EE282
Computer Architecture

Lecture 2:
Instruction-Set Architecture: Part I

Oct 2, 2001

Marc Tremblay
Stanford University
marctrem@csl.stanford.edu

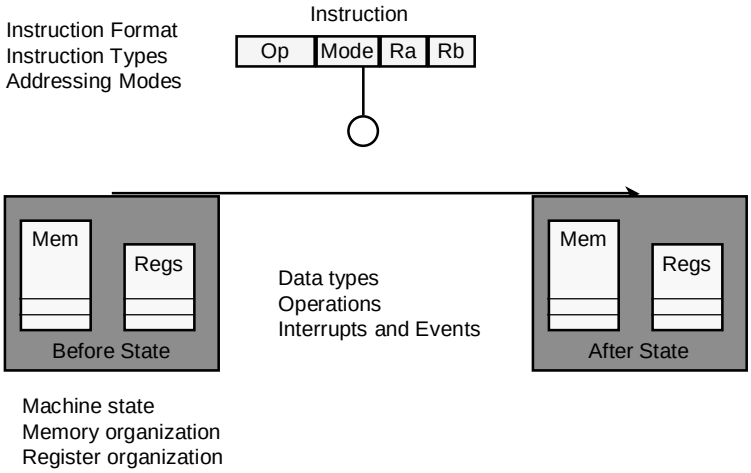
Today

- Instruction-set architecture
  - Architecture vs Implementation
  - Machine state
  - Opcodes and Operands
  - Register Organization

Instruction Set Architecture

- The ISA is an agreement (contract) between the programmer and the hardware.
  - Defines the visible state of the system
  - Defines how the state changes in response to instructions

Instruction-Set Architecture (ISA) at a glance



Why an ISA Contract?

- Programmers use the ISA to model how programs will execute.
- Hardware implementers use the ISA as a formal definition of the correct way to execute programs.
- Traditionally the ISA is defined in terms of the binary encoding of the instruction set

Who is impacted by the details of the ISA?

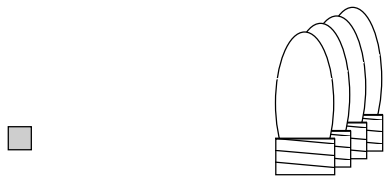
How does the ISA impact hardware?
How does the ISA impact software?

Architecture vs. Implementation

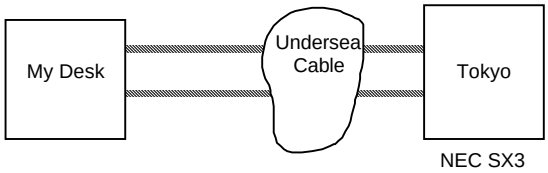
- Architecture defines what a computer system does in response to a program and a set of data.
  - programmer visible elements of the computer system.
- Implementation defines how a computer does it.
  - Sequence of steps to complete operations
  - Time it takes to execute operations
  - Hidden "bookkeeping" functions

Example:
Variations of an identical architecture

Same Final Answer



.18 µ CMOS **Vacuum Tubes**



Simulator 5000 Miles away

Why separate architecture from implementation?

- Compatibility
  - VAX architecture - 1 chip to mainframe
  - ARM - 20x performance range
- Longevity
  - X86 in 7th generation

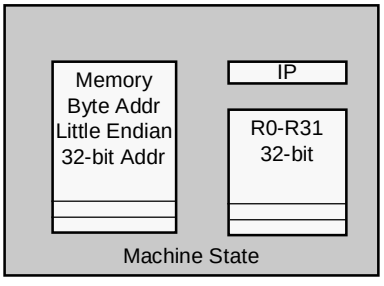
  - retain software investment
  - maintain homogeneous environment
  - amortize costs over multiple markets

Architecture Families

- Many architectures 'grow' with time
- Manufactures produce families of chips that execute the same programs.
- 8088, 8086, 80286, 80386, 80486, Pentium, PII, PIII, P4
- Chips have different ISA's - but all support a core set of user instructions -this is the 'family architecture'.

Machine State

- Registers
  - size and type
    - at least an instruction pointer (IP)
    - accumulators
    - index registers
    - general registers
    - control registers
- Memory
  - visible hierarchy (if any)
  - addressability
    - word, byte, bit
    - little vs big endian (aliasing)
    - maximum size
  - protection and relocation



Component of Instructions

- **A set of possible operations (opcode)**
- **For each operation:**
  - **Number of operands**
  - **Operand Specifiers**
- **In practice - there will be restrictions on operand specifiers.**
- **An encoding of the instructions is required.**
- **Typically instructions are grouped into classes with similar formats.**

Operand Number

• **No operands**	`HALT`	
	`NOP`	
• **1 operand**	`NOT AX`	`AX <- ~AX`
	`JMP LABEL1`	
• **2 operand**	`ADD R1,R2`	`R1 <- R1+R2`
	`LDI R3,#1234`	
• **3 operand**	`ADD R3,R1,R2`	`R3 <- R1+R2`
• **>3 operand**	`MADD R4,R3,R2,R1`	`R4 <- R3+R2*R1`

Effect of Operand Number

```
E = (C+D) * (C-D);
```

Assign

```
C -> r1
D -> r2
E -> r3
```

3 operand machine

```
add    r3,r1,r2
sub    r4,r1,r2
mult   r3,r4,r3
```

2 operand machine

```
mov    r3,r1
add    r3,r2
sub    r2,r1
mult   r3,r2
```


Operand Types

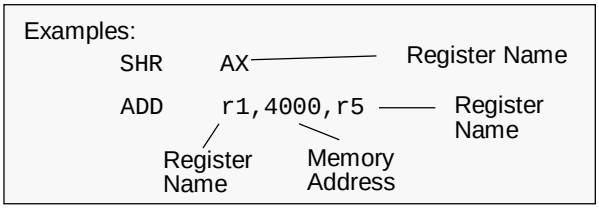
How do we specify an operand?

1) **We don't. - Operands can be implicit.**
   Example: RET (Top of stack is implicit (via SP))

2) **We include operand specifiers.**
   a) **Name (usually explicit)**
   b) **Name Space (usually implicit)**

Name Spaces

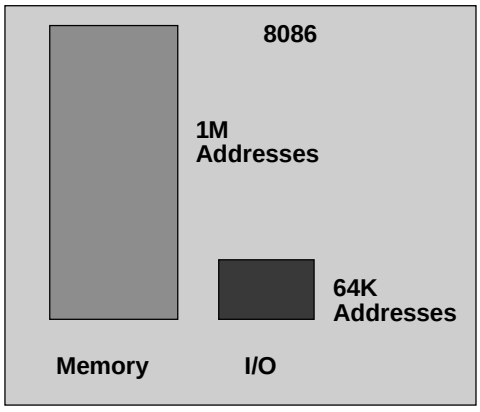
- Each name space contains a separately enumerable set of names.
- For a typical GPR machine, this includes:
  - Register Numbers
  - Memory Addresses
- Name space is usually implied for each operand by the opcode.

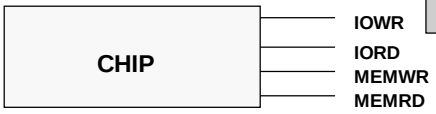


- Some ISA (VAX) include name space descriptor in the operand specifier.

Alternate Name Spaces

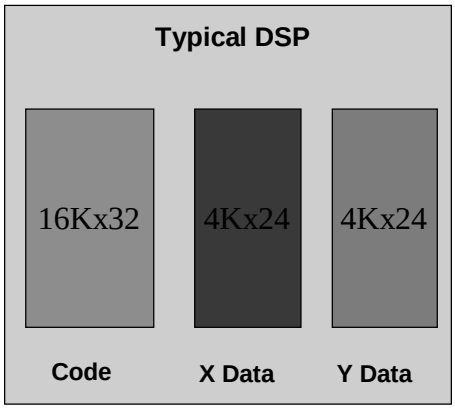
- **Other name spaces often exist as well.**
  - **Condition Codes**
  - **Immediate Data**
  - **I/O space**



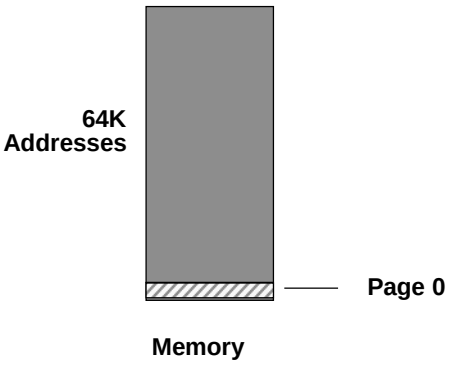


Multiple Name Spaces

- **Architectures can define an arbitrary number of name spaces**
  - **Instructions must specify or imply the name space for each operand.**
  - **Hardware must be able to address each name space properly**



Name space overlap (aliasing)



- **On M6800 - Page 0 is a 256 byte separate address space.**
  Regular address space can also address page 0

Describing Operands

- 2 choices
- Universal Operand Specifiers
  - Include space and name
- Name space specific
  - Space is implied by operation
  - Name is specified in Operand Field.

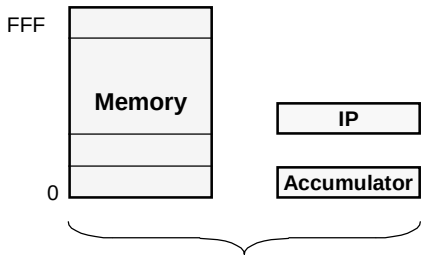
OP	Name Space	Name
RET	Implicit	
INC	Register	5
DEC	Memory	0x2000

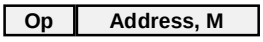
OR

OP	Name
RET	
INCR	5
DECM	0x2000

Evolution of Register Organization

- In the beginning, there was the *accumulator*
  - two 'working' instruction types: op and store
    - A ← A op M
    - A ← A op *M
    - *M ← A
  - a *one* address architecture
    - each instruction encodes a single memory address
  - two addressing modes
    - *immediate*, M
    - *direct* addressing, *M
  - Typical of early machines
    - EDSAC, EDVAC, …





Instruction Format

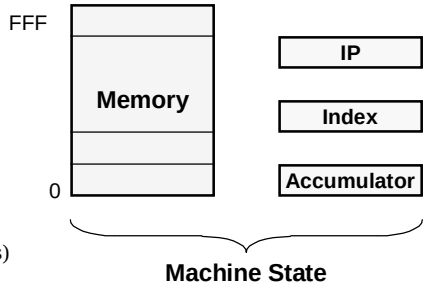
(Op indicates addressing mode)

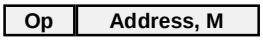
Why Accumulator Architectures?

- Registers (flip-flops) were very expensive in early implementation technologies.
- Instruction decoding was simple
  - Logic was expensive as well
  - Critical programs were small - extra instructions no problem
- Minimal cycle time
  - Hardwired programs were fast - logic was slow
- Model matched earlier 'tabulating' machines
  - Think "adding machine"

The Index Register

- Add an *indexed* addressing mode
  - A ← A op *(M + I)
  - *(M + I) ← A
  - facilitates array accesses
    - x[j]
      - address of x[0] in instruction
      - j in index register, I
  - one register for each key function
    - IP points to instructions (addresses)
    - I points to data (addresses)
    - A holds data values
  - Logic optimized for each type
    - eg. Counters used for IP, I
  - *Indirect* addressing is a special case
    - A ← A op *(I)





Instruction Format

Example of Indexed Addressing

```
sum = 0 ;
for(i=0;i<n;i++)
   sum = sum + y[i] ;
```

```
START:  CLRA
        CLRX
LOOP:   ADDA    y(X)
        INCX
        CMPX    n
        BNE     LOOP
```

With index register

```
START:  CLR     i
        CLR     sum
LOOP:   LOAD    IX
        AND     #MASK
        OR      i
        STORE   IX
        LOAD    sum
IX:     ADD     y
        STORE   sum
        LOAD    i
        ADD     #1
        STORE   i
        CMP     n
        BNE     LOOP
```

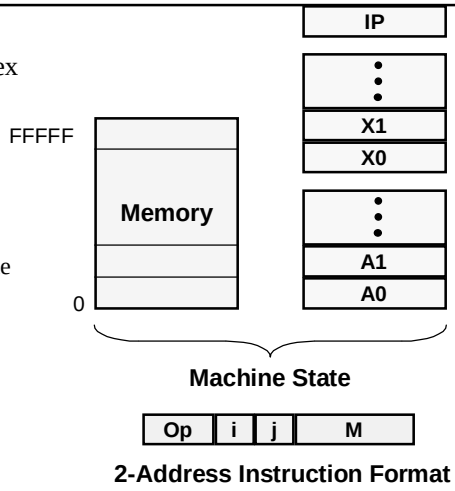
Without index register

But what about

```
sum = 0 ;
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
    sum = sum + x[j]*y[i] ;
```

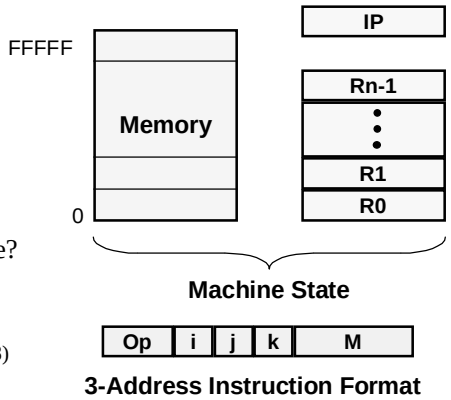
More Registers give Denser Code

- Keep several variables in accumulators, pointers in index registers
  - fewer loads and stores
    - $A_i \leftarrow A_i$ op A_j
    - $A_i \leftarrow A_i$ op *$(M+X_j)$
  - includes a *register* mode
  - called a *two-address* machine because each instruction contains two *names*.
  - A *three-address* machine allows the destination to be separately specified
    - $A_i \leftarrow A_j$ op A_k
  - but takes more bits



General Registers

- Eliminate distinction between *accumulators* and *index regs (i.e. data reg vs address reg)*
- Use any register as a variable or pointer
  - simpler
  - more orthogonal
  - better utilization of fast storage
  - but - assumes addresses & data are similar sizes.
- How many registers should there be?
  - More - fewer loads/stores
  - but - more bits of instruction
    - PDP-11 (16), VAX (32), EPIC (128)



Specialized Registers

- Some architectures use specialized data or address registers in addition to or instead of general-purpose registers.

SP	Dedicated stack pointer
LOOP	Dedicated loop counter
Remainder	Extended divide precision

- Specialized registers complicate the architecture model but often simplify implementation

Load/Store Machines

- Only *load* and *store* instructions reference memory
- All arithmetic is performed on registers
  - requires more instructions
    - sparser code
  - uses more visible registers
  - same number of operations
  - easier for compiler
    - enables re-use of results, variables loaded
  - easier to implement
    - simpler pipeline
    - limits certain exceptions

```
LOAD    R1 <- A
ADD     R2 <- R1 + B
ADD     C  <- R2 + 3
```

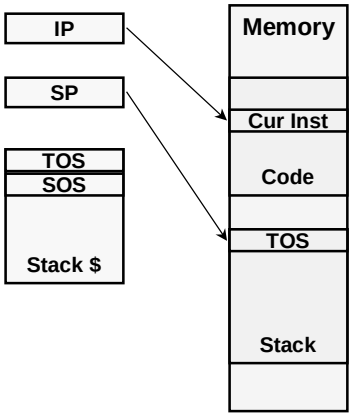
Memory Operands Allowed

```
LOAD    R1 <- A
LOAD    R2 <- B
ADD     R3 <- R1 + R2
ADD     R4 <- R3 + 3
STORE   C  <- R4
```

Strict Load/Store

Stack Machines

- Register state is IP and SP
- All instructions performed on TOS (top-of-stack) and SOS
  - pushes and pops implied
    - op TOS SOS
    - op TOS M
    - op TOS *M
    - op TOS *(M+SP)
- Many instructions are *zero* address
- Stack cache is required for performance



Example of Stack Code

```
a = b + c * d ;
e = a + f[j] + c ;
```

```
PUSH    d
MUL     c
ADD     b
PUSH    j
PUSHX   f
ADD     c
ADD
POP     e
```

One Address Stack

8inst, 7addr

```
PUSH    d
PUSH    c
MUL
PUSH    b
ADD
PUSH    j
PUSHX   f
PUSH    c
ADD
ADD
POP     e
```

Pure Stack (zero address)

11inst, 7addr

```
LOAD    R1, d
LOAD    R2, c
MUL     R3,R1,R2
LOAD    R4, b
ADD     R5,R4,R3
LOAD    R6, j
LOAD    R7, f(R6)
ADD     R8,R7,R2
ADD     R9,R5,R8
STORE   e, R10
```

Load/Store (many GP registers)

10inst, 6addr

Instruction Semantics

- The ISA definition includes
  - system state
  - the effect of each operation on system state
- and actually one more item
  - the relationship between instructions

- Most traditional ISA definitions specify sequential execution
  - Each instruction completes all state changes in program order
- Some modern ISAs define relationships between instructions

Non-sequential semantics

```
LBL: add  R3,R1,R3
     bne  R0,R4,LBL
     subi R4,R4,#1
```

MIPS - delayed branch

EPIC - concurrent execution

Next Time

- Instruction-set architecture
  - Addressing Modes
  - Data Types
  - Common instruction types
  - ISA Design Styles
  - Binary Encodings
  - Interrupts & Events