Lecture 5: Instruction Set Architectures II

Announcements

- Turn in Homework #1
- XSPIM tutorials in PAI 5.38 during TA office hours
  - Tue Feb 2: 2-3:30pm
  - Wed Feb 3: 1:30-3pm
  - Thu Feb 4: 3-4:30pm
- Take QUIZ 2 before 11:59pm today over Chapter 1
- Quiz 1: 100% - 29; 80% - 25; 60% - 17; 40% - 3

Lecture 5: Instruction Set Architectures II

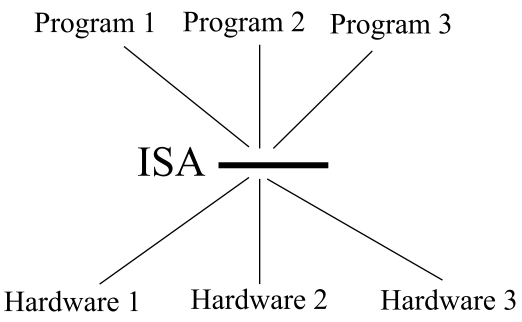
Last Time

- Performance analysis
- ISA basics

Today

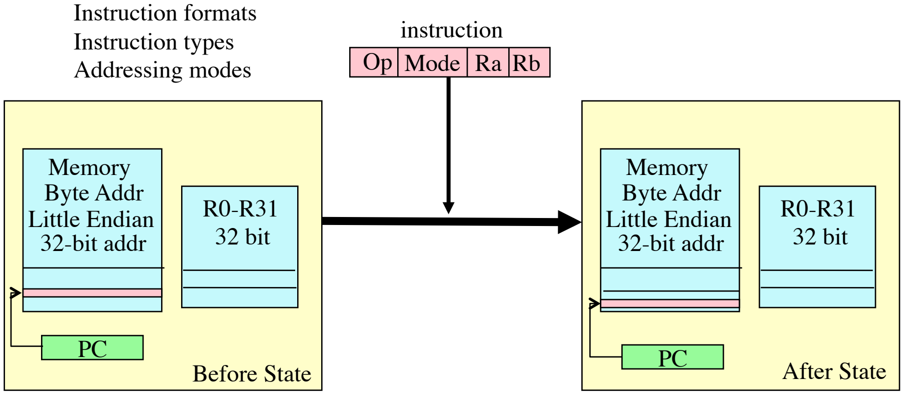
- ISA II
  - Machine state (memory, computation, control)
  - Design principles
  - Instruction formats
  - Representing and addressing data

ISA is an interface (abstraction layer)



ISA Specifies How the Computer Changes State

Instruction formats
Instruction types
Addressing modes



Machine state includes
PC, memory state register state

Design Considerations:

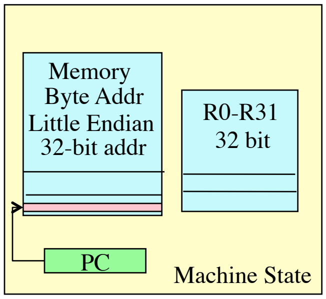
Changes to Machine State Imply Instruction Classes & Design

Changes to Memory State

- **Data representation**
- **ALU Operations**
  - arithmetic `(add, sub, mult, div)`
  - logical `(and, or, xor, srl, sra)`
  - data type conversions `(cvtf2d,cvtf2i)`
- **Data movement**
  - memory reference `(lb, lw, sb, sw)`
  - register to register `(movi2fp, movf)`

Control: what instruction to do next

- PC = ??
- tests/compare `(slt, seq)`
- branches and jumps `(beq, bne, j, jr)`
- procedure calls `(jal, jalr)`
- operating system entry `(trap)`

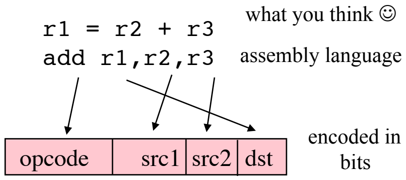


ISA Design Principles

1. Simplicity favors regularity
   - e.g., instruction size, instruction formats, data formats
   - eases implementation by simplifying hardware
2. Smaller is faster
   - fewer bits to read, move, & write
   - use/reuse the register file instead of memory
3. Make the common case fast
   - e.g., small constants are common, thus immediate fields can be small
4. Good design demands compromise
   - special formats for important exceptions
   - e.g., a jump far away (beyond a small constant)

ISA Design
Components of Instructions

- Operations (opcodes)
  - ALU, Data, Control
- Number of operands
- Operand specifiers
- Instruction encodings



Operand Number
Affects All Instruction Classes

• No Operands	HALT	NOP	
• 1 operand	NOT R4	R4 ⇐ R4	JMP _L1
• 2 operands	ADD R1, R2	R1 ⇐ R1 + R2	LDI R3, #12
• 3 operands	ADD R1, R2, R3	R1 ⇐ R2 + R3	
• > 3 operands	MADD R4,R1,R2,R3	R4 ⇐ R1+(R2*R3)	

Effect of Operand Number

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

Assign

```
C ⇒ r1
D ⇒ r2
E ⇒ r3
```

3 operand machine

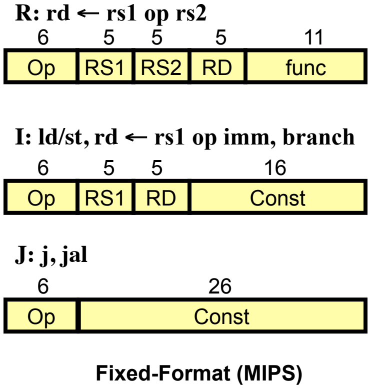
```
add  r3,r1,r2
sub  r4,r1,r2
mult r3,r4,r3
```

2 operand machine

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

Instruction Formats: Can not be perfectly uniform

- ALU, control, and data instructions need to specify different information
  - return
  - increment R1
  - R3 ← R1 + R2
  - jump to 64-bit address
- Frequency varies
  - instructions
  - constants
  - registers
- Encoding choices
  - fixed format
  - small number of formats
  - byte/bit variable



R: rd ← rs1 op rs2

6	5	5	5	11
Op	RS1	RS2	RD	func

I: ld/st, rd ← rs1 op imm, branch

6	5	5	16
Op	RS1	RD	Const

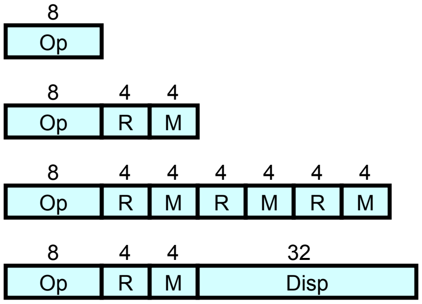
J: j, jal

6	26
Op	Const

Fixed-Format (MIPS)

Fixed or Variable-Length Instructions?

- Variable-length instructions give more efficient encodings
  - no bits to represent unused fields/operands
  - can frequency code operations, operands, and addressing modes
  - Examples
    - VAX-11, Intel x86 (byte variable)
    - Intel 432 (bit variable)
- But - can make fast implementation difficult
  - sequential determination of location of each operand



8
Op

8	4	4
Op	R	M

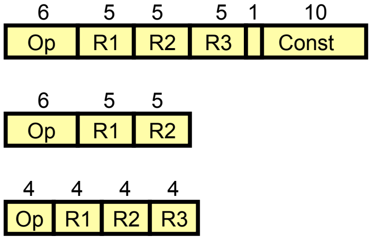
8	4	4	4	4	4	4
Op	R	M	R	M	R	M

8	4	4	32
Op	R	M	Disp

VAX instrs: 1-53 bytes!

Compromise: A Few Formats and A Few Sizes

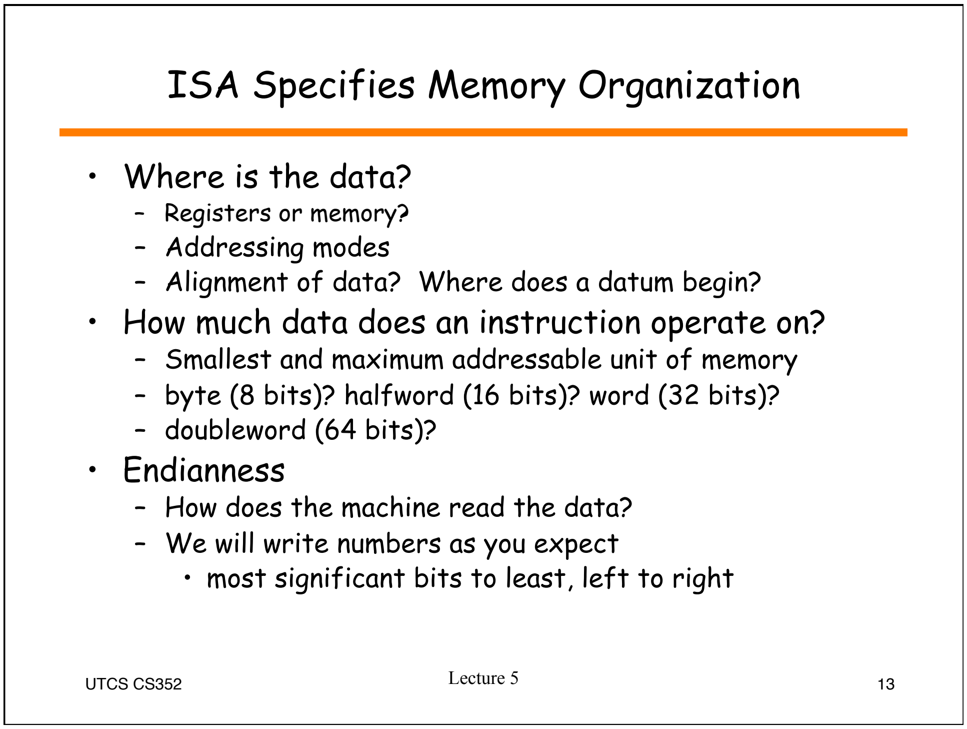
- Gives much better code density than fixed-format
  - important for embedded processors
- Simple to decode
- Examples:
  - ARM Thumb, MIPS 16
- Another approach
  - On-the fly instruction decompression (IBM CodePack)



6	5	5	5	1	10
Op	R1	R2	R3		Const

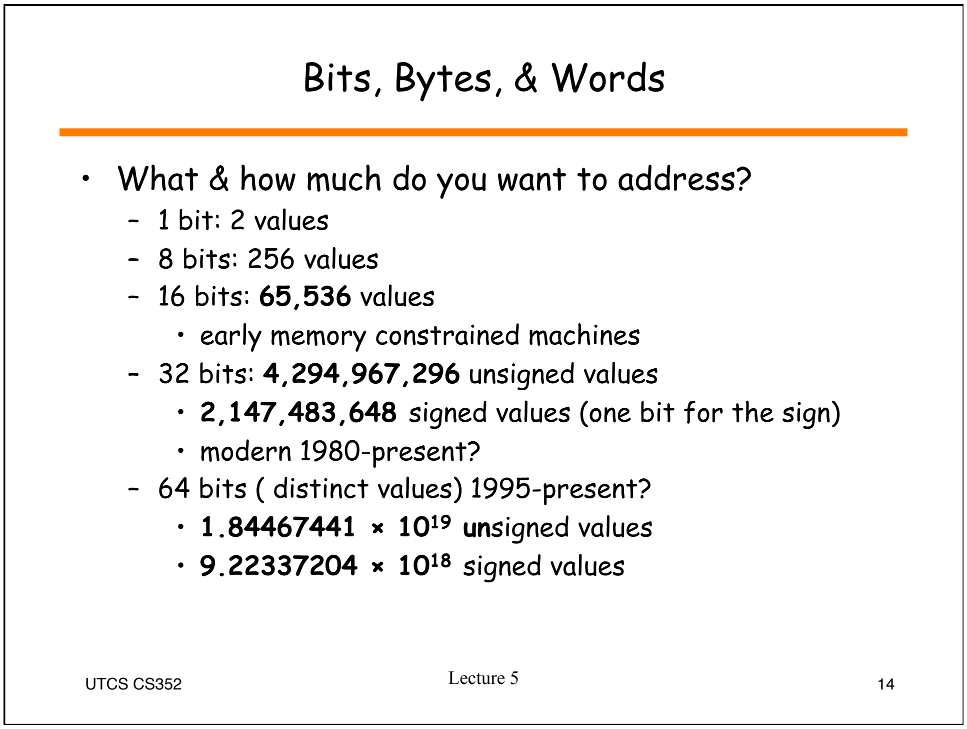
6	5	5
Op	R1	R2

4	4	4	4
Op	R1	R2	R3



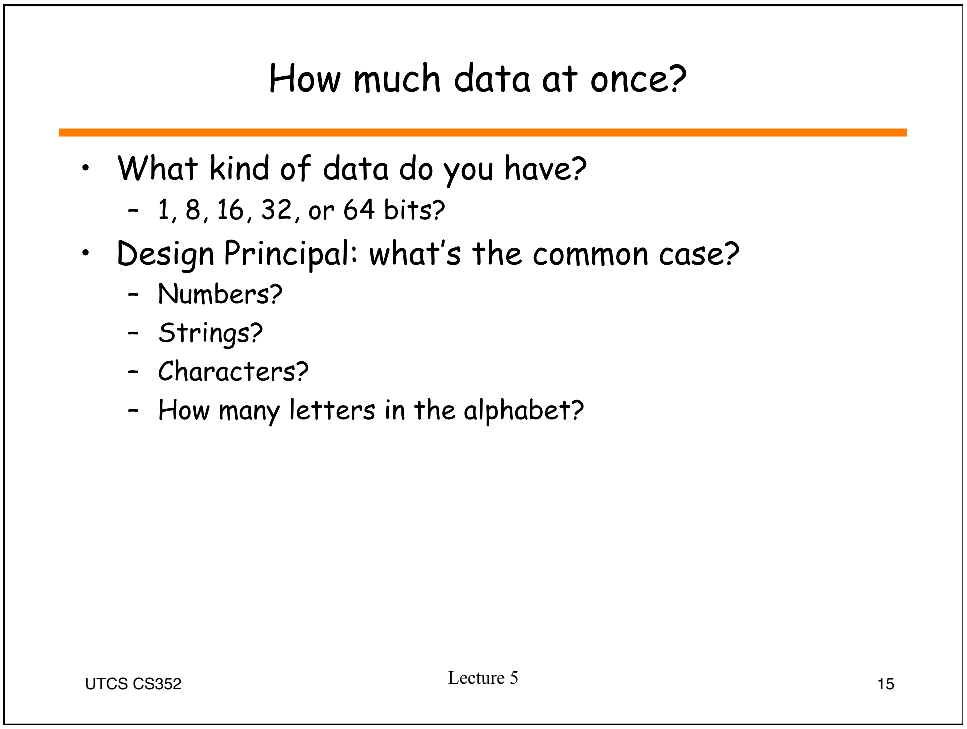
ISA Specifies Memory Organization

- Where is the data?
  - Registers or memory?
  - Addressing modes
  - Alignment of data? Where does a datum begin?
- How much data does an instruction operate on?
  - Smallest and maximum addressable unit of memory
  - byte (8 bits)? halfword (16 bits)? word (32 bits)?
  - doubleword (64 bits)?
- Endianness
  - How does the machine read the data?
  - We will write numbers as you expect
    - most significant bits to least, left to right



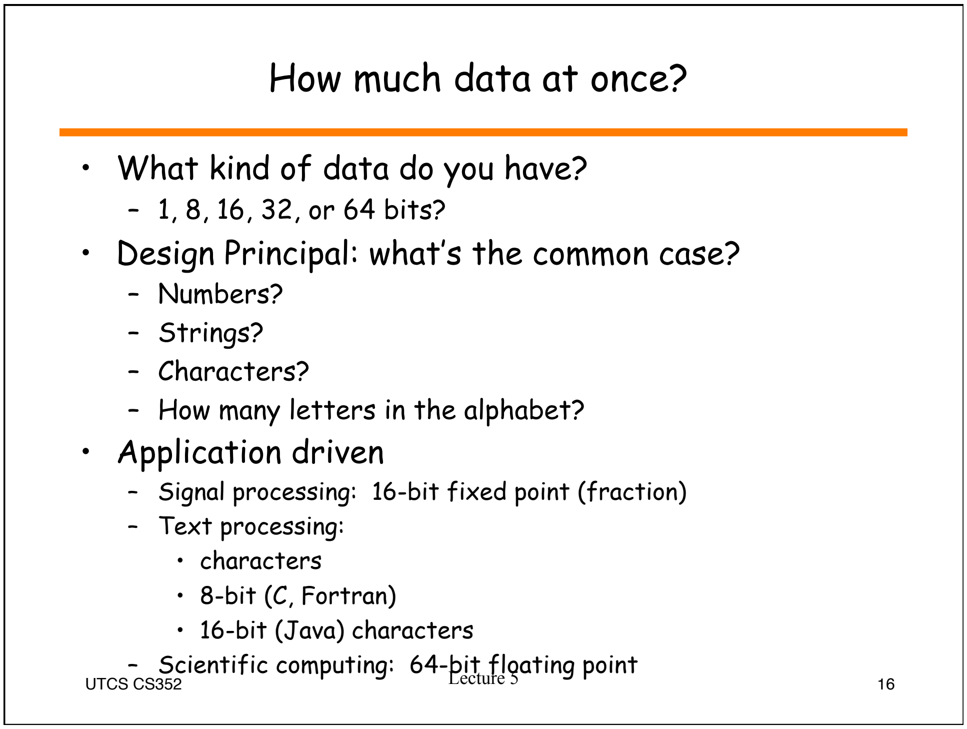
Bits, Bytes, & Words

- What & how much do you want to address?
  - 1 bit: 2 values
  - 8 bits: 256 values
  - 16 bits: **65,536** values
    - early memory constrained machines
  - 32 bits: **4,294,967,296** unsigned values
    - **2,147,483,648** signed values (one bit for the sign)
    - modern 1980-present?
  - 64 bits (distinct values) 1995-present?
    - **$1.84467441 \times 10^{19}$ unsigned** values
    - **$9.22337204 \times 10^{18}$** signed values



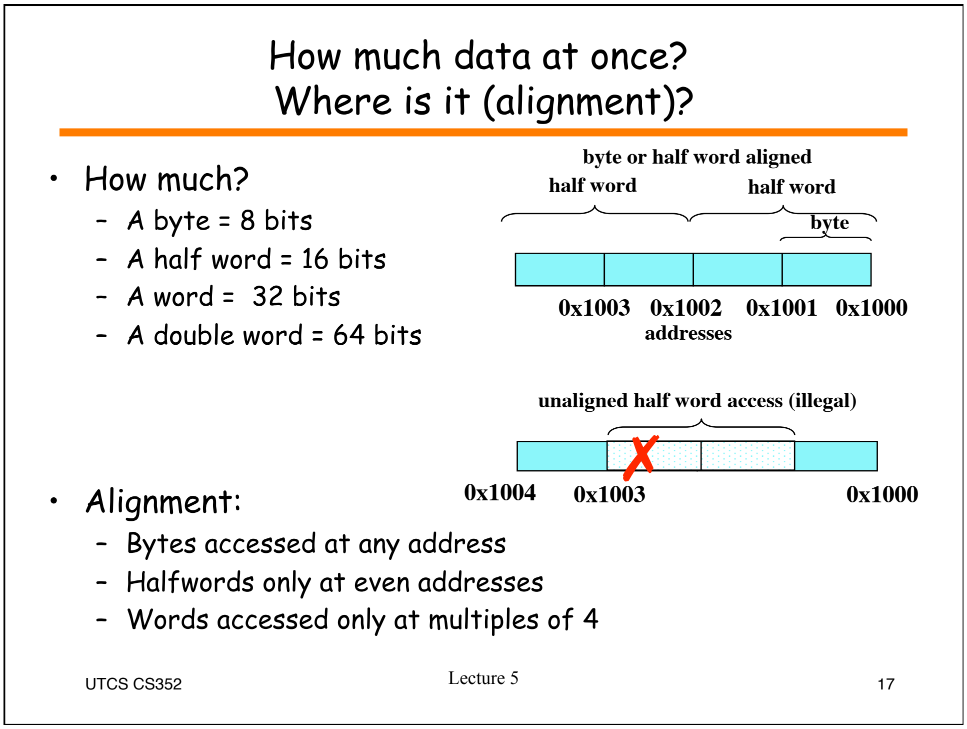
How much data at once?

- What kind of data do you have?
  - 1, 8, 16, 32, or 64 bits?
- Design Principal: what's the common case?
  - Numbers?
  - Strings?
  - Characters?
  - How many letters in the alphabet?



How much data at once?

- What kind of data do you have?
  - 1, 8, 16, 32, or 64 bits?
- Design Principal: what's the common case?
  - Numbers?
  - Strings?
  - Characters?
  - How many letters in the alphabet?
- Application driven
  - Signal processing: 16-bit fixed point (fraction)
  - Text processing:
    - characters
    - 8-bit (C, Fortran)
    - 16-bit (Java) characters
  - Scientific computing: 64-bit floating point

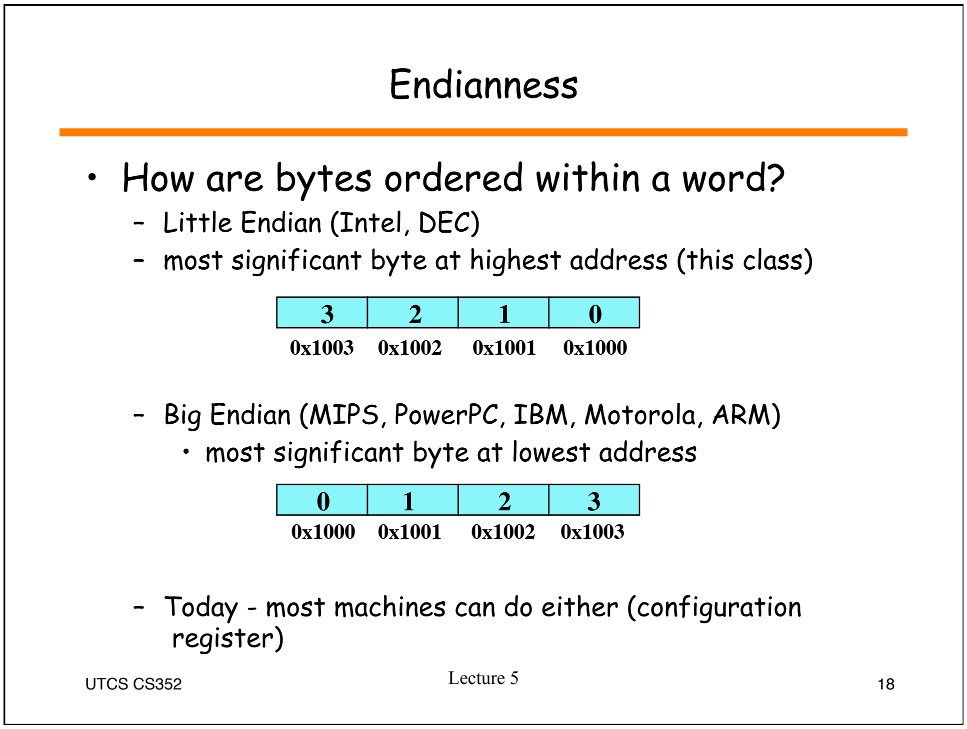


How much data at once? Where is it (alignment)?

- How much?
  - A byte = 8 bits
  - A half word = 16 bits
  - A word = 32 bits
  - A double word = 64 bits

byte or half word aligned
half word
half word
byte
0x1003 0x1002 0x1001 0x1000
addresses
unaligned half word access (illegal)
✗
0x1004 0x1003 0x1000

- Alignment:
  - Bytes accessed at any address
  - Halfwords only at even addresses
  - Words accessed only at multiples of 4



Endianness

- How are bytes ordered within a word?
  - Little Endian (Intel, DEC)
  - most significant byte at highest address (this class)

3	2	1	0
0x1003	0x1002	0x1001	0x1000

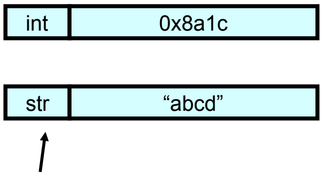
  - Big Endian (MIPS, PowerPC, IBM, Motorola, ARM)
    - most significant byte at lowest address

0	1	2	3
0x1000	0x1001	0x1002	0x1003

  - Today - most machines can do either (configuration register)

Data Types

- How does the machine interpret the contents of memory and registers?
- Explicit or implicit?
  - tag
  - use



int 0x8a1c

str "abcd"

Examples of tags (ie. Symbolics machine)

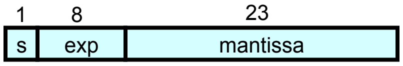
- Most *general purpose* computers correlate size with type
  - 8, 16, 32, 64-bit
  - signed and unsigned
  - fixed and floating
  - characters
  - addresses vs. values

Example: 32-bit Floating Point

- Floating Point Type specifies mapping from bits to real numbers
  - format
    - 1 bit sign
    - 8-bit exponent
    - 23-bit mantissa
  - interpretation
    - mapping from bits to abstract set

$$v = (-1)^S \times 2^{(E-127)} \times 1.M$$

  - operations
    - add, mult, sub, sqrt, div

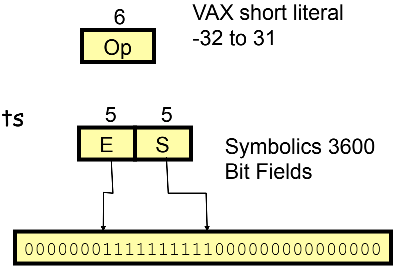


1 8 23

s exp mantissa

Integer Constants

- Integer constants
  - mostly small
  - positive or negative
- Bit fields
  - contiguous field of 1s within 32 bits (64 bits)
- Other
  - addresses, characters, symbols
- A good architecture
  - uses a few bits to encode the most common.
  - allows any constant to be generated (table reference)
  - MIPS stores 32 bits of zero in $zero



VAX short literal
-32 to 31

Symbolics 3600
Bit Fields

Strings & Characters

- Programming Language driven
- C/C++, Fortran, etc
  - ASCII
    - American Standard Code for Information Interchange
    - 1 character per byte
    - 256 charcters
    - 4 characters per word

    - Example: 'B' is 66, 'a' is 97, 't' is 116 '!' is 33
    - 'Bat!'

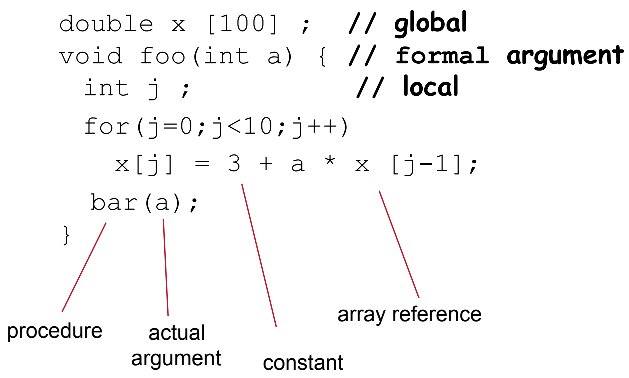
0100 0010	0110 0001	0111 0100	0010 0001

Strings & Characters

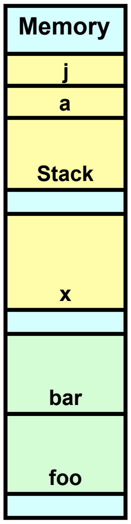
- Programming Language driven
- Real Language driven
  - Universal Character Set
  - UTC-8, UTC-16, UTC-32
  - Lots of languages, not just English!
- Java uses Unicode
  - UTC 16
  - 2 characters per word (instead of 4 in ASCII)
  - Converts between other encodings

Where is the data?
Addressing Modes

Driven by Executable Layout & Program Usage

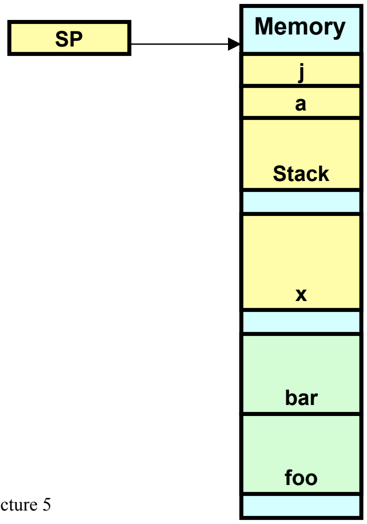


```
double x [100] ;   // global
void foo(int a) {  // formal argument
  int j ;           // local
  for(j=0;j<10;j++)
    x[j] = 3 + a * x [j-1];
  bar(a);
}
```



Addressing Modes

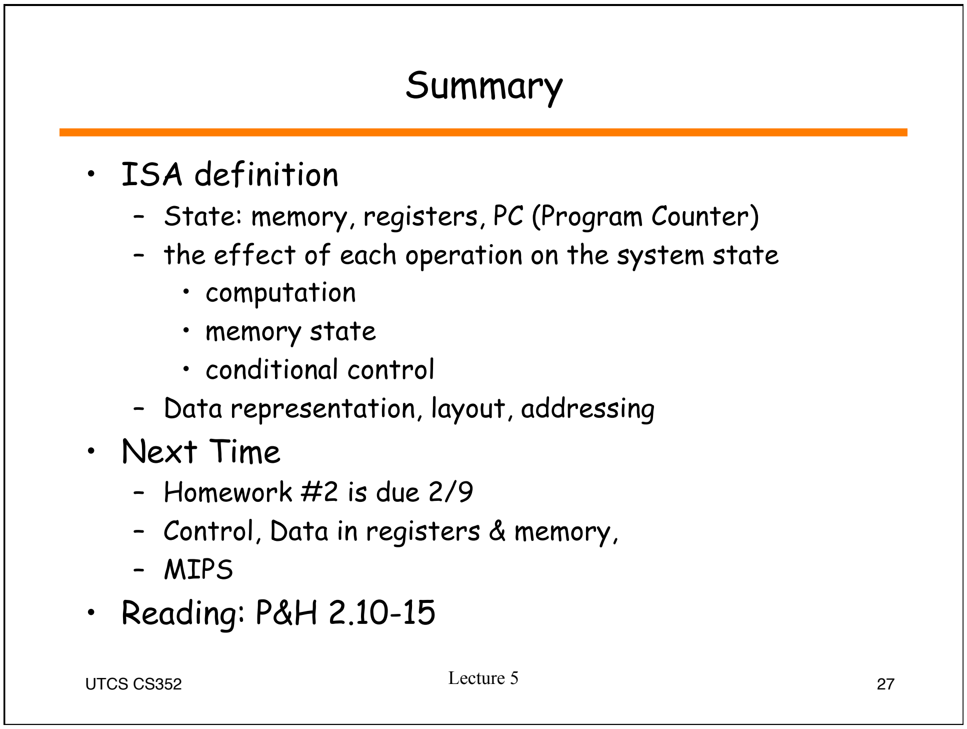
- Stack relative for locals and arguments
  a, j: *(R30+x)
- Short immediates (small constants)
  3
- Long immediates (global addressing)
  &x[0], &bar: 0x3ac1e400
- Indexed for array references
  *(R4+R3)



Addressing Modes

#n	immediate
(0x1000)	absolute
Rn	Register value
(Rn)	Register indirect (as address)
-(Rn)	predecrement
(Rn)+	postincrement
*(Rn)	Memory indirect
*(Rn)+	postincrement
d(Rn)	Displacement
d(Rn)[Rx]	Scaled

VAX 11 had 27 addressing modes (why?)



Summary

- ISA definition
  - State: memory, registers, PC (Program Counter)
  - the effect of each operation on the system state
    - computation
    - memory state
    - conditional control
  - Data representation, layout, addressing
- Next Time
  - Homework #2 is due 2/9
  - Control, Data in registers & memory,
  - MIPS
- Reading: P&H 2.10-15