ifthenelse8(equal8(intToByte(12),add), mem[12], mem11)
    mem13 = ifthenelse8(equal8(intToByte(13),add), mem[13], mem12)
    mem14 = ifthenelse8(equal8(intToByte(14),add), mem[14], mem13)
    mem15 = ifthenelse8(equal8(intToByte(15),add), mem[15], mem14)
    mem16 = ifthenelse8(equal8(intToByte(16),add), mem[16], mem15)
    mem17 = ifthenelse8(equal8(intToByte(17),add), mem[17], mem16)
    mem18 = ifthenelse8(equal8(intToByte(18),add), mem[18], mem17)
    mem19 = ifthenelse8(equal8(intToByte(19),add), mem[19], mem18)
    mem20 = ifthenelse8(equal8(intToByte(20),add), mem[20], mem19)
    mem21 = ifthenelse8(equal8(intToByte(21),add), mem[21], mem20)
    mem22 = ifthenelse8(equal8(intToByte(22),add), mem[22], mem21)
    mem23 = ifthenelse8(equal8(intToByte(23),add), mem[23], mem22)
    mem24 = ifthenelse8(equal8(intToByte(24),add), mem[24], mem23)
    mem25 = ifthenelse8(equal8(intToByte(25),add), mem[25], mem24)
    mem26 = ifthenelse8(equal8(intToByte(26),add), mem[26], mem25)
    mem27 = ifthenelse8(equal8(intToByte(27),add), mem[27], mem26)
    mem28 = ifthenelse8(equal8(intToByte(28),add), mem[28], mem27)
    mem29 = ifthenelse8(equal8(intToByte(29),add), mem[29], mem28)
    mem30 = ifthenelse8(equal8(intToByte(30),add), mem[30], mem29)
    mem31 = ifthenelse8(equal8(intToByte(31),add), mem[31], mem30)
    mem32 = ifthenelse8(equal8(intToByte(32),add), mem[32], mem31)
    mem33 = ifthenelse8(equal8(intToByte(33),add), mem[33], mem32)
    mem34 = ifthenelse8(equal8(intToByte(34),add), mem[34], mem33)
    mem35 = ifthenelse8(equal8(intToByte(35),add), mem[35], mem34)
    mem36 = ifthenelse8(equal8(intToByte(36),add), mem[36], mem35)
    mem37 = ifthenelse8(equal8(intToByte(37),add), mem[37], mem36)
    mem38 = ifthenelse8(equal8(intToByte(38),add), mem[38], mem37)
    mem39 = ifthenelse8(equal8(intToByte(39),add), mem[39], mem38)
    mem40 = ifthenelse8(equal8(intToByte(40),add), mem[40], mem39)
    mem41 = ifthenelse8(equal8(intToByte(41),add), mem[41], mem40)
    mem42 = ifthenelse8(equal8(intToByte(42),add), mem[42], mem41)
    mem43 = ifthenelse8(equal8(intToByte(43),add), mem[43], mem42)
    mem44 = ifthenelse8(equal8(intToByte(44),add), mem[44], mem43)
    mem45 = ifthenelse8(equal8(intToByte(45),add), mem[45], mem44)
    mem46 = ifthenelse8(equal8(intToByte(46),add), mem[46], mem45)
    mem47 = ifthenelse8(equal8(intToByte(47),add), mem[47], mem46)
    mem48 = ifthenelse8(equal8(intToByte(48),add), mem[48], mem47)
    mem49 = ifthenelse8(equal8(intToByte(49),add), mem[49], mem48)
    mem50 = ifthenelse8(equal8(intToByte(50),add), mem[50], mem49)
    mem51 = ifthenelse8(equal8(intToByte(51),add), mem[51], mem50)
    mem52 = ifthenelse8(equal8(intToByte(52),add), mem[52], mem51)
    mem53 = ifthenelse8(equal8(intToByte(53),add), mem[53], mem52)
    mem54 = ifthenelse8(equal8(intToByte(54),add), mem[54], mem53)
    mem55 = ifthenelse8(equal8(intToByte(55),add), mem[55], mem54)
    mem56 = ifthenelse8(equal8(intToByte(56),add), mem[56], mem55)
    mem57 = ifthenelse8(equal8(intToByte(57),add), mem[57], mem56)
    mem58 = ifthenelse8(equal8(intToByte(58),add), mem[58], mem57)
    mem59 = ifthenelse8(equal8(intToByte(59),add), mem[59], mem58)
    mem60 = ifthenelse8(equal8(intToByte(60),add), mem[60], mem59)
    mem61 = ifthenelse8(equal8(intToByte(61),add), mem[61], mem60)
    mem62 = ifthenelse8(equal8(intToByte(62),add), mem[62], mem61)
    mem63 = ifthenelse8(equal8(intToByte(63),add), mem[63], mem62)
    return mem63
```

Writing to Memory

- Similar circuit allows memory cells to be altered.
- $mem[address] = newval$
- If needed for future processing, copied back up at the end of the cycle.



Memory Write

```
def memwrite(active, add, mem, val):
    return [
        ifthenelse8(active and equal8(intoByte( 0),add), val, mem[ 0]),
        ifthenelse8(active and equal8(intoByte( 1),add), val, mem[ 1]),
        ifthenelse8(active and equal8(intoByte( 2),add), val, mem[ 2]),
        ifthenelse8(active and equal8(intoByte( 3),add), val, mem[ 3]),
        ifthenelse8(active and equal8(intoByte( 4),add), val, mem[ 4]),
        ifthenelse8(active and equal8(intoByte( 5),add), val, mem[ 5]),
        ifthenelse8(active and equal8(intoByte( 6),add), val, mem[ 6]),
        ifthenelse8(active and equal8(intoByte( 7),add), val, mem[ 7]),
        ifthenelse8(active and equal8(intoByte( 8),add), val, mem[ 8]),
        ifthenelse8(active and equal8(intoByte( 9),add), val, mem[ 9]),
        ifthenelse8(active and equal8(intoByte(10),add), val, mem[10]),
        ifthenelse8(active and equal8(intoByte(11),add), val, mem[11]),
        ifthenelse8(active and equal8(intoByte(12),add), val, mem[12]),
        ifthenelse8(active and equal8(intoByte(13),add), val, mem[13]),
        ifthenelse8(active and equal8(intoByte(14),add), val, mem[14]),
        ifthenelse8(active and equal8(intoByte(15),add), val, mem[15]),
        ifthenelse8(active and equal8(intoByte(16),add), val, mem[16]),
        ifthenelse8(active and equal8(intoByte(17),add), val, mem[17]),
        ifthenelse8(active and equal8(intoByte(18),add), val, mem[18]),
        ifthenelse8(active and equal8(intoByte(19),add), val, mem[19]),
        ifthenelse8(active and equal8(intoByte(20),add), val, mem[20]),
        ifthenelse8(active and equal8(intoByte(21),add), val, mem[21]),
        ifthenelse8(active and equal8(intoByte(22),add), val, mem[22]),
        ifthenelse8(active and equal8(intoByte(23),add), val, mem[23]),
        ifthenelse8(active and equal8(intoByte(24),add), val, mem[24]),
        ifthenelse8(active and equal8(intoByte(25),add), val, mem[25]),
        ifthenelse8(active and equal8(intoByte(26),add), val, mem[26]),
        ifthenelse8(active and equal8(intoByte(27),add), val, mem[27]),
        ifthenelse8(active and equal8(intoByte(28),add), val, mem[28]),
        ifthenelse8(active and equal8(intoByte(29),add), val, mem[29]),
        ifthenelse8(active and equal8(intoByte(30),add), val, mem[30]),
        ifthenelse8(active and equal8(intoByte(31),add), val, mem[31]),
        ifthenelse8(active and equal8(intoByte(32),add), val, mem[32]),
        ifthenelse8(active and equal8(intoByte(33),add), val, mem[33]),
        ifthenelse8(active and equal8(intoByte(34),add), val, mem[34]),
        ifthenelse8(active and equal8(intoByte(35),add), val, mem[35]),
        ifthenelse8(active and equal8(intoByte(36),add), val, mem[36]),
        ifthenelse8(active and equal8(intoByte(37),add), val, mem[37]),
        ifthenelse8(active and equal8(intoByte(38),add), val, mem[38]),
        ifthenelse8(active and equal8(intoByte(39),add), val, mem[39]),
        ifthenelse8(active and equal8(intoByte(40),add), val, mem[40]),
        ifthenelse8(active and equal8(intoByte(41),add), val, mem[41]),
        ifthenelse8(active and equal8(intoByte(42),add), val, mem[42]),
        ifthenelse8(active and equal8(intoByte(43),add), val, mem[43]),
        ifthenelse8(active and equal8(intoByte(44),add), val, mem[44]),
        ifthenelse8(active and equal8(intoByte(45),add), val, mem[45]),
        ifthenelse8(active and equal8(intoByte(46),add), val, mem[46]),
        ifthenelse8(active and equal8(intoByte(47),add), val, mem[47]),
        ifthenelse8(active and equal8(intoByte(48),add), val, mem[48]),
        ifthenelse8(active and equal8(intoByte(49),add), val, mem[49]),
        ifthenelse8(active and equal8(intoByte(50),add), val, mem[50]),
        ifthenelse8(active and equal8(intoByte(51),add), val, mem[51]),
        ifthenelse8(active and equal8(intoByte(52),add), val, mem[52]),
        ifthenelse8(active and equal8(intoByte(53),add), val, mem[53]),
        ifthenelse8(active and equal8(intoByte(54),add), val, mem[54]),
        ifthenelse8(active and equal8(intoByte(55),add), val, mem[55]),
        ifthenelse8(active and equal8(intoByte(56),add), val, mem[56]),
        ifthenelse8(active and equal8(intoByte(57),add), val, mem[57]),
        ifthenelse8(active and equal8(intoByte(58),add), val, mem[58]),
        ifthenelse8(active and equal8(intoByte(59),add), val, mem[59]),
        ifthenelse8(active and equal8(intoByte(60),add), val, mem[60]),
        ifthenelse8(active and equal8(intoByte(61),add), val, mem[61]),
        ifthenelse8(active and equal8(intoByte(62),add), val, mem[62]),
        ifthenelse8(active and equal8(intoByte(63),add), val, mem[63])]
```

Memory for Variables

We'll use memory locations to represent the variables:
x is mem[35], y is mem[36], z is mem[37], i is mem[40]

```
x = 0 ..... mem[35] = 0
y = 1 ..... mem[36] = 1
z = 2 ..... mem[37] = 2
          ..... mem[40] = 0
for i in range(0,11): LOOP: if mem[40] == 11: goto LOOP
  x = x + y ..... mem[35] = mem[35] + mem[36]
  y = y + z ..... mem[36] = mem[36] + mem[37]
                ..... goto LOOP
```

Simplify Statements

- We'd like circuits to implement each statement.
- Too many possible statements: We can convert statements into sequences of simpler statements.

```
x = x + y
mem[35] = mem[35] + mem[36]
```

We create a special memory location called the "accumulator" (abbreviated "acc") to hold intermediate results.

```
acc = 0
acc = acc + mem[35]
acc = acc + mem[36]
mem[35] = acc
```

Simplified Statements

```
mem[35] = 0
mem[36] = 1
mem[37] = 2
mem[40] = 0
LOOP: if mem[40] == 11: goto LOOP
  mem[35] = mem[35] + mem[36]
  mem[36] = mem[36] + mem[37]
  goto LOOP
```

```
acc = 0
mem[35] = acc
acc = 1
mem[36] = acc
acc = 2
mem[37] = acc
acc = 0
mem[40] = acc
LOOP: acc = -11
  acc = acc + mem[40]
  if acc == 0: goto LOOP
acc = 0
acc = acc + mem[35]
acc = acc + mem[36]
mem[35] = acc
acc = 0
acc = acc + mem[36]
acc = acc + mem[37]
mem[36] = acc
acc = 0
if acc == 0: goto LOOP
```

Instruction List

- We've whittled the program down to 4 kinds of statements:
 - $\text{acc} = \text{constant}$
 - $\text{acc} = \text{acc} + \text{mem}[\text{constant}]$
 - $\text{mem}[\text{constant}] = \text{acc}$
 - if $\text{acc} == 0$: goto constant
- We can build dedicated circuits for a small set like this.

One Instruction Per Byte

b7 b6 b5 b4 b3 b2 b1 b0

instruction (2 bits) constant (6 bits)

- 00: $\text{acc} = \text{constant}$
- 01: $\text{acc} = \text{acc} + \text{mem}[\text{constant}]$
- 10: if $\text{acc} == 0$: goto constant
- 11: $\text{mem}[\text{constant}] = \text{acc}$

01101000

- instruction = 01
- constant = 101000 = 40
- So, $\text{acc} = \text{acc} + \text{mem}[40]$

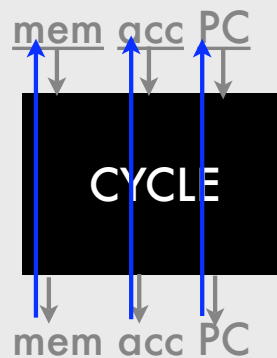
Program is Series of Bytes

address	contents (decimal)	contents (binary)	contents (instruction)
0	0	00000000	ACC = 0
1	227	11100011	mem[35] = ACC
2	1	00000001	ACC = 1
3	228	11101100	mem[36] = ACC
4	2	00000010	ACC = 2
5	229	11100101	mem[37] = ACC
6	0	00000000	ACC = 0
7	232	11101000	mem[40] = ACC
8	53	00110101	ACC = 53
9	104	01101000	ACC = ACC + mem[40]
10	138	10001010	if ACC == 0: PC = 10
11	63	00111111	ACC = 63
12	104	01101000	ACC = ACC + mem[40]
13	232	11101000	mem[40] = ACC
14	0	00000000	ACC = 0
15	99	01100011	ACC = ACC + mem[35]
16	100	01100100	ACC = ACC + mem[36]

Michael Littman's
Minimal Length
Machine Language
(ML³)

von Neumann Architecture

- A computer is just a big logic gate.
- Input: registers, memory
- Output: new values for registers, memory
- PC = Program counter, the address of the statement to be executed.

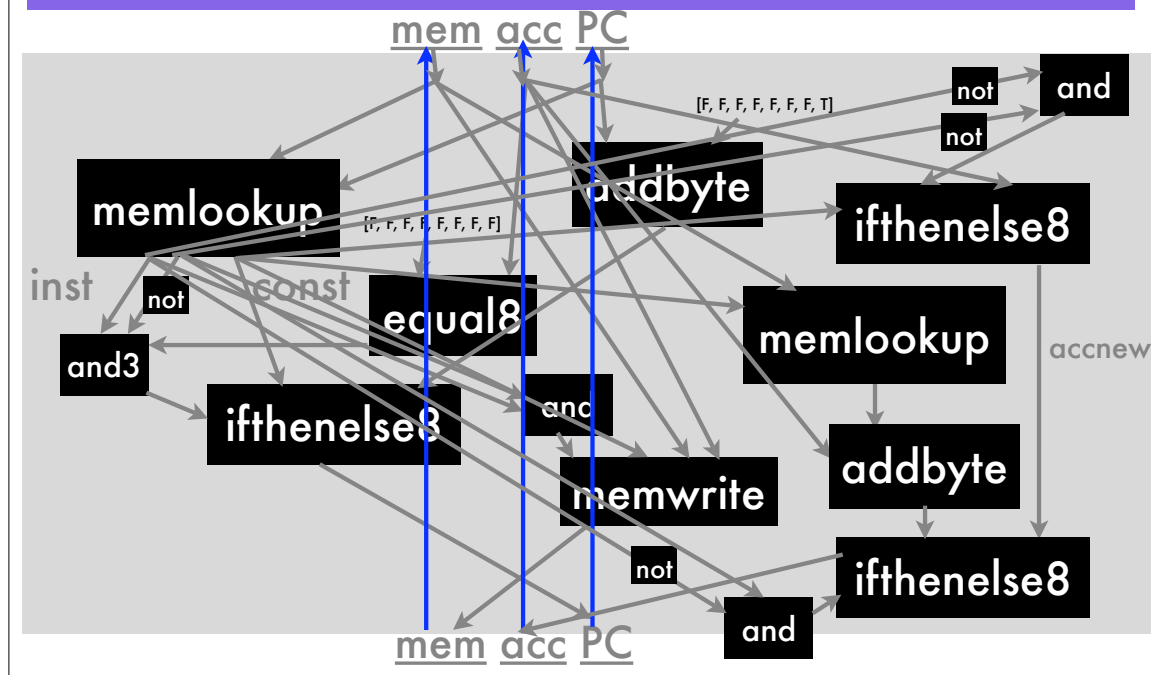


8x64 bits 8 bits
8 bits

528 bits total

CPU = Central
Processing Unit

Cycle: A Whole Computer



Cycle

```
def cycle(input):
    [mem,pc,acc] = input
    ir = memlookup(pc,mem)
    pcnew = ifthenelse8((ir[0] and not ir[1]) and equal8(acc, intToByte(0)),
        [False, False, ir[2], ir[3], ir[4], ir[5], ir[6], ir[7]],
        addbyte(pc, intToByte(1)))
    # note the sign extension! (to allow for negation)
    accnew = ifthenelse8(not ir[0] and not ir[1],
        [ir[2], ir[2], ir[2], ir[3], ir[4], ir[5], ir[6], ir[7]],
        acc)
    accnewnew = ifthenelse8(not ir[0] and ir[1],
        addbyte(acc, memlookup([False, False, ir[2], ir[3], ir[4], ir[5], ir[6], ir[7]],mem)),
        accnew)
    memnew = memwrite(ir[0] and ir[1],[False, False, ir[2], ir[3], ir[4], ir[5], ir[6], ir[7]],mem,acc)
    return [memnew,pcnew,accnewnew]
```

- DEMO

Instruction Sets

- ML³ used a particular design that made it relatively easy to fit in a lecture slide.
- Computer manufacturers have different goals in mind: cost, speed, ease of running modern programs.
- Some quick examples:

x86: Intel's Old Set

AAA: Ascii Adjust for Addition	INC: Increment	NEG: Two's Complement Negation	SETL/SETNG: Set if Less or Equal / Set if Not greater or Equal (386+)
AAD: Ascii Adjust for Division	INS: Input String from Port (80188+)	NOP: No Operation (90h)	SETG/SETNLE: Set if Greater / Set if Not Less or Equal (386+)
AAM: Ascii Adjust for Multiplication	INT: Interrupt	NOT: One's Complement Negation (Logical NOT)	SETO/SETNLS: Set if Signed (386+)
AAS: Ascii Adjust for Subtraction	INTO: Interrupt on Overflow	OR: Inclusive Logical OR	SETNS: Set if Not Signed (386+)
ADD: Add With Carry	INVD: Invalidate Cache (486+)	OUT: Output Data to Port	SETC: Set if Carry (386+)
ADD: Arithmetic Addition	INVLPG: Invalidate Translation Look-Aside Buffer Entry (486+)	OUTS: Output String to Port (80188+)	SETNC: Set if Not Carry (386+)
AND: Logical And	IRET/IRETD: Interrupt Return	POP: Pop Word off Stack	SETO: Set if Overflow (386+)
ARPL: Adjusted Requested Privilege Level of Selector (286+ PM)	Jxx: Jump Instructions Table	POPA/POPAD: Pop All Registers onto Stack (80188+)	SETNO: Set if Not Overflow (386+)
BOUND: Array Index Bound Check (80188+)	JCXZ/JECXZ: Jump if Register [E]CX is Zero	POPF/POPPD: Pop Flags off Stack	SETP/SETPE: Set if Parity / Set if Parity Even (386+)
BSF: Bit Scan Forward (386+)	JMP: Unconditional Jump	PUSH: Push Word onto Stack	SETNP/SETPO: Set if No Parity / Set if Parity Odd (386+)
BSR: Bit Scan Reverse (386+)	LAHF: Load Register AH From Flags	PUSHA/PUSHAD: Push All Registers onto Stack (80188+)	SGDT: Store Global Descriptor Table (286+ privileged)
BSWAP: Byte Swap (486+)	LAR: Load Access Rights (286+ protected)	RCL: Rotate Through Carry Left	SIDT: Store Interrupt Descriptor Table (286+ privileged)
BT: Bit Test (386+)	LDS: Load Pointer Using DS	RCR: Rotate Through Carry Right	SHL: Shift Logical Left
BTC: Bit Test with Complement (386+)	LEA: Load Effective Address	REP: Repeat String Operation	SHR: Shift Logical Right
BTR: Bit Test with Reset (386+)	LEAVE: Restore Stack for Procedure Exit (80188+)	REPNE/REPNZ: Repeat Equal / Repeat Zero Not Zero	SHLD/SHRD: Double Precision Shift (386+ privileged)
BTS: Bit Test and Set (386+)	LES: Load Pointer Using ES	REPE/REPZ: Repeat Equal / Repeat Zero	SIDT: Store Local Descriptor Table (286+ privileged)
CALL: Procedure Call	LFS: Load Pointer Using FS (386+)	REPNE/REPZ: Repeat Not Equal / Repeat Not Zero	SMSW: Store Machine Status Word (286+ privileged)
CBW: Convert Byte to Word	LGDT: Load Global Descriptor Table (286+ privileged)	RET/RETF: Return From Procedure	STC: Set Carry
CDQ: Convert Double to Quad (386+)	LIDT: Load Local Descriptor Table (286+ privileged)	ROL: Rotate Left	STD: Set Direction Flag
CLC: Clear Carry	LIGS: Load Pointer Using GS (386+)	ROR: Rotate Right	STI: Set Interrupt Flag (Enable Interrupts)
CLD: Clear Direction Flag	LJDT: Load Local Descriptor Table (286+ privileged)	SAHF: Store AH Register into FLAGS	STOS: Store String (Byte, Word or Doubleword)
CLI: Clear Interrupt Flag (disable)	LMSW: Load Machine Status Word (286+ privileged)	SAL/SHL: Shift Arithmetic Left / Shift Logical Left	STR: Store Task Register (286+ privileged)
CLTS: Clear Task Switched Flag (286+ privileged)	LOCK: Lock Bus	SAR: Shift Arithmetic Right	SUB: Subtract
CMC: Complement Carry Flag	LODS: Load String (Byte, Word or Double)	SBB: Subtract with Borrow/Carry	TEST: Test For Bit Pattern
CMP: Compare	LOOP: Decrement CX and Loop if CX Not Zero	SCAS: Scan String (Byte, Word or Doubleword)	VERR: Verify Read (286+ protected)
CMPS: Compare String (Byte, Word or Doubleword)	LOOPE/LOOPZ: Loop While Equal / Loop While Zero	SETAE/SETNB: Set if Above or Equal / Set if Not Below (386+)	VERW: Verify Write (286+ protected)
CMPSCHG: Compare and Exchange	LOPNZ/LOPNE: Loop While Not Zero / Loop While Not Equal	SETBE/SETNAE: Set if Below or Equal / Set if Not Above (386+)	WAIT/FWAIT: Event Wait
CWD: Convert Word to Doubleword	LSSL: Load Segment Limit (286+ protected)	SETB/SETNB: Set if Below or Equal / Set if Not Below (386+)	WBINVD: Write-Back and Invalidate Cache (486+)
CWDE: Convert Word to Extended Doubleword (386+)	LSS: Load Pointer Using SS (386+)	SETE/SETZ: Set if Equal / Set if Zero (386+)	XCHG: Exchange
DAA: Decimal Adjust for Addition	LTR: Load Task Register (286+ privileged)	SETNE/SETNZ: Set if Not Equal / Set if Not Zero (386+)	XLAT/XLATB: Translate
DAS: Decimal Adjust for Subtraction	MOV: Move Byte or Word	SETNL/SETNL: Set if Not Less or Equal / Set if Not Less (386+)	XOR: Exclusive OR
DEC: Decrement	MOVS: Move String (Byte or Word)		
DIV: Divide	MOVSB: Move with Sign Extend (386+)		
ENTER: Make Stack Frame (80188+)	MOVSW: Move with Zero Extend (386+)		
ESC: Escape	MUL: Unsigned Multiply		
HLT: Halt CPU			
IDIV: Signed Integer Division			
IMUL: Signed Multiply			
IN: Input Byte or Word From Port			

Next Time

- Subroutines, recursion.
- Hillis Chapter 3, first section.