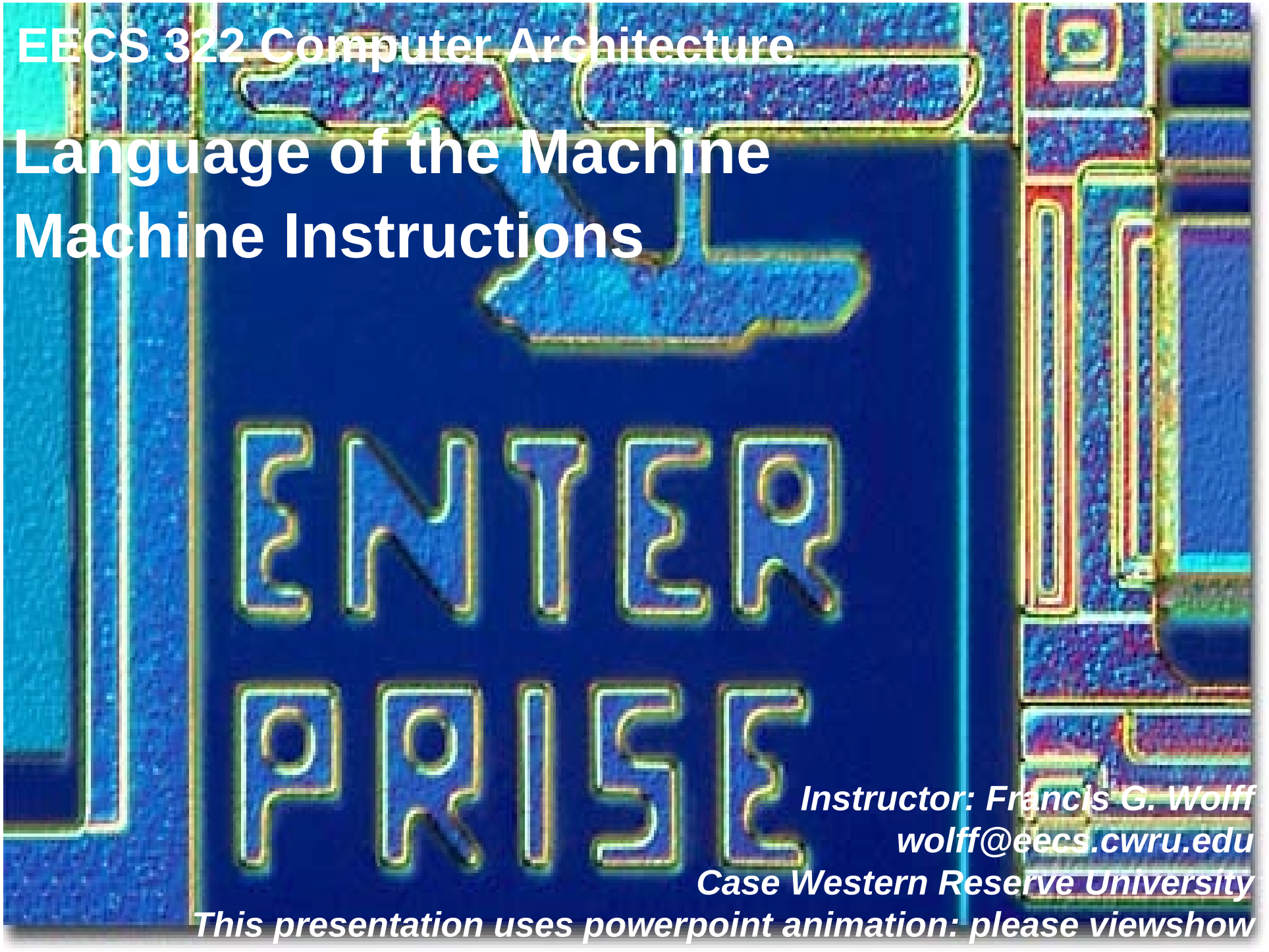
EECS 322 Computer Architecture

Language of the Machine
Machine Instructions

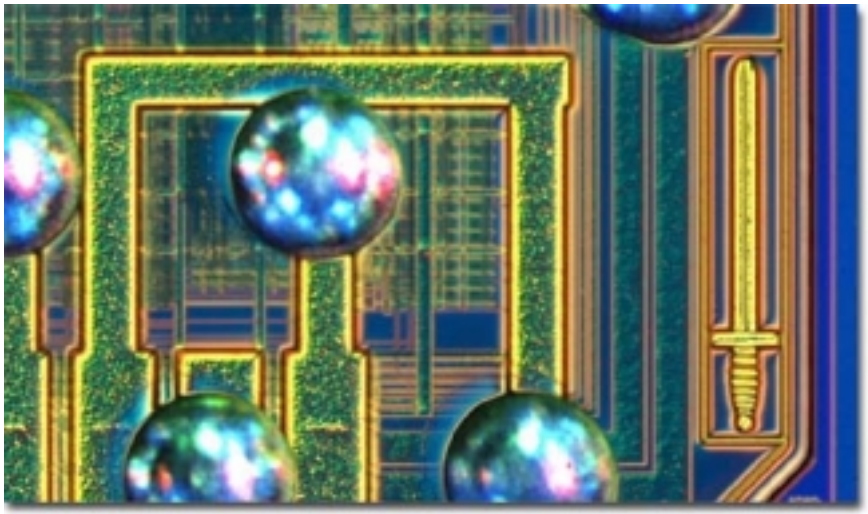


Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

Signatures and Silicon Art

- Just as the great architects, place their hidden Φ signature, so too do computer designers.



The “Pentium Killer” Macintosh G3 chips were code-named "Arthur" as in Camelot, and the sword represents Excalibur.

Motorola/IBM PowerPC 750

MIPS R10000 Processor

Review: Function calling

- Follow calling conventions & nobody gets hurt.

- Function Call Bookkeeping:
  - Caller:
    - Arguments — $a0, $a1, $a2, $a3, ($sp)
    - Return address — $ra
    - Call function — jal label # $ra=pc+4;pc=label
  - Callee:
    - Not restored — $t0 - $t9
    - Restore caller's — $s0 - $s7, $sp, $fp
    - Return value — $v0, $v1
    - Return — jr $ra # pc = $ra

Argument Passing greater than 4

- C code fragment

```
g=f(a, b, c, d, e);
```

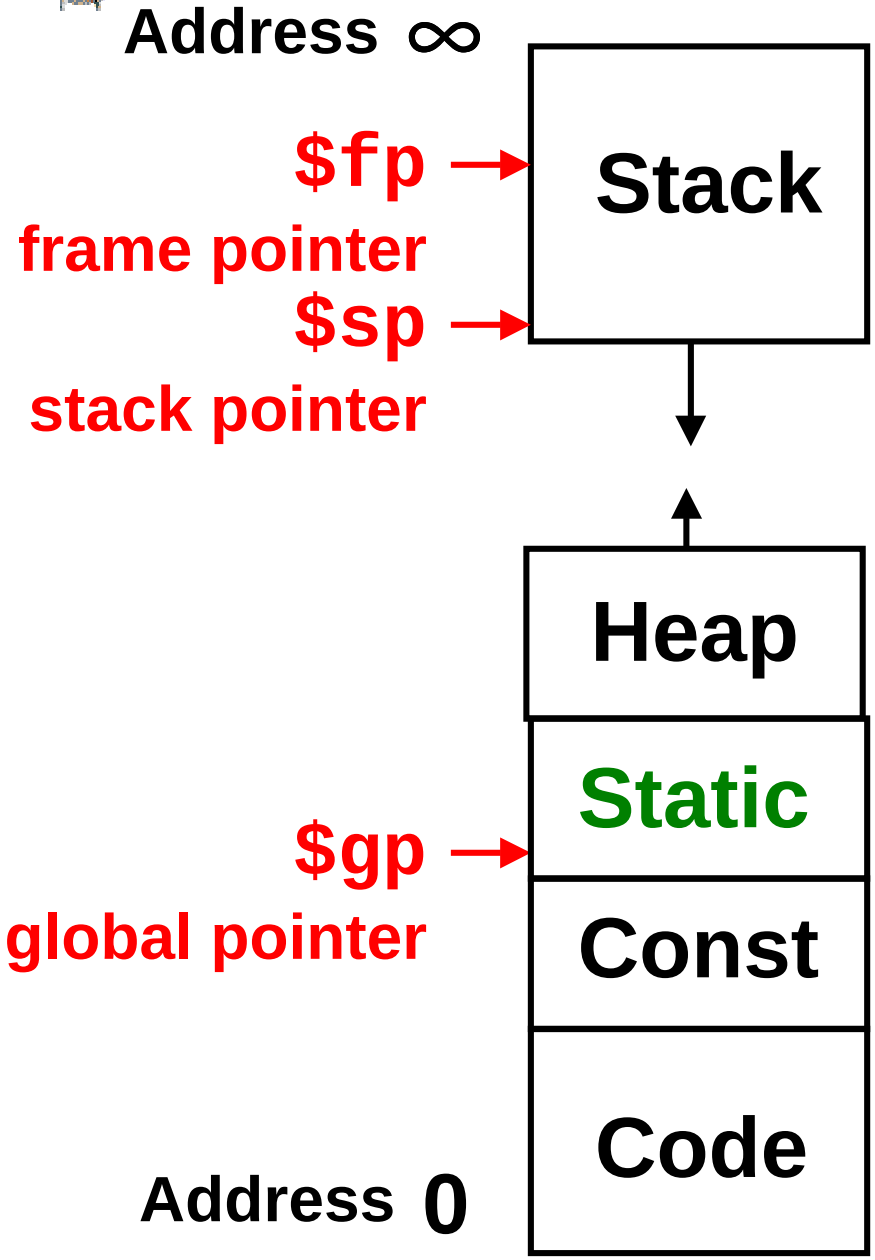
- MIPS assembler

```
addi    $sp, $sp, -4
sw      $s4, 0($sp)   # push e
add     $a3, $s3, $0  # register push d
add     $a2, $s2, $0  # register push c
add     $a1, $s1, $0  # register push b
add     $a0, $s0, $0  # register push a
jal     f             # $ra = pc + 4
add     $s5, $v0, $0  # g=return value
```

Review: MIPS registers and conventions

Name	Number	Conventional usage
$0	0	Constant 0
$v0-$v1	2-3	Expression evaluation & function return
$a0-$a3	4-7	Arguments 1 to 4
$t0-$t9	8-15,24,35	Temporary (not preserved across call)
$s0-$s7	16-23	Saved Temporary (preserved across call)
$k0-$k1	26-27	Reserved for OS kernel
$gp	28	Pointer to global area
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address (used by function call)

Review: Program memory layout



Memory Management: R:read only W:write only X:execute only

RW	(.sdata) Stack data for saved procedure information: return address, dynamic variables, >4 arguments
RW	Explicitly created space, e.g., malloc(); C pointers
RW	(.data) Global static vars (.bss initialized to 0)
R	(.rdata) const variables and const strings
X	(.text) Program: machine instructions

Alternate Static allocation (scope: public to everyone)

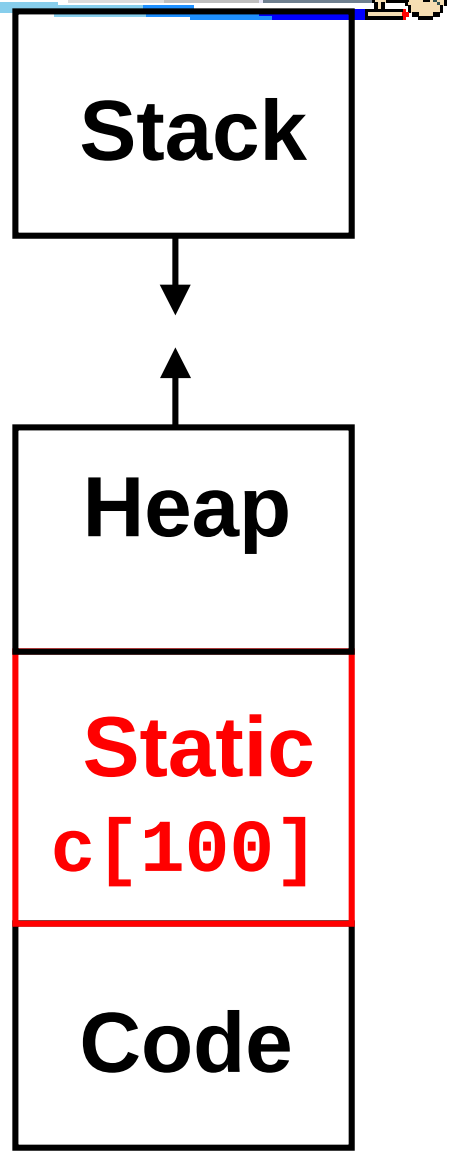
- Static declaration

```
int c[100];

int *sumarray(int a[],int b[]) {
    int i;
    static int c[100];

    for(i=0;i<100;i=i+1)
        c[i] = a[i] + b[i];
    return c;
}
```

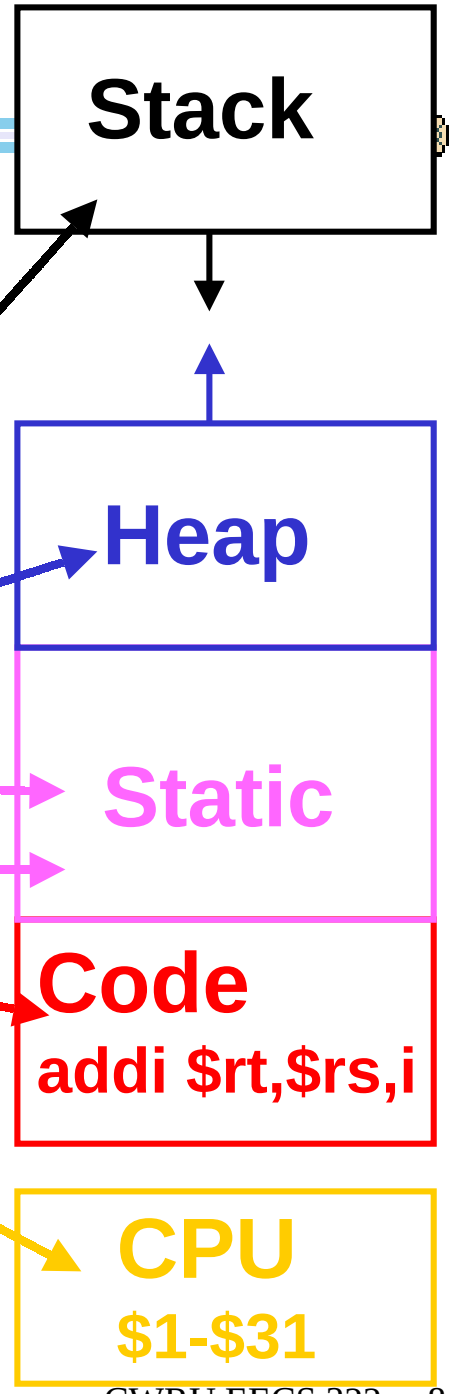
- The variable scope of `c` is very public and is accessible to everyone outside the function



Review: memory allocation model

```
int e[100];
int *sumarray(int a[],int b[]) {
    register int x;
    int i; int d[32]; int *p;
    static int c[100];
    const int f = 3;

    p = (int *)malloc(sizeof(int));
    . . .
}
```



- public scope: variable `e`
- private scope:
  - `x, i, d, p, c, f`

Performance of memory model

speed performance: fastest to slowest

no setup time, fast access:	`register int x;`
no setup time, fast access:	`const int f = 3;`
no setup time($gp), slow access (lw,sw):	`static int c;`
fast setup time(addi $sp,$sp,-size), slow access (lw,sw):	`int i; int d[32];`
high setup time(search loop), slow access (lw,sw):	`malloc(); free();`

storage performance: reuse

unrestricted:	`malloc(); free();`
unrestricted but cannot free:	`static int c;`
nested within scope:	`int i; int d[32];`
limited resource:	`register int x;`
restricted:	`const int f = 3;`

Global/Static storage examples

- Global C code fragment

```
(outside a function)
char a;
char b[ ] = “hello\n”
char c[6] = { -1, 20, 0xf };
short d;
int e;
int f = 0x1f;
int *g;
int *******h;
int *i[5];
int (*j)(int x, int y);
```

- MIPS assembler

```
        .data
a:      .byte       0
b:      .asciiz     “hello\n”
c:      .byte -1,20,0xf,0,0,0
d:      .half       0
e:      .word       0
f:      .word       0x1f
g:      .word       0
h:      .word       0
i:      .word       0
j:      .word       0
```

Global variables

- Global C code fragment (outside a function)

```
char  a;
char  *b;
char  *c = &a;
char  ***d;
short e;
short *f;
short ***g;
float h;
float  *i;
double **j
```

- MIPS assembler

```
        .data
a:      .byte   0
b:      .word   0
c:      .word   a
d:      .word   0
e:      .half   0
f:      .word   0
g:      .word   0
h:      .float  0
i:      .word   0
j:      .word   0
```

Dynamic Allocation and access

- C code

```
funcion( ) {
char  a;
char  *b;
char  *c=&a;
char  ***d;
short e;
short *f;
short ***g;
float h;
float  *i;
double **j
```

- add $fp,$sp,$0
- add $sp,$sp,-67

```
# 0($fp)     #a:   .byte
# -1($fp)    #b:   .word
# -5($fp)    #c:   .word
# -9($fp)    #d:   .word
# -13($fp)   #e:   .half
# -15($fp)   #f:   .word
# -19($fp)   #g:   .word
# -23($fp)   #h:   .float
# -27($fp)   # i:  .word
# -31($fp)   # j:  .word
```

- Stack offset

```
add $sp,$sp,-67
31($sp)
30($sp)
26($sp)
22($sp)
20($sp)
16($sp)
12($sp)
8($sp)
4($sp)
0($sp)
```

Dynamic initialization of variables

- C code

```
funcion( ) {


char  *c=&a;
```

- add $fp,$sp,$0
- add $sp,$sp,-67

```
# -5($fp)     #c:     .word
...
addi          $t1,$fp,0     #address of a
sw            $t1,5($fp)    #initialize c
```

Static/Global Struct (by default a public class)

- C code

```
struct point {
   float x, y;
 };


struct point *p;

struct point  g;


struct point h={7,8};
```

- Same as C++ code

```
class point {
  public:
        float  x, y;
};
```

- MIPS assembler

```
p:          .word       0

g:          .float      0
            .float      0

h:          .float      7,8
```

Static Classes: inheritance

- C++ code

```
class point { /* base */
public:
   float x, y;
 };
class rectangle:
     public point { /*derived */
public:
  float x2, y2;
};
/* create an instance */
class point a;
class rectangle b;
```

- MIPS Assembler

```
a:      .float      0       #x
        .float      0       #y

b:      .float      0       #x
        .float      0       #y
        .float      0       #x2
        .float      0       #x3
```

Instruction as Number Example (decimal)

- C code: `i = j + k; /* i-k:$s0-$s2 */`
- Assembly: `add $s0,$s1,$s2 #s0=s1+s2`
- Decimal representation:

0	17	18	16	0	32

  - Segments called fields
  - 1st and last tell MIPS computer to add
  - 2nd is 1st source operand (17 = $s1)
  - 3rd is 2nd source operand (18 = $s2)
  - 4th is destination operand (16 = $s0)
  - 5th unused, so set to 0

Order differs: destination 1st v.last! (common error)

Numbers: Review

- **Number Base B => B symbols per digit:**
  - **Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
    Base 2 (Binary): 0, 1
- **Number representation: $d_4d_3d_2d_1d_0$**
  - **$d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$**
  - **$10010_{ten} = 1x10^4 + 0x10^3 + 0x10^2 + 1x10^1 + 0x10^0$**
    $= 1x10000 + 0x1000 + 0x100 + 1x10 + 0x1$
    $= 10000 + 0 + 0 + 10 + 0$
    $= 10010_{ten}$
  - **$10010_{two} = 1x2^4 + 0x2^3 + 0x2^2 + 1x2^1 + 0x2^0$**
    $= 1x16 + 0x8 + 0x4 + 1x2 + 0x1$
    $= 16_{ten} + 0_{ten} + 0_{ten} + 2_{ten} + 0_{ten}$
    $= 18_{ten}$

Numbers: Decimal, Binary, Octal, Hex

base 10: Decimal
base 2: Binary
base 8: Octal
base 16: Hex

Octal example:
01111101
=175

Hex example:
01111101
=7d

00	00000	00	16	10000	10
01	00001	01	17	10001	11
02	00010	02	18	10010	12
03	00011	03	19	10011	13
04	00100	04	20	10100	14
05	00101	05	21	10101	15
06	00110	06	22	10110	16
07	00111	07	23	10111	17
08	01000	08	24	11000	18
09	01001	09	25	11001	19
10	01010	0a	26	11010	1a
11	01011	0b	27	11011	1b
12	01100	0c	28	11100	1c
13	01101	0d	29	11101	1d
14	01110	0e	30	11110	1e
15	01111	0f	31	11111	1f

Instruction as Number Example (binary)

- C code: `i = j + k; /* i-k:$s0-$s2 */`
- Assembly: `add $s0,$s1,$s2 #s0=s1+s2`
- Decimal representation:

0	17	18	16	0	32

- Binary representation:

000000	10001	10010	10000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - Called Machine Language Instruction
  - Layout called Instruction Format
  - All MIPS instructions 32 bits (word): simple!

Big Idea: Stored-Program Concept

- **Computers built on 2 key principles:**
  - **1) Instructions are represented as numbers**
  - **2) Programs can be stored in memory to be read or written just like numbers**
- **Simplifies SW/HW of computer systems:**
  - **– Memory technology for data also used for programs**
  - **– Compilers can translate HLL (data) into machine code (instructions)**

Big Consequence #1: Everything addressed

- Since all instructions and data are stored in memory as numbers, everything has a memory address: instructions, data words
  - branches use memory address of instruction
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Better name is Instruction Address Register, but PC is traditional name

Big Consequence #2: Binary Compatibility

- **Programs are distributed in binary form**
  - **Programs bound to instruction set architecture**
  - **Different version for Macintosh and IBM PC**
- **New machines want to run old programs ("binaries") as well as programs compiled to new instructions**
- **Leads to instruction set evolving over time**
- **Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium II); could still run program from 1981 PC today**

Instruction Format Field Names

– Fields have names:

– op: basic operation of instruction, "opcode"
– rs: 1st register source operand
– rt: 2nd register source operand
– rd: register destination operand, gets the result
– shamt: shift amount (used later, so 0 for now)
– funct: function; selects the specific variant of the operation in the op field; sometimes called the function code

Instruction Formats

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- What if want longer fields? e.g, `lw $2 32($5)`
  - 5 bits => address of 2^5 or 32 => too small
  - But want all instructions same length!
- Principle: Good design demands good compromises
  - Add 2nd format with larger address

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

  - 1st format R (register); 2nd format I (immediate)

Notes about Register and Imm. Formats

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address		
	6 bits	5 bits	5 bits	16 bits		

- To make it easier for hardware (HW), 1st 3 fields same in R-format and I-format
- Alas, `rt` field meaning changed
  - R-format: `rt` is 2nd source operand
  - I-format: `rt` can be register destination operand
- How HW know which format is which?
  - Distinct values in 1st field (op) tell whether last 16 bits are 3 fields (R-format) or 1 field (I-format)

Instructions, Formats, “opcodes”

Instruction	Format	op	funct
–add	Register	0	32
–sub	Register	0	34
–slt	Register	0	42
–jr	Register	0	8
–lw	Immediate	35	
–sw	Immediate	43	
–addi	Immediate	8	
–beq	Immediate	4	
–bne	Immediate	5	
–slti	Immediate	10	

Register Format if op field = 0

Immediate Instruction in Machine Code

- C code: `i = j + 4; /* i,j:$s0,$s1 */`
- Assembly: `addi $s0,$s1,4` *`#s0=$s1+4`*
- Format:

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- Decimal representation:

8	17	16	4
6 bits	5 bits	5 bits	16 bits

- Binary representation:

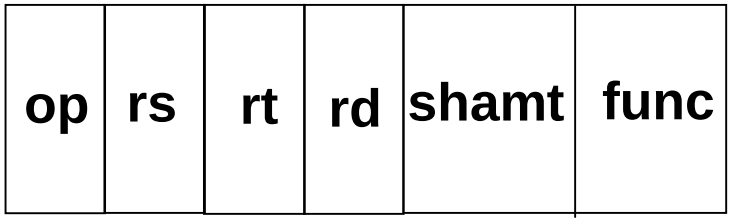
001000	10001	10000	0000 0000 0000 0100
6 bits	5 bits	5 bits	16 bits

MIPS instructions

ALU	alu $rd,$rs,$rt	$rd = $rs <alu> $rt
ALUi	alui $rd,$rs,value	$rd = $rs <alu> value
Data Transfer	lw $rt,offset($rs)	$rt = Mem[$rs + offset]
	sw $rt,offset($rs)	Mem[$rs + offset] = $rt
Branch	beq $rs,$rt,offset	$pc = ($rd == $rs)? (pc+4+offset):(pc+4);
Jump	j address	pc = address

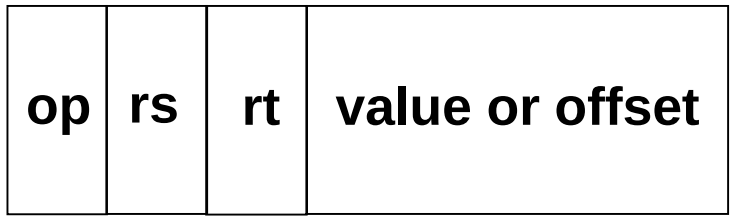
MIPS fixed sized instruction formats

R - Format



ALU	alu $rd,$rs,$rt

I - Format



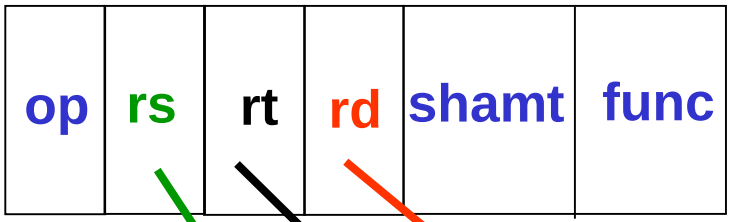
ALUi	alui $rt,$rs,value
Data Transfer	lw $rt,offset($rs) sw $rt,offset($rs)
Branch	beq $rs,$rt,offset

J - Format

Jump	j address

Assembling Instructions

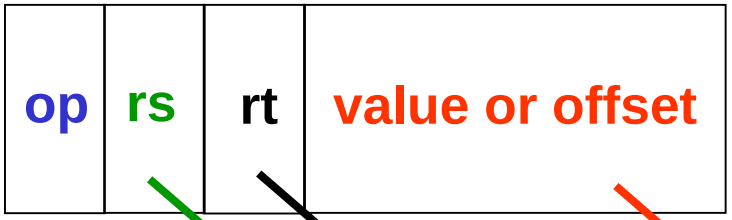
Suppose there are 32 registers, addu opcode=001001, addi op=001000



ALU **alu $rd,$rs,$rt**

0x00400020 **addu $23, $0, $31**

001001:00000:11111:10111:00000:000000



ALUi **alui $rt,$rs,value**

0x00400024 **addi $17, $0, 5**

001000:00000:00101:0000000000000101

MIPS instruction formats

Arithmetic

addi $rt, $rs, value

add $rd,$rs,$rt

Data Transfer

lw $rt,offset($rs)

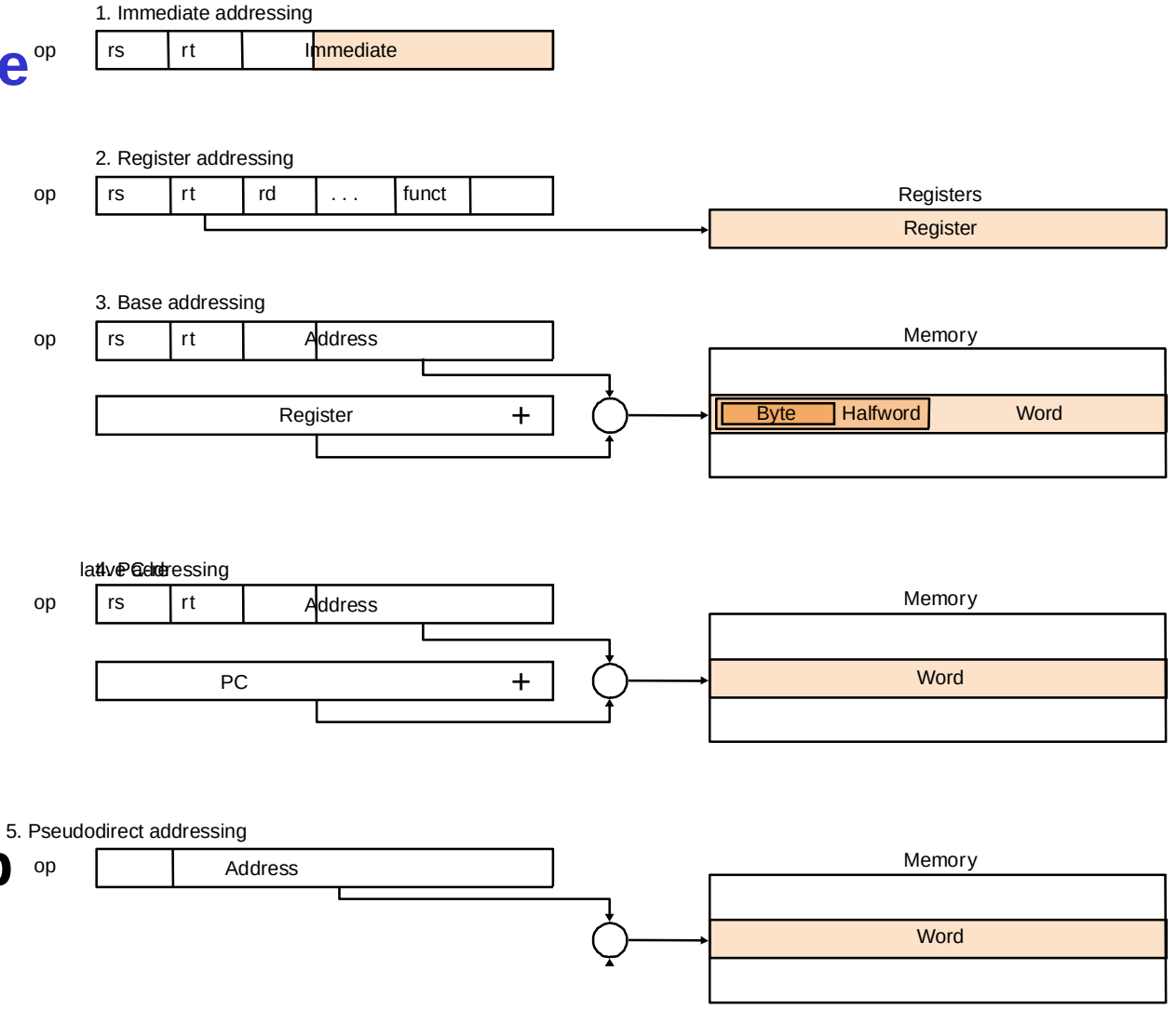
sw $rt,offset($rs)

Conditional branch

beq $rs,$rt,offset

Unconditional jump

j address



C function to MIPS Assembly Language

```
int power_2(int y) { /* compute x=2^y; */
        register int x, i; x=1; i=0; while(i<y) { x=x*2; i=i+1; }
        return x;
}
```

Exit condition of a while loop is
if (i >= y) then goto w2

Assember .s			Comments
	addi	$t0, $0, 1	# x=1;
	addu	$t1, $0, $0	# i=0;
w1:	bge	$t1,$a0,w2	# while(i<y) { /* bge= greater or equal */
	addu	$t0, $t0, $t0	# x = x * 2; /* same as x=x+x; */
	addi	$t1,$t1,1	# i = i + 1;
	beq	$0,$0,w1	# }
w2:	addu	$v0,$0,$t0	# return x;
	jr	$ra	# jump on register (pc = ra;)

Power_2.s: MIPS storage assignment

Byte address, not word address

```
        .text
0x00400020  addi   $8, $0, 1      # addi $t0, $0, 1
0x00400024  addu   $9, $0, $0     # addu $t1, $0, $0
0x00400028  bge    $9, $4, 2      # bge  $t1, $a0, w2
0x0040002c  addu   $8, $8, $8     # addi $t0, $t0, $t0
0x00400030  addi   $9, $9, 1      # addi $t1, $t1, 1
0x00400034  beq    $0, $0, -3     # beq  $0, $0, w1
0x00400038  addu   $2, $0, $8     # addu $v0, $0, $t0
0x0040003c  jr     $31            # jr   $ra
```

2 words after pc fetch

after bge fetch pc is 0x00400030 plus 2 words is 0x00400038

Machine Language Single Stepping

Assume power2(0); is called; then $a0=0 and $ra=700018

Values changes after the instruction!

$pc	$v0	$a0	$t0	$t1	$ra		
	$2	$4	$8	$9	$31		
00400020	?	0	?	?	700018	addi	$t0, $0, 1
00400024	?	0	1	?	700018	addu	$t1, $0, $0
00400028	?	0	1	0	700018	bge	$t1,$a0,w2
00400038	?	0	1	0	700018	add	$v0,$0,$t0
0040003c	1	0	1	0	700018	jr	$ra
00700018	?	0	1	0	700018	...	