Lecture 5: Machine Language

CS105: Great Insights in Computer Science

Michael L. Littman, Fall 2006

Recap

- Using logic gates, we know how to do a bunch of things with bits:
  - 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.
- We can use the state-machine idea to trade gates for time...

Concrete Example: Adding

```
  10          c1c0
   11           x1x0
+  10          +y1y0
  101         z2z1z0
```

- We want to compute the sum of x and y (2-bit numbers). z (3 bits) is the answer and c (2 bits) is the carry.
- z_0 = (x_0 and not y_0) or (not x_0 and y_0)
- c_0 = (x_0 and y_0)

Concrete Example: Adding

```
  10        c1c0
   11         x1x0
+  10       +y1y0
  101       z2z1z0
```

- z_1 = (x_1 and not y_1 and not c_0) or (not x_1 and y_1 and not c_0) or (not x_1 and not y_1 and c_0) or (x_1 and y_1 and c_0)
- $c_1 = z_2$ = (x_1 and y_1 and not c_0) or (x_1 and not y_1 and c_0) or (not x_1 and y_1 and c_0) or (x_1 and y_1 and c_0)

Adding Bytes

- Computing z_i and c_i from x_i, y_i, and c_{i-1} can be carried out with 6 ands, 3 ors, 4 nots.
- The previous slide uses 16 ands, 6 ors, and 9 nots (not as good).
- This operation is called a "full adder".
- By chaining together one full adder per bit, we can make a circuit that adds any number of bits (4, 8, 16, 32, 64, etc.).

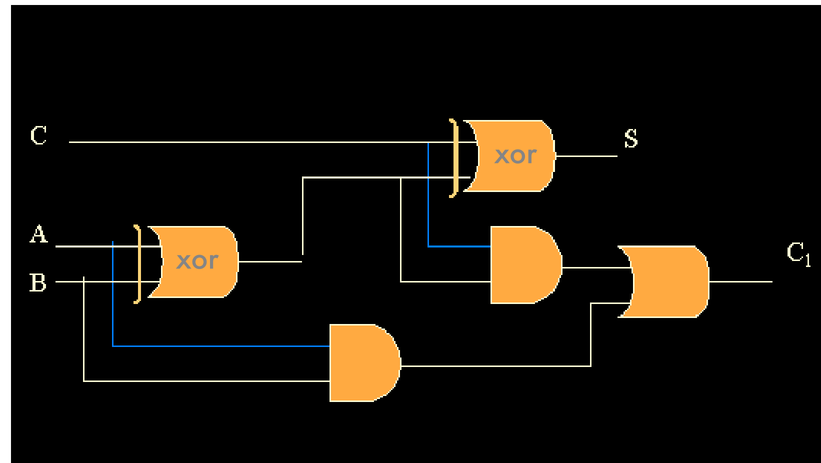
Hardware

- So, any function we want to implement from bits to bits can be carried out by constructing the right circuit of and/or/nots.
- Creating a circuit solves the problem "in hardware".
- The advantage of hardware solutions are that they are fast.
- The disadvantage is that they are inflexible.

Software

- The lovely thing about a computer is that the hardware does not have to change for the computer to change its behavior.
- A fixed set of circuits can actually change its behavior to represent any desired function!
- Build one, reprogram into anything.
- Disadvantage of the software approach: Can be much slower.

Programming an Adder



Circuit level:

Instruction level:

- H = A xor B = *(A and not B) or (not A and B)*
- C_1 = (A and B) or (C and H)
- S = H xor C = *(H and not C) or (not H and C)*

Simple Statements

- Still too many different statements.
- Break complex statements down into a set of simple statements.
- Instead of E = (H and not C) or (not H and C):

- acc = not C
- acc = acc and H
- P = acc
- acc = not H
- acc = acc and C
- acc = acc or P
- E = acc

Instruction Set: 7 Bits

- **V** in 0000...1111 (variables A- P)
- **acc**: special temporary variable
- **000V**: acc = acc and V
- **001V**: acc = acc or V
- **010V**: acc = V
- **011V**: acc = not V
- **100V**: V = acc and V
- **101V**: V = acc or V
- **110V**: V = acc
- **111V**: V = not acc

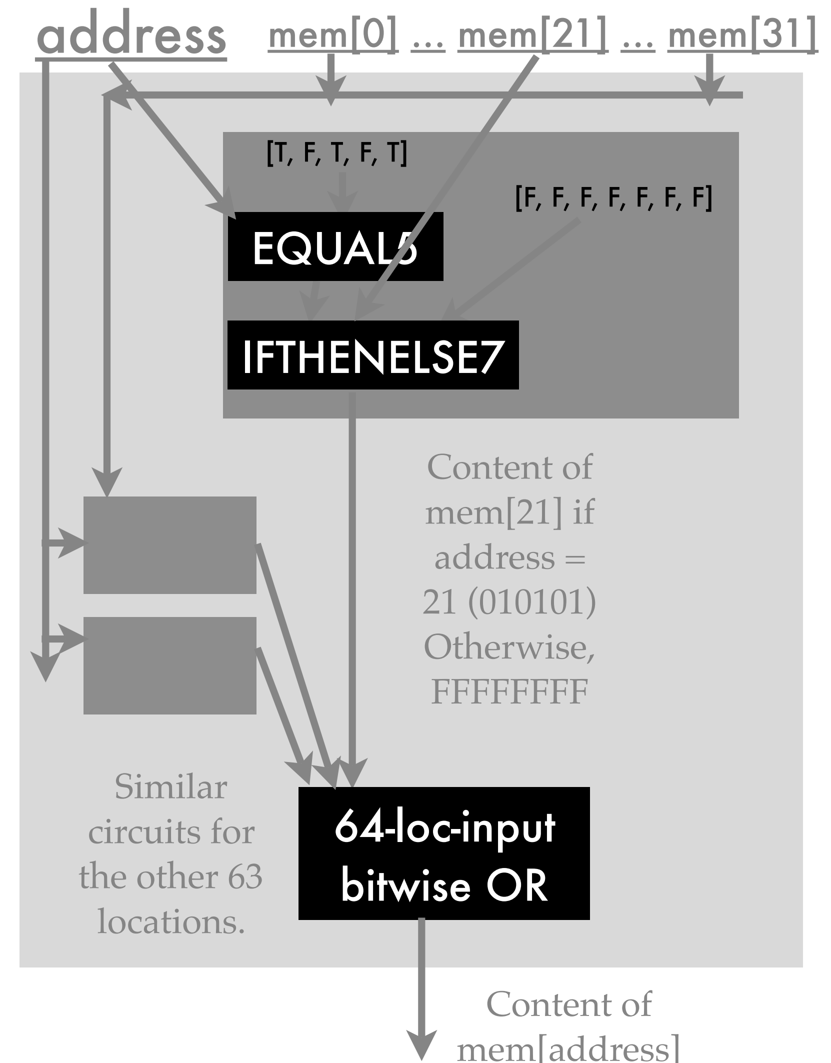
0000 A	0010 C	0100 E	0110 G	1000 I	1010 K	1100 M	1110 O
0001 B	0011 D	0101 F	0111 H	1001 J	1011 L	1101 N	1111 P

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 **value**.
- Need to be able to store and retrieve values.

Memory Circuit

- We'll have 32 memory locations, for instructions.
- Each one has an 5-bit name (0-31) called its "address".
- Memory circuit takes the contents of memory (32 x 7 bits) and an address, 229 bits in all, & outputs the data stored at the corresponding address.

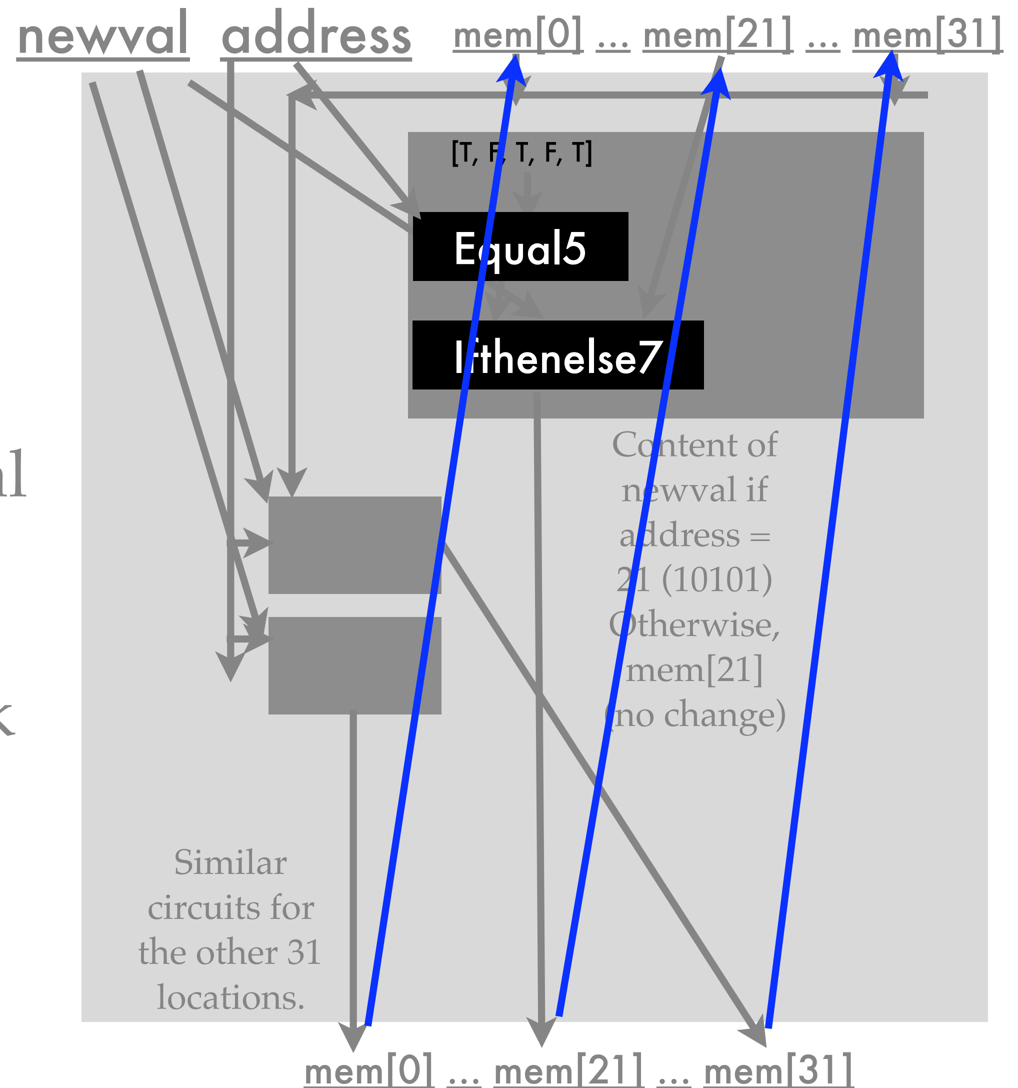


Memory Lookup

```
def memlookup(add, mem):
    mem00 = mem[0]
    mem01 = ifthenelse7(equal5(intToByte( 1),add), mem[ 1], mem00)
    mem02 = ifthenelse7(equal5(intToByte( 2),add), mem[ 2], mem01)
    mem03 = ifthenelse7(equal5(intToByte( 3),add), mem[ 3], mem02)
    mem04 = ifthenelse7(equal5(intToByte( 4),add), mem[ 4], mem03)
    mem05 = ifthenelse7(equal5(intToByte( 5),add), mem[ 5], mem04)
    mem06 = ifthenelse7(equal5(intToByte( 6),add), mem[ 6], mem05)
    mem07 = ifthenelse7(equal5(intToByte( 7),add), mem[ 7], mem06)
    mem08 = ifthenelse7(equal5(intToByte( 8),add), mem[ 8], mem07)
    mem09 = ifthenelse7(equal5(intToByte( 9),add), mem[ 9], mem08)
    mem10 = ifthenelse7(equal5(intToByte(10),add), mem[10], mem09)
    mem11 = ifthenelse7(equal5(intToByte(11),add), mem[11], mem10)
    mem12 = ifthenelse7(equal5(intToByte(12),add), mem[12], mem11)
    mem13 = ifthenelse7(equal5(intToByte(13),add), mem[13], mem12)
    mem14 = ifthenelse7(equal5(intToByte(14),add), mem[14], mem13)
    mem15 = ifthenelse7(equal5(intToByte(15),add), mem[15], mem14)
    mem16 = ifthenelse7(equal5(intToByte(16),add), mem[16], mem15)
    mem17 = ifthenelse7(equal5(intToByte(17),add), mem[17], mem16)
    mem18 = ifthenelse7(equal5(intToByte(18),add), mem[18], mem17)
    mem19 = ifthenelse7(equal5(intToByte(19),add), mem[19], mem18)
    mem20 = ifthenelse7(equal5(intToByte(20),add), mem[20], mem19)
    mem21 = ifthenelse7(equal5(intToByte(21),add), mem[21], mem20)
    mem22 = ifthenelse7(equal5(intToByte(22),add), mem[22], mem21)
    mem23 = ifthenelse7(equal5(intToByte(23),add), mem[23], mem22)
    mem24 = ifthenelse7(equal5(intToByte(24),add), mem[24], mem23)
    mem25 = ifthenelse7(equal5(intToByte(25),add), mem[25], mem24)
    mem26 = ifthenelse7(equal5(intToByte(26),add), mem[26], mem25)
    mem27 = ifthenelse7(equal5(intToByte(27),add), mem[27], mem26)
    mem28 = ifthenelse7(equal5(intToByte(28),add), mem[28], mem27)
    mem29 = ifthenelse7(equal5(intToByte(29),add), mem[29], mem28)
    mem30 = ifthenelse7(equal5(intToByte(30),add), mem[30], mem29)
    mem31 = ifthenelse7(equal5(intToByte(31),add), mem[31], mem30)
return mem31
```

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 [
        ifthenelse7(active and equal5(intToByte( 0),add), val, mem[ 0]),
        ifthenelse7(active and equal5(intToByte( 1),add), val, mem[ 1]),
        ifthenelse7(active and equal5(intToByte( 2),add), val, mem[ 2]),
        ifthenelse7(active and equal5(intToByte( 3),add), val, mem[ 3]),
        ifthenelse7(active and equal5(intToByte( 4),add), val, mem[ 4]),
        ifthenelse7(active and equal5(intToByte( 5),add), val, mem[ 5]),
        ifthenelse7(active and equal5(intToByte( 6),add), val, mem[ 6]),
        ifthenelse7(active and equal5(intToByte( 7),add), val, mem[ 7]),
        ifthenelse7(active and equal5(intToByte( 8),add), val, mem[ 8]),
        ifthenelse7(active and equal5(intToByte( 9),add), val, mem[ 9]),
        ifthenelse7(active and equal5(intToByte(10),add), val, mem[10]),
        ifthenelse7(active and equal5(intToByte(11),add), val, mem[11]),
        ifthenelse7(active and equal5(intToByte(12),add), val, mem[12]),
        ifthenelse7(active and equal5(intToByte(13),add), val, mem[13]),
        ifthenelse7(active and equal5(intToByte(14),add), val, mem[14]),
        ifthenelse7(active and equal5(intToByte(15),add), val, mem[15]),
        ifthenelse7(active and equal5(intToByte(16),add), val, mem[16]),
        ifthenelse7(active and equal5(intToByte(17),add), val, mem[17]),
        ifthenelse7(active and equal5(intToByte(18),add), val, mem[18]),
        ifthenelse7(active and equal5(intToByte(19),add), val, mem[19]),
        ifthenelse7(active and equal5(intToByte(20),add), val, mem[20]),
        ifthenelse7(active and equal5(intToByte(21),add), val, mem[21]),
        ifthenelse7(active and equal5(intToByte(22),add), val, mem[22]),
        ifthenelse7(active and equal5(intToByte(23),add), val, mem[23]),
        ifthenelse7(active and equal5(intToByte(24),add), val, mem[24]),
        ifthenelse7(active and equal5(intToByte(25),add), val, mem[25]),
        ifthenelse7(active and equal5(intToByte(26),add), val, mem[26]),
        ifthenelse7(active and equal5(intToByte(27),add), val, mem[27]),
        ifthenelse7(active and equal5(intToByte(28),add), val, mem[28]),
        ifthenelse7(active and equal5(intToByte(29),add), val, mem[29]),
        ifthenelse7(active and equal5(intToByte(30),add), val, mem[30]),
        ifthenelse7(active and equal5(intToByte(31),add), val, mem[31])]
```

Persistence of Memory

- We can use this memory idea to store the Boolean variables (A-P).
- We can also use another set of memory locations to store the series of instructions to be executed(program).
- How is are the instructions stored?

One Instruction

b6 b5 b4 b3 b2 b1 b0

load/store (1 bit)

- 0: load; 1: store

instruction (2 bits)

- 00: acc **or** V
- 01: acc **and** V
- 10: acc (load) / V (store)
- 11: **not** acc (load) / not V (store)

variable name (4 bits)

1011000

- store = 1
- instruction = 01
- constant = 1000 = I
- So, “I = acc and I”

Series of Instructions

contents (decimal)

contents (binary)

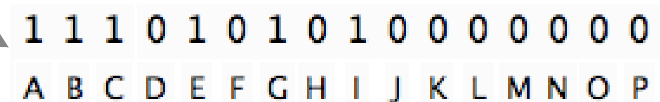
Michael Littman's
Mini Logic
Machine Language
(ML^3)

address

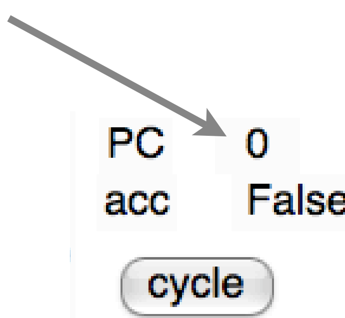
address	contents (decimal)	contents (binary)	contents (instruction)
0	51	0110011	acc = not D
1	17	0010001	acc = acc and B
2	111	1101111	P = acc
3	49	0110001	acc = not B
4	19	0010011	acc = acc and D
5	15	0001111	acc = acc or P
6	104	1101000	I = acc
7	33	0100001	acc = B
8	19	0010011	acc = acc and D

contents (instruction)

Registers: Boolean variables and their values



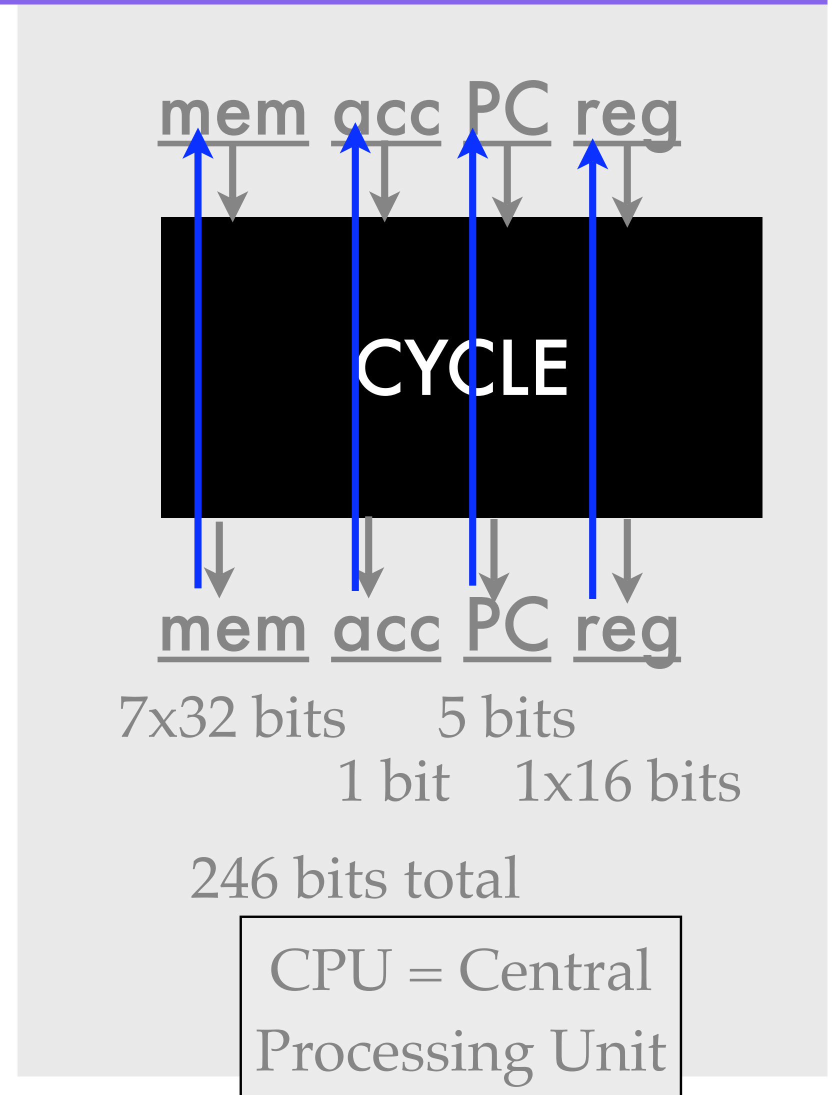
Program counter: which address's instruction to process next



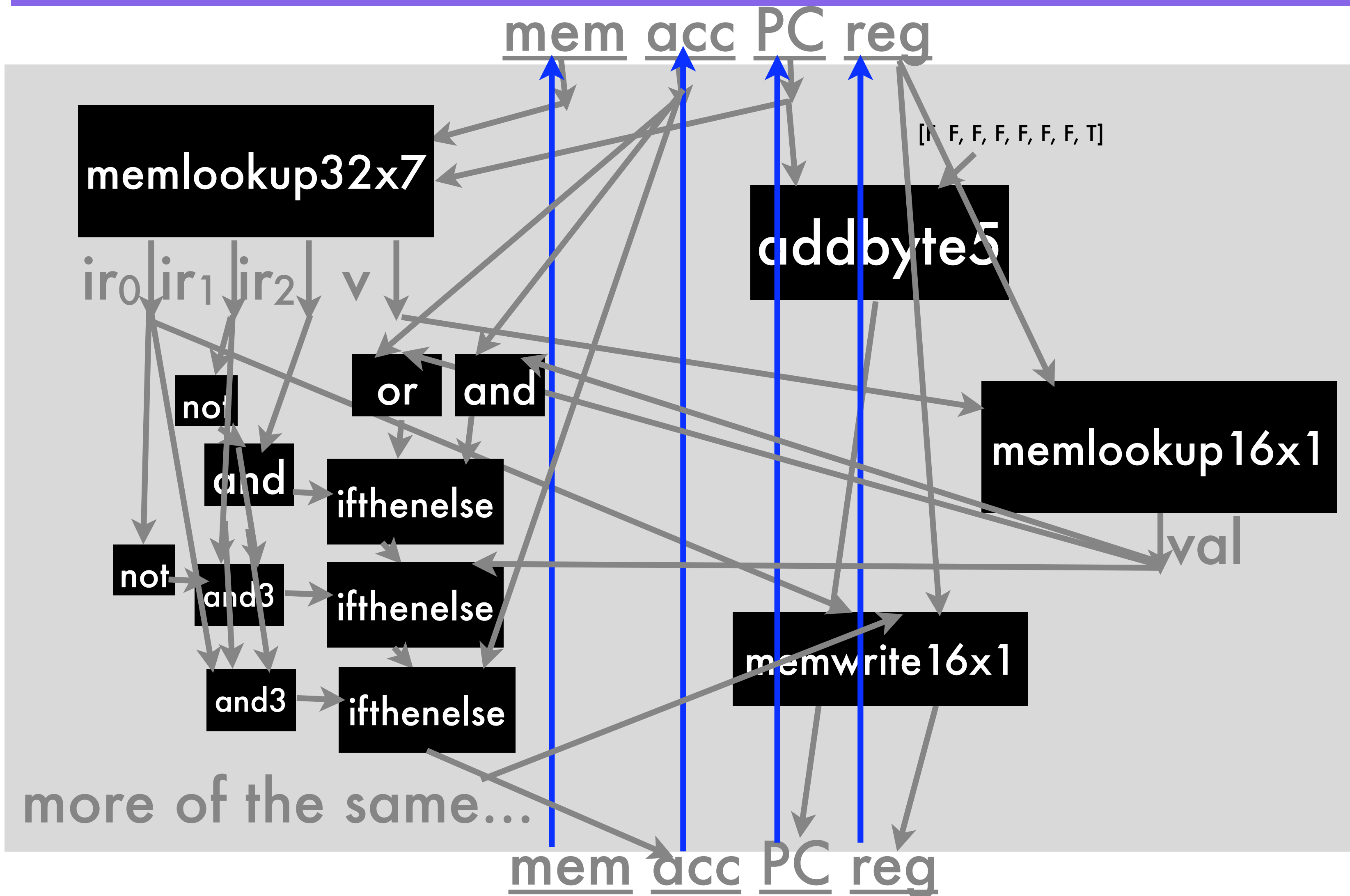
Accumulator: Special register

von Neumann Architecture

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



Cycle: A Whole Computer



Cycle: Symbolically

```
def cycle(input):
    [mem,pc,acc,reg] = input
    ir = memlookup32x7(pc,mem)
    pcnew = addbyte5(pc, intToByte5(1))
    v = ir[3:]
    val = memlookup16x1(v,reg)
    res = ifthenelse(not ir[1] and not ir[2], acc or val, False)
    res = ifthenelse(not ir[1] and ir[2], acc and val, res)
    res = ifthenelse(not ir[0] and ir[1] and not ir[2], val, res)
    res = ifthenelse(ir[0] and ir[1] and not ir[2], acc, res)
    res = ifthenelse(not ir[0] and ir[1] and ir[2], not val, res)
    res = ifthenelse(ir[0] and ir[1] and ir[2], not acc, res)
    accnew = ifthenelse(ir[0], acc, res)
    regnew = memwrite16x1(ir[0],v,reg,res)
    return [mem,pcnew,accnew,regnew]
```

- DEMO

Instruction Sets

- ML³ used a particular design that made it relatively easy to fit in a lecture slide while handling 2-bit addition.
- Computer manufacturers have different goals in mind: cost, speed, ease of running modern programs.
- Some quick examples:

x86: Intel’s Old Set

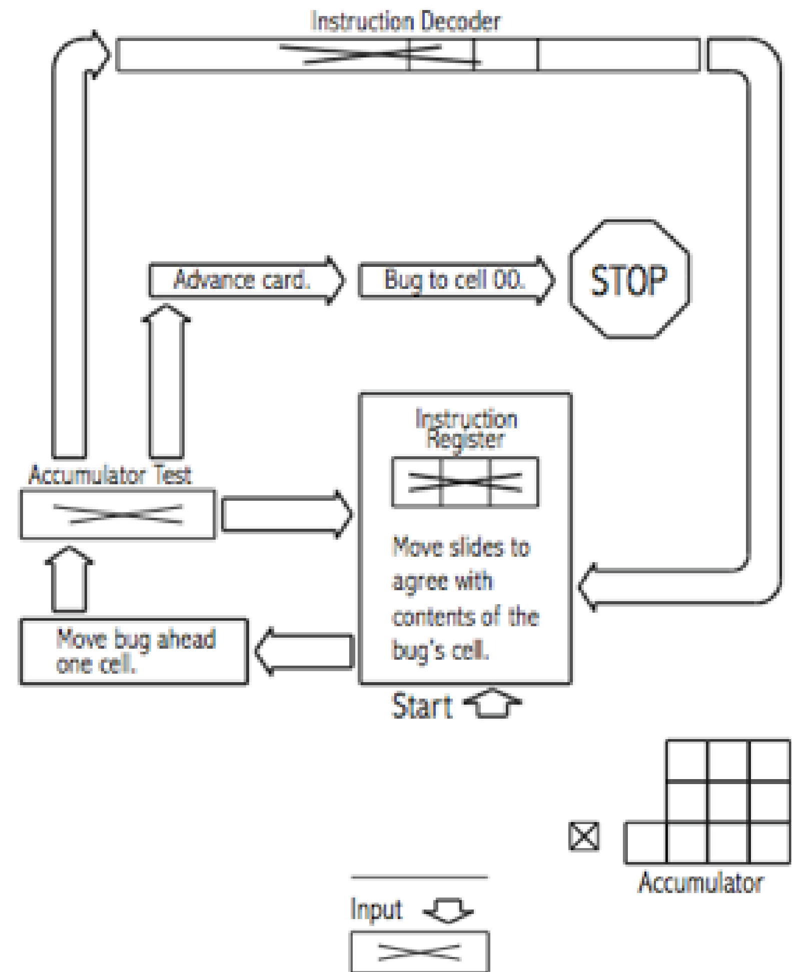
AAA: Ascii Adjust for Addition
AAD: Ascii Adjust for Division
AAM: Ascii Adjust for Multiplication
AAS: Ascii Adjust for Subtraction
ADC: Add With Carry
ADD: Arithmetic Addition
AND: Logical And
ARPL: Adjusted Requested Privilege Level of Selector (286+ PM)
BOUND: Array Index Bound Check (80188 +)
BSF: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

Z80: My First

ADC A,(HL)
ADC A,(IX+N)
ADC A,(IY+N)
ADC A,r
ADC A,HX
ADC A,HY
ADC A,LX
ADC A,LY
ADC A,N
ADC HL,BC
ADC HL,DE
ADC HL,HL
ADC HL,SP
ADD A,(HL)
ADD A,(IX+N)
ADD A,(IY+N)
ADD A,r
ADD A,HX
ADD A,HY
ADD A,LX
ADD A,LY
ADD A,N
ADD HL,BC
ADD HL,DE
ADD HL,HL
ADD HL,SP
ADD IX,BC
ADD IX,DE
ADD IX,IX
ADD IX,SP
ADD IY,BC
ADD IY,DE
ADD IY,IY
ADD IY,SP
AND (HL)
AND (IX+N)
AND (IY+N)
AND r
AND HX
AND HY
AND LX
AND LY
AND N
BIT b,(HL)
BIT b,(IX+N)
BIT b,(IY+N)
BIT b,r
CALL C,NN
CALL M,NN
CALL NC,NN
CALL NC,NN
CALL NN
CALL NZ,NN
CALL P,NN
CALL PE,NN
CALL PO,NN
CALL Z,NN
CCF
CP (HL)
CP (IX+N)
CP (IY+N)
CP r
CP HX
CP HY
CP LX
CP LY
CP N
CPD
CPDR
CPI
CPIR
CPL
DAA
DEC (HL)
DEC (IX+N)
DEC (IY+N)
DEC A
DEC B
DEC BC
DEC C
DEC D
DEC DE
DEC E
DEC H
DEC HL
DEC IX
DEC IY
DEC L
DEC SP
DI
DJNZ $N+2
EI
EX (SP),HL
EX (SP),IX
EX (SP),IY
EX AF,AF'
EX DE,HL
EXX
HALT
IM 0
IM 1
IM 2
IN A,(C)
IN A,(N)
IN B,(C)
IN C,(C)
IN D,(C)
IN E,(C)
IN H,(C)
IN L,(C)
IN (C)
INC (HL)
INC (IX+N)
INC (IY+N)
INC A
INC B
INC BC
INC C
INC D
INC DE
INC E
INC H
INC HL
INC HX
INC HY
INC IX
INC IY
INC L
INC LX
INC LY
INC SP
IND
INDR
INI
INIR
JP $NN
JP (HL)
JP (IX)
JP (IY)
JP C,$NN
JP M,$NN
JP NC,$NN
JP NZ,$NN
JP P,$NN
JP PE,$NN
JP PO,$NN
JP Z,$NN
JR $N+2
JR C,$N+2
JR NC,$N+2
JR NZ,$N+2
JR Z,$N+2
LD (BC),A
LD (DE),A
LD (HL),r
LD (HL),N
LD (IX+N),r
LD (IX+N),N
LD (IY+N),r
LD (IY+N),N
LD (NN),A
LD (NN),BC
LD (NN),DE
LD (NN),HL
LD (NN),IX
LD (NN),IY
LD (NN),SP
LD A,(BC)
LD A,(DE)
LD A,(HL)
LD A,(IX+N)
LD A,(IY+N)
LD A,(NN)
LD A,r
LD A,HX
LD A,HY
LD A,LX
LD A,LY
LD A,I
LD A,N
LD A,R
LD B,(HL)
LD B,(IX+N)
LD B,(IY+N)
LD B,HX
LD B,HY
LD B,LX
LD B,LY
LD B,r
LD B,N
LD BC,(NN)
LD BC,NN
LD C,(HL)
LD C,(IX+N)
LD C,(IY+N)
LD C,HX
LD C,HY
LD C,LX
LD C,LY
LD C,r
LD C,N
LD D,(HL)
LD D,(IX+N)
LD D,(IY+N)
LD D,HX
LD D,HY
LD D,LX
LD D,LY
LD D,r
LD D,N
LD DE,(NN)
LD DE,NN
LD E,(HL)
LD E,(IX+N)
LD E,(IY+N)
LD E,HX
LD E,HY
LD E,LX
LD E,LY
LD E,r
LD E,N
LD H,(HL)
LD H,(IX+N)
LD H,(IY+N)
LD H,r
LD H,N
LD HL,(NN)
LD HL,NN
LD HX,r*
LD HX,N
LD HY,r*
LD HY,N
LD I,A
LD IX,(NN)
LD IX,NN
LD IY,(NN)
LD IY,NN
LD L,(HL)
LD L,(IX+N)
LD L,(IY+N)
LD L,r
LD L,N
LD LX,r*
LD LX,N
LD LY,r*
LD LY,N
LD R,A
LD SP,(NN)
LD SP,HL
LD SP,IX
LD SP,IY
LD SP,NN
LDD
LDDR
LDI
LDIR
NEG
NOP
OR (HL)
OR (IX+N)
OR (IY+N)
OR r
OR HX
OR HY
OR LX
OR LY
OR N
OTDR
OTIR
OUT (C),A
OUT (C),B
OUT (C),C
OUT (C),D
OUT (C),E
OUT (C),H
OUT (C),L
OUT (C),0
OUT (N),A
OUTD
OUTI
POP AF
POP BC
POP DE
POP HL
POP IX
POP IY
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
RES b,(HL)
RES b,(IX+N)
RES b,(IY+N)
RES b,r
RET
RET C
RET M
RET NC
RET NZ
RET P
RET PE
RET PO
RET Z
RETI
RETN
RL (HL)
RL r
RL (IX+N)
RL (IY+N)
RLA
RLC (HL)
RLC (IX+N)
RLC (IY+N)
RLC r
RLCA
RLD
RR (HL)
RR r
RR (IX+N)
RR (IY+N)
RRA
RRC (HL)
RRC (IX+N)
RRC (IY+N)
RRC r
RRCA
RRD
RST 0
RST 8H
RST 10H
RST 18H
RST 20H
RST 28H
RST 30H
RST 38H
SBC A,(HL)
SBC A,(IX+N)
SBC A,(IY+N)
SBC A,r
SBC HX
SBC HY
SBC LX
SBC LY
SBC A,N
SBC HL,BC
SBC HL,DE
SBC HL,HL
SBC HL,SP
SCF
SET b,(HL)
SET b,(IX+N)
SET b,(IY+N)
SET b,r
SLA (HL)
SLA (IX+N)
SLA (IY+N)
SLA r
SLL (HL)
SLL (IX+N)
SLL (IY+N)
SLL r
SRA (HL)
SRA (IX+N)
SRA (IY+N)
SRA r
SRL (HL)
SRL (IX+N)
SRL (IY+N)
SRL r
SUB (HL)
SUB (IX+N)
SUB (IY+N)
SUB r
SUB HX
SUB HY
SUB LX
SUB LY
SUB N
XOR (HL)
XOR (IX+N)
XOR (IY+N)
XOR r
XOR HX
XOR HY
XOR LX
XOR LY
XOR N

CARDIAC (1968)

Opcode	Mnemonic	Description
0	INP	Input – take a number from the input card and put it in a specified memory cell.
1	CLA	Clear and add – clear the accumulator and add the contents of a memory cell to the accumulator.
2	ADD	Add - add the contents of a memory cell to the accumulator.
3	TAC	Test accumulator contents – performs a sign test on the contents of the accumulator.
4	SFT	Shift – shifts the accumulator x places left, then y places right.
5	OUT	Output – take a number from the specified memory cell and write it on an output card.
6	STO	Store – copy the contents of the accumulator into a specified memory cell.
7	SUB	Subtract – subtract the contents of a specified memory cell from the accumulator.
8	JMP	Jump - jump to a specified memory cell
9	HRS	Halt and reset – stop program execution, move bug to cell 00.



CARDboard Illustrative Aid to Computation

Next Time

- Subroutines, recursion.
- Hillis Chapter 3, first two sections.