CPS104 Computer Organization

Lecture 5
Data Types & Memory
Bitwise operations
Instruction Set Architecture

September 8 , 2008

Gershon Kedem

©GK Fall 2008

Administratrivia

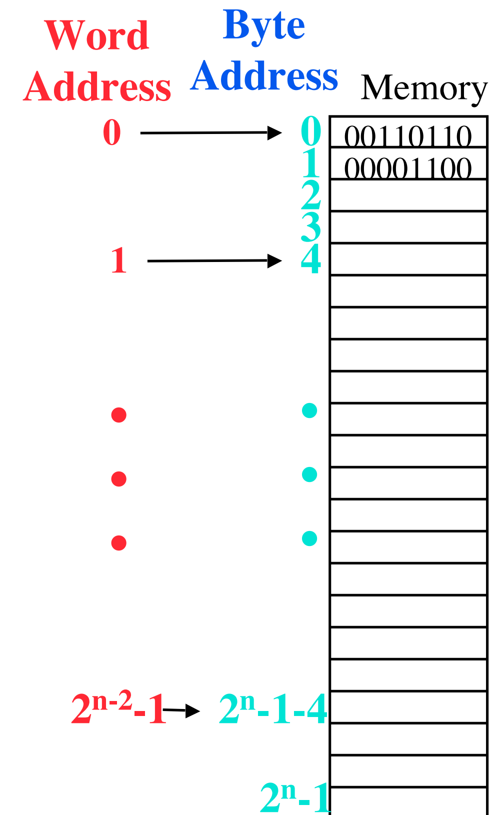
- **Homework-1 On the Web:** ***http://kedem.cs.duke.edu/cps104/Homework.html***
- **Due Today.**
- **Submit your homework *in class*, in writing.**
  - **Written homework MUST be stapled!**

- **Programming Puzzle-1 is on the Web, see:** `kedem.cs.duke.edu/cps104/ppz/PPZ1.html`
- **Due: September 10 (11:59 pm).**
- **Submit the `*.java` file to `cps104/PPZ1`**
- **Use the Eclipse system to submit the file.**
- **Help is on-line.**

©GK Fall 2008

Review: A Program's View of Memory

- **What is Memory?** a bunch of bits
- **Looks like** a large linear array
- **Find things by** indexing into array
  - Uses unsigned integer!
- Most computers support byte (8-bit) addressing
  - Each byte has a unique address (location).
  - Byte of data at address `0x100` and `0x101`
  - Word of data at address `0x100` and `0x104`
- 32-bit v.s. 64-bit addresses
  - we will assume 32-bit for rest of course, unless otherwise stated

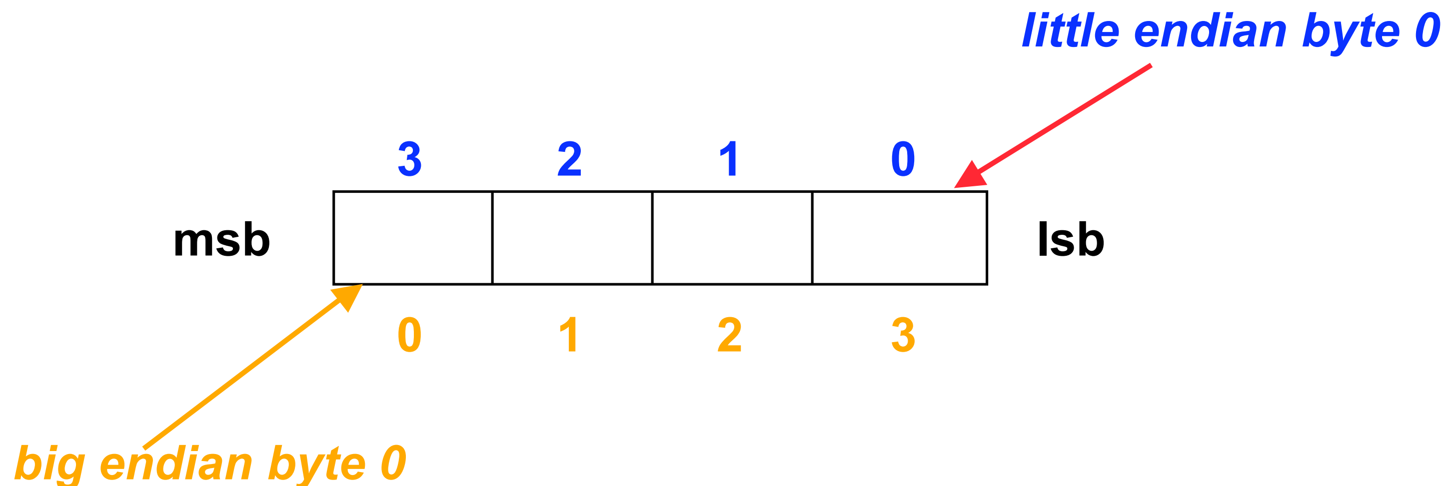


©GK Fall 2008

Review: Buzz Word Definition: Endianess

Byte Order

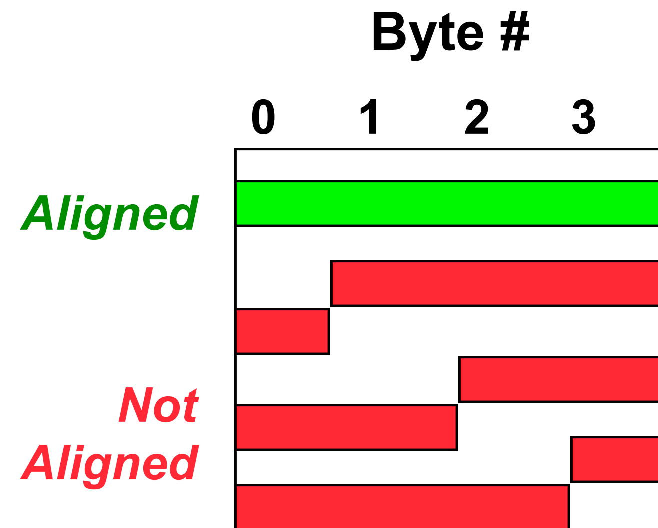
- **Big Endian:** byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** byte 0 is 8 least significant bits Intel X86, DEC Vax, DEC Alpha



©GK Fall 2008

Review: Definition: Alignment

- **Alignment:** require that objects fall on address that is multiple of their size.
- 32-bit integer
  - Aligned if address % 4 = 0
- 64-bit integer?
  - Aligned if address % 8 = 0



©GK Fall 2008

Review: Pointers and reference variables

- A pointer is a memory location (variable) that contains the address of another memory location
- C uses “address of” operator &
  and the “dereference” operator *
  - don’t confuse with bitwise AND operator (later today)

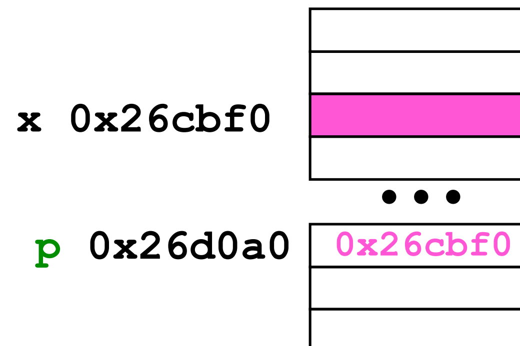
Given

```
int x; int *p;
p = &x;
```

Then

```
*p = 2;  and x = 2; produce the same result
```

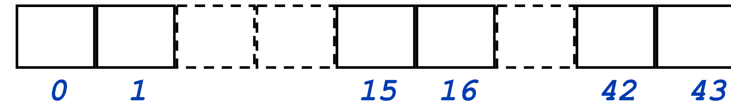
On 32-bit machine, p is 32-bits



Please note! Java does NOT support pointers
It supports references instead.

©GK Fall 2008

More Pointer Arithmetic



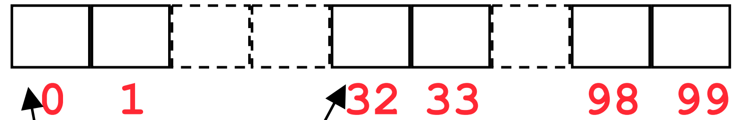
- address one past the end of an array is ok for pointer comparison only
- what's at `*(begin+44)`?
- what does `begin++` mean?
- how are pointers compared using < and using == ?
- what is value of `end - begin`?

```
char * a = new char[44];
char * begin = a;
char * end = a + 44;
while (begin < end)
{
    *begin = 'z';
    begin++;
}
```


©GK Fall 2008

More Pointers & Arrays

```
int * a = new int[100];
```




```
a is a pointer
*a is an int
a[0] is an int (same as *a)
a[1] is an int
a+1 is a pointer
a+32 is a pointer
*(a+1) is an int (same as a[1])
*(a+99) is an int
*(a+100) is trouble
```


©GK Fall 2008

Array Example (in C++)

```
#include <iostream.h>

main()
{
  int *a = new int[100];
  int *p = a;
  int k;

  for (k = 0; k < 100; k++)
    {
      *p = k;
      p++;
    }

  cout << "entry 3 = " << a[3] <<
endl;

}
```


©GK Fall 2008

Array of Classes (Linked List)

```
#include <iostream.h>
class node {
public:
  int me;
  node *next;
};
main()
{
  node *ar = new node[10];
  node *p = ar;
  int k;
  for (k = 0; k < 9; k++)
    {
      p->me = k;
      p->next = &ar[k+1];
      p++;
    }
 p->me = 9;
  p->next = NULL;
  p = &ar[0];
  while (p != NULL) {
    cout << p->me << " " <<
 hex << p << " " << p->next <<
 endl;
    p = p->next;
  }
}
```

- Given `ar = 0x10000`, what does memory layout look like?

©GK Fall 2008

Memory Layout

Output

Me	p	p->next
0	0x26ca8	0x26cb0
1	0x26cb0	0x26cb8
2	0x26cb8	0x26cc0
3	0x26cc0	0x26cc8
4	0x26cc8	0x26cd0
5	0x26cd0	0x26cd8
6	0x26cd8	0x26ce0
7	0x26ce0	0x26ce8
8	0x26ce8	0x26cf0
9	0x26cf0	0x0

Memory Address	Memory Contents	Source Symbol
0x26ca8	0	me } ar[0]
	0x26cb0	next } ar[0]
0x26cb0	1	
	0x26cb8	
0x26cb8	2	
	0x26cc0	
0x26cc0	3	
	0x26cc8	
0x26cc8	4	
	0x26cd0	
0x26cd0	5	
	0x26cd8	
0x26cd8	6	
	0x26ce0	
0x26ce0	7	
	0x26ce8	
0x26ce8	8	
	0x26cbf0	
0x26cf0	9	me } ar[9]
	0x0	next } ar[9]

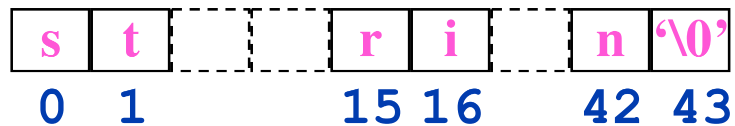
me is int (4 bytes)

next is node* (4 bytes)

©GK Fall 2008

C-Style Strings as Arrays

- A string is an array of characters with '\0' at the end
- Each element is one byte, ASCII code
- '\0' is null (ASCII code 0)



©GK Fall 2008

Composite Data Structure layout: Example

```
class foo {
public:
      int a;
      char c[3];
      char *p;
      foo * next;
};
foo x = new foo();
```

How is `x` **stored in memory?**

If `&X = 0x000f4020` **What is** `&(x.p)` **?**

©GK Fall 2008

Strlen()

- `strlen()` returns the # of characters in a string
  - same as # elements in char array?

```
int strlen(char * s)
 // pre: '\0' terminated
 // post: returns # chars
 {
     int count=0;
     while (*s++)
        count++;
     return count;
 }
```


©GK Fall 2008

Bit Manipulations

Problem

- 32-bit word contains many values
  - e.g., input device, sensors, etc.
  - current x,y position of mouse and which button (left, mid, right)
- Assume x, y position is 0-255
- How many bits for position?
- How many for button?

Goal

- Extract position and button from 32-bit word
- Need operations on individual bits of binary numbers

©GK Fall 2008

Bitwise AND / OR

- **&** operator performs bitwise **AND**
- **|** operator performs bitwise **OR**
- Per bit

AND	OR
0 & 0 = 0	0 | 0 = 0
0 & 1 = 0	0 | 1 = 1
1 & 0 = 0	1 | 0 = 1
1 & 1 = 1	1 | 1 = 1

- For multiple bits, apply operation to individual bits in same position

AND	OR
011010	011010
101110	101110
001010	111110

©GK Fall 2008

Mouse Example

- 32-bit word with x,y and button fields
  - bits 0-7 contain x position
  - bits 8-15 contain y position
  - bits 16-17 contain button (0 = left, 1 = middle, 2 = right)
- to extract value need to clear all other bits
- How do I use bitwise operations to do this?

button y x

```
0x1a34c = 01 1010 0011 0100 1100
```


©GK Fall 2008

Mouse Solution

- **AND** with a bit mask
  - specific values that clear some bits, but pass others through
- To extract x position use mask 0x000ff

```
xpos = 0x1a34c & 0x000ff

                button    y           x
   0x1a34c = 01 1010 0011 0100 1100
   0x000ff = 00 0000 0000 1111 1111
   0x0004c = 00 0000 0000 0100 1100
```


©GK Fall 2008

More of the Mouse Solution

- To extract y position use mask 0x0ff00

```
ypos = 0x1a34c & 0x0ff00
```

- Similarly, button is extracted with mask 0x3ffff

```
button = 0x1a34c & 0x30000
```

- Not quite done...why?

```
                  button    y              x
0x1a34c =  01 1010 0011 0100 1100
0x0ff00 =  00 1111 1111 0000 0000
0x0a300 =  00 1010 0011 0000 0000
```


©GK Fall 2008

The SHIFT operator

- `>>` is shift right, `<<` is shift left, operands are `int` and number of positions to shift
- `(1 << 3)` is ...000001 -> ...0001000 (it's 2^3)
- `0xff00` is `(0xff << 8)`, and `0xff` is `(0xff00 >> 8)`
- So, true `ypos` value is:

```
ypos = (0x1a34c & 0x0ff00) >> 8

button = (0x1a34c & 0x30000) >> 16
```


©GK Fall 2008

Mask and Shift example: Extracting Parts of Floating Point Number

```
// x is 32-bit word holding a float

#define EXP_BITS 8

#define MANTISSA_BITS 23

#define SIGN_MASK 0x80000000

#define EXP_MASK  0x7f800000

#define MANTISSA_MASK  0x007fffff

class myfloat {

public:

  int sign;

  unsigned int exp;

  unsigned int mantissa;

};

myfloat num;

 num->sign = (x & SIGN_MASK) >> (EXP_BITS + MANTISSA_BITS);

 num->exp  = (x & EXP_MASK) >> MANTISSA_BITS;

 num->mantissa = x & MANTISSA_MASK;
```

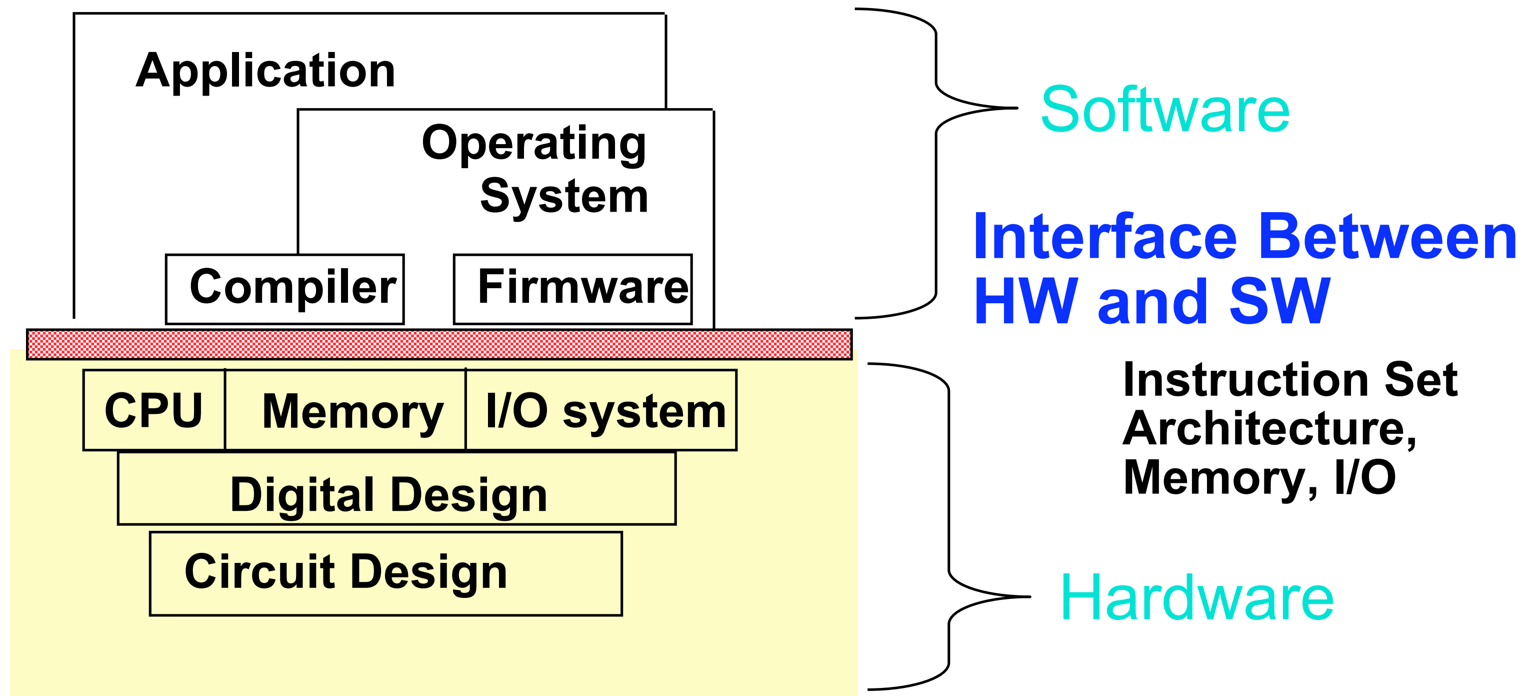

©GK Fall 2008

Summary

- Computer memory is a linear array of bytes
- Pointer is memory location that contains address of another memory location
- Bitwise operations
- Code examples are linked to course web page
- We'll visit these topics again throughout semester.

©GK Fall 2008

Instruction Set Architecture



©GK Fall 2008

Levels of Representation

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

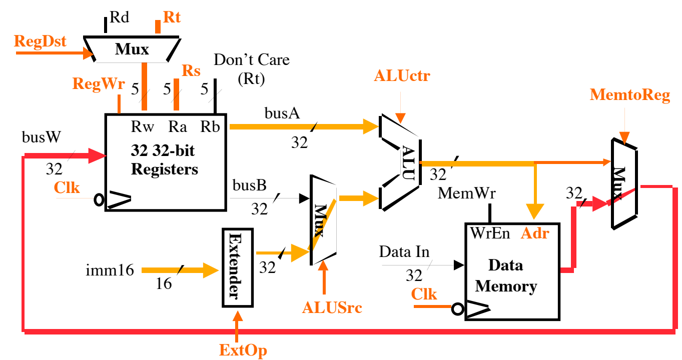
Machine Interpretation

Control Signal Specification

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $15,   0($2)
lw $16,   4($2)
sw$16,   0($2)
sw$15,   4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

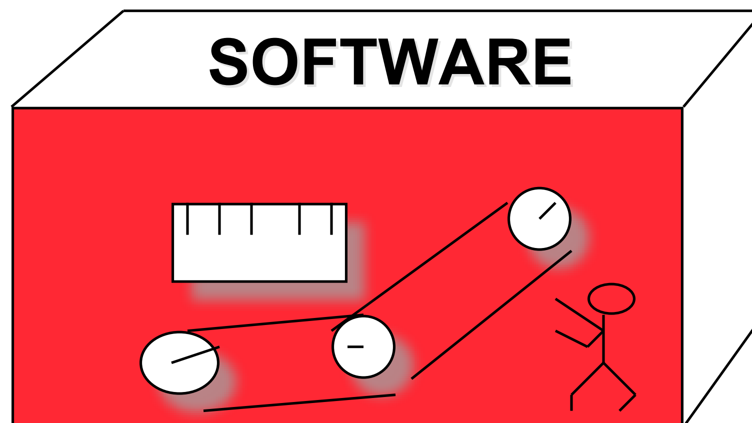


©GK Fall 2008

Computer Architecture?

. . . the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

Amdahl, Blaaw, and Brooks, 1964



©GK Fall 2008

Requirements for ISA

```
#include <stdio.h>

main()
{
  int a[100];
  int k;

  for (k = 0; k < 100; k++)
    {
      a[k] = k*7+5;
    }

  printf("entry 3 = %d\n", a[3]);
}
```

**What primitive operations do we need?
(i.e., What should be implemented in hardware?)**

©GK Fall 2008

Design Space of ISA

Five Primary Dimensions

• Operations	add, sub, mul, . . . How is it specified?
• Number of explicit operands	(0, 1, 2, 3)
• Operand Storage	Where besides memory?
• Memory Address	How is memory location specified?
• Type & Size of Operands	byte, int, float, vector, . . . How is it specified?

Other Aspects

• Successor instruction	How is it specified?
• Conditions	How are they determined?
• Encoding	Fixed or variable? Wide?
• Parallelism	

©GK Fall 2008

Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next (JAVA VM)

General Purpose Register:

2 address	add A B	A ← A + B
3 address	add A B C	A ← B + C

Load/Store:

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

©GK Fall 2008

Accumulator

- Instruction set: Accumulator is implicit operand

  one explicit operand

  add, sub, mult, div, . . .

  clear

Example: a*b - (a+c*b)

Instruction	Accumulator
clear	0
add c	2
mult b	6
add a	10
st tmp	10
clear	0
add a	4
mult b	12
sub tmp	2

9 instructions

time

a	4
b	3
c	2
tmp	

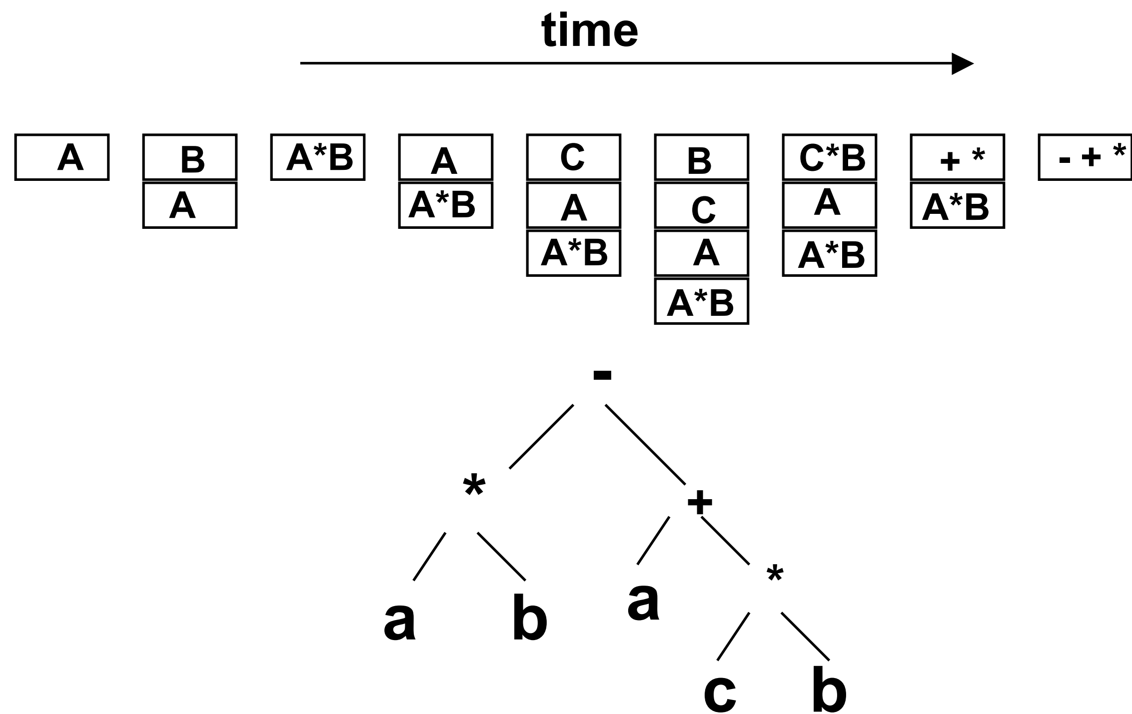
©GK Fall 2008

Stack Machines

- **Instruction set:**
  - **add, sub, mult, div . . . Top of stack (TOS) and TOS+1 are implicit**
  - **push A, pop A   TOS is implicit operand, one explicit operand**

Example: a*b - (a+c*b)

push a
push b
mult
push a
push c
push b
mult
add
sub
9 instructions



©GK Fall 2008

2-address ISA

- Instruction set: Two explicit operands, one implicit
  - add, sub, mult, div, …
  - one source operand is also destination
    - add a,b   a <- a + b

Example: a*b - (a+c*b)

	tmp1, tmp2
add tmp1, b	3, ?
mult tmp1, c	6, ?
add tmp1, a	10, ?
add tmp2, b	10, 3
mult tmp2, a	10, 12
sub tmp2, tmp1	10, 2

a	4
b	3
c	2
tmp1	
tmp2	

6 instructions

©GK Fall 2008

3-address ISA

- **Instruction set: Three explicit operands, ZERO implicit**
  - add, sub, mult, div, …
    - add a,b,c   a <- b + c

Example: a*b - (a+c*b)

	tmp1, tmp2
mult tmp1, b, c	6, ?
add tmp1, tmp1, a	10, ?
mult tmp2, a, b	10, 12
sub tmp2, tmp2, tmp1	10, 2

a	4
b	3
c	2
tmp1	
tmp2	

4 instructions

©GK Fall 2008

Adding Registers to an ISA

- A place to hold values that can be named within the instruction
- Like memory, but much smaller
  - 32-128 locations
- How many bits to specify a register?

r0

• • •

r31

Byte Address | Data

0 00110110
1 00001100
2
3
4
•
•
•
2^{32}-1-4
2^{32}-1

©GK Fall 2008

3-address General Purpose Register ISA

- **Instruction set:** Three explicit operands, ZERO implicit

  add, sub, mult, div, …

  add a,b,c   a <- b + c

Example: a*b - (a+c*b)

	r1, r2
mult r1, b, c	6, ?
add r1, r1, a	10, ?
mult r2, a, b	10, 12
sub r2, r2, r1	10, 2

a	4
b	3
c	2

4 instructions

©GK Fall 2008

LOAD / STORE ISA

- **Instruction set:**

  add, sub, mult, div, … only on operands in registers

  ld, st, to move data from and to memory, only way to access memory

Example: a*b - (a+c*b)

Instruction	r1, r2, r3
ld r1, c	2, ?, ?
ld r2, b	2, 3, ?
mult r1, r1, r2	6, 3, ?
ld r3, a	6, 3, 4
add r1, r1, r3	10, 3, 4
mult r2, r2, r3	10, 12, 4
sub r3, r2, r1	10, 12, 2

a	4
b	3
c	2

7 instructions

©GK Fall 2008

Using Registers to Access Memory

- Registers can hold memory addresses

Given

```
int x; int *p;

p = &x;

*p = *p + 8;
```

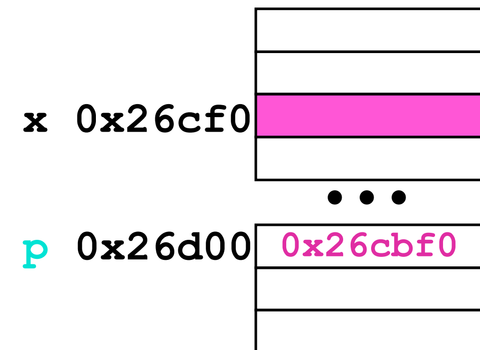
Instructions

```
ld r1, p            // r1 <- mem[p]

ld r2, r1           // r2 <- mem[r1]

add r2, r2, 0x8     // increment x by 8

st r1, r2           // mem[r1] <- r2
```



- Many different ways to address operands
  - not all Instruction sets include all modes

©GK Fall 2008

Making Instructions Machine Readable

- So far, still too abstract
  - add r1, r2, r3
- Need to specify instructions in machine readable form
- Bunch of Bits
- Instructions are bits with well defined fields
  - Like a floating point number has different fields
- Instruction Format
  - Establishes a mapping from “instruction” to binary values
  - Which bit positions correspond to which parts of the instruction (operation, operands, etc.)

©GK Fall 2008

A "Typical" RISC

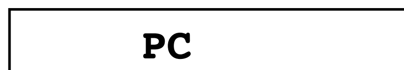
- 32-bit fixed format instruction (3 formats)
- 32 64-bit GPR (R0 contains zero)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection

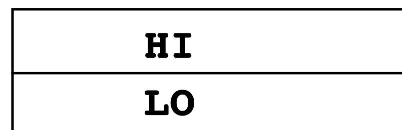
see: SPARC, MIPS, MC88100, AMD2900, i960, i860
PARISC, PowerPC, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

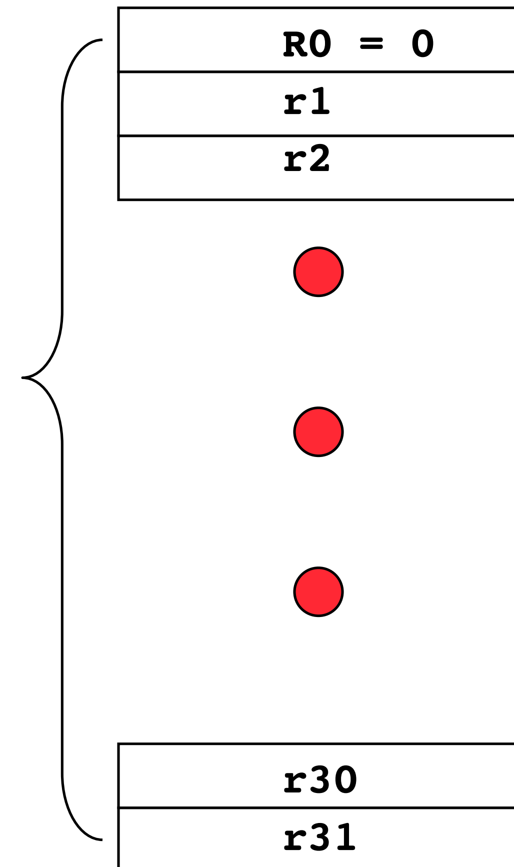
©GK Fall 2008

Example: MIPS Integer Registers

- **Registers: fast memory, Integral part of the CPU.**
- **Programmable storage 2^{32} bytes**
- **31 x 32-bit GPRs (R0 = 0)**
- **32 x 32-bit FP regs (paired DP)**
- **32-bit HI, LO, PC**



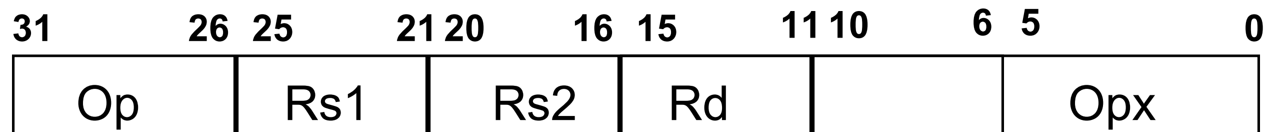




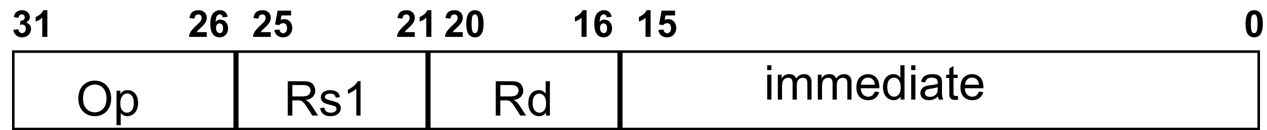
©GK Fall 2008

Example: MIPS

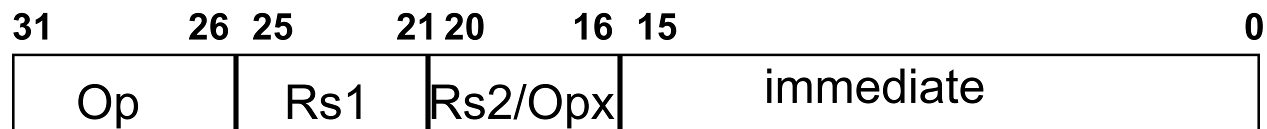
Register-Register



Register-Immediate



Branch



Jump / Call

©GK Fall 2008

Reading Assignment

<u>Reading:</u>

- Chapter 3.1-3.3, 3.6 pages 160-170, 189-217
- Read Chapter 1, Skim 2

©GK Fall 2008