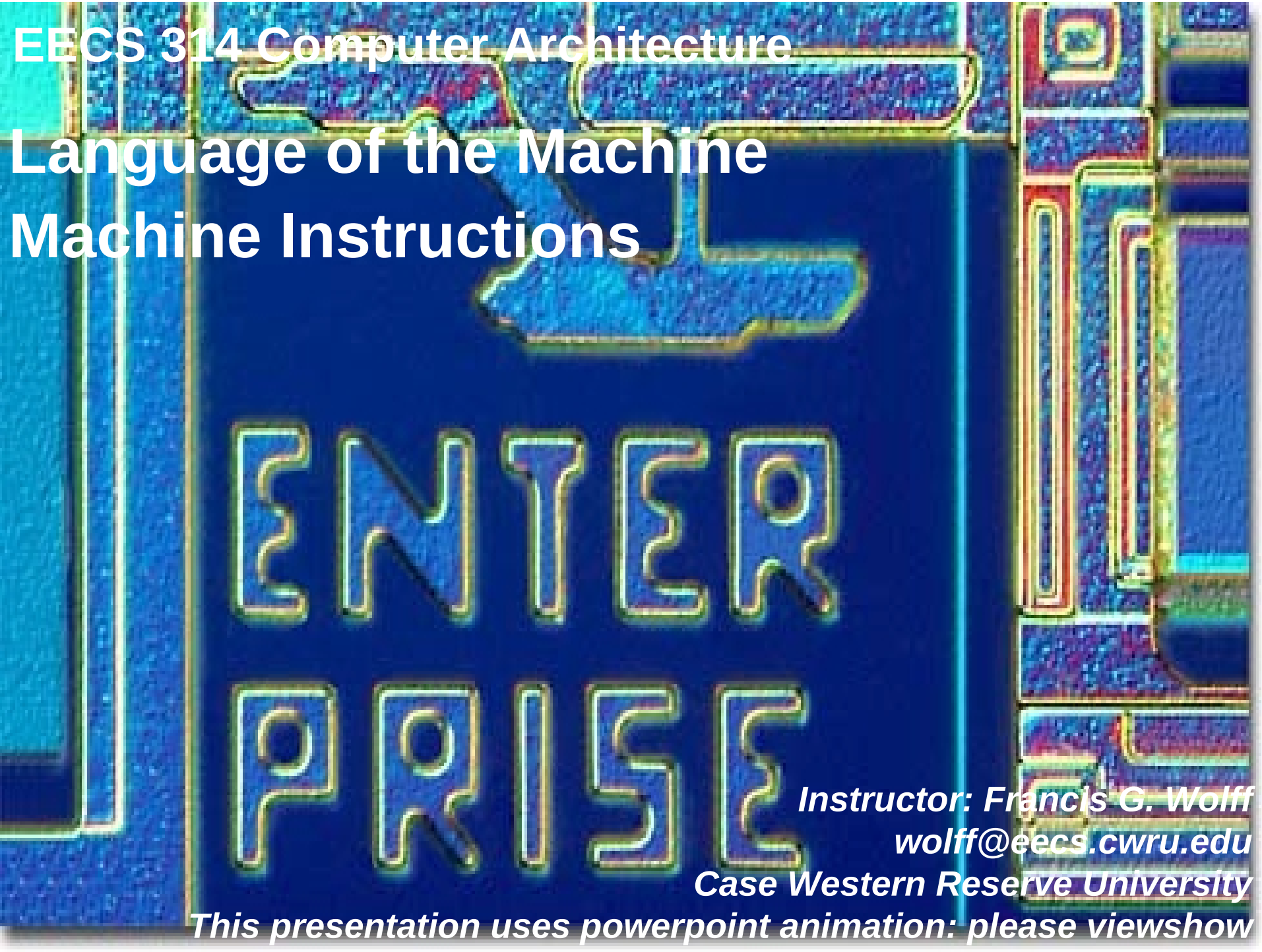




EECS 314 Computer Architecture

Language of the Machine

Machine Instructions

ENTER

PRISE

Instructor: Francis G. Wolff
wolff@eecs.cwru.edu

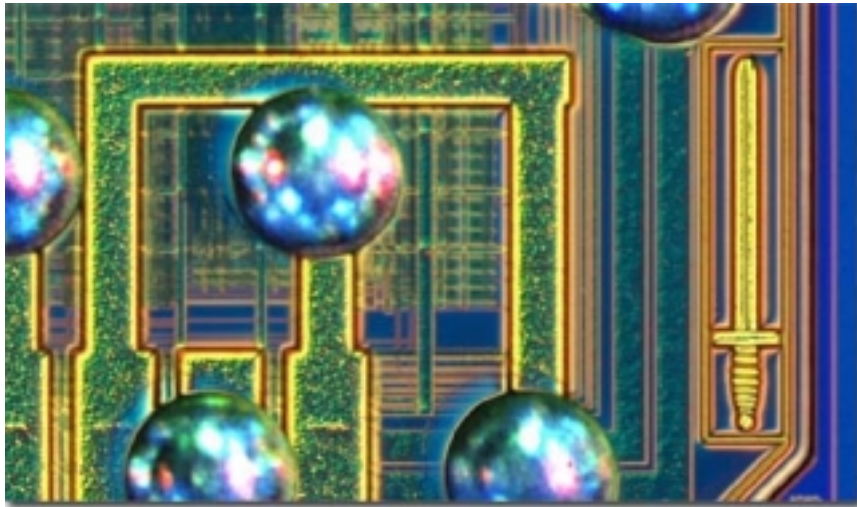
Case Western Reserve University

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`**

Compiling nested C func into MIPS

[updated]



```
int sumSquare(int x, int y) { return mult(x,x)+ y; }
```

sumSquare:

Prologue <pre> addi \$sp, \$sp, -4 sw \$ra, 0(\$sp) addi \$sp, \$sp, -4 sw \$a1, 0(\$sp) addi \$sp, \$sp, -4 sw \$a0, 0(\$sp) </pre>	<pre> # push return address # # push argument y = \$a1 # # push argument x = \$a0 # # mult(\$a0=x, \$a1=x) # call mult; \$v0=return # pop x # # pop y # # pop return address # </pre>
Body <pre> addi \$a1, \$a0, \$0 jal mult </pre>	
Epilogue <pre> lw \$a0, 0(\$sp) addi \$sp, \$sp, 4 lw \$a1, 4(\$sp) addi \$sp, \$sp, 4 lw \$ra, 8(\$sp) addi \$sp, \$sp, 4 add \$v0, \$v0, \$a1 jr \$ra </pre>	<pre> # return = \$v0 + (\$a1 = y) # return to caller </pre>

Compiling nested C func into MIPS: combine addi's

```
int sumSquare(int x, int y) {  
    return mult(x,x)+ y;  
}
```

sumSquare:

Prologue	addi	\$sp, \$sp, -12
	sw	\$ra, 8(\$sp)
	sw	\$a1, 4(\$sp)
	sw	\$a0, 0(\$sp)
Body	addi	\$a1, \$a0, \$0
	jal	mult
Epilogue	lw	\$a0, 0(\$sp)
	lw	\$a1, 4(\$sp)
	lw	\$ra, 8(\$sp)
	add	\$v0, \$v0, \$a1
	addi	\$sp, \$sp, 12
	jr	\$ra

push stack stack
push return addr
push y
push x
mult(x,x)
call mult
pop x
pop y
pop return addr
mult()+y
pop stack space

Argument Passing greater than 4

- C code fragment

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

- MIPS assembler

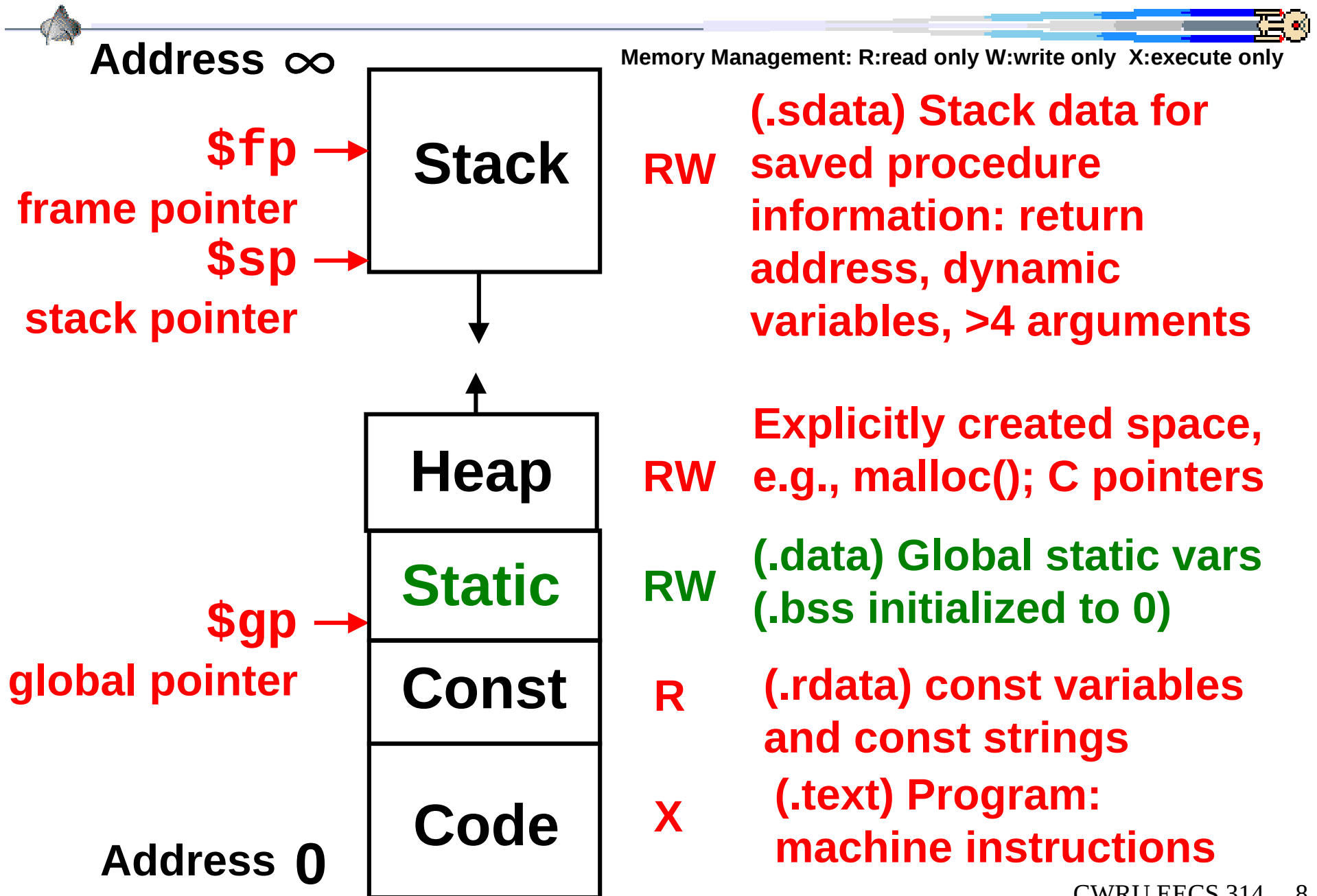
```
addi    $sp, $sp, -4
sw       $s4, 0($sp)    # push e
add      $a3, $s3, $0    # register push d
add      $a2, $s2, $0    # register push c
add      $a1, $s1, $0    # register push b
add      $a0, $s0, $0    # register push a
jal      f               # $ra = pc + 4
add      $s5, $v0, $0    # g=return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

Review: MIPS registers and conventions

[updated]

<u>Name</u>	<u>Number</u>	<u>Conventional usage</u>
\$0	0	Constant 0 (writing \$0 will not change \$0)
\$at	1	Reserved by assembler for pseudo-instructions
\$v0-\$v1	2-3	Function “return value;” & temporary usage
\$a0-\$a3	4-7	“function(\$a0, \$a1, \$a2 \$a3);” calling arguments
\$t0-\$t9	8-15,24,25	Temporary (not preserved across call)
\$s0-\$s7	16-23	Saved Temporary (preserved across call)
\$k0-\$k1	26-27	Reserved for OS kernel
\$gp	28	Global pointer (global static variables)
\$sp	29	Stack pointer (args > 4 & local function variables)
\$fp	30	Frame pointer
\$ra	31	Function return address (see jal instruction)

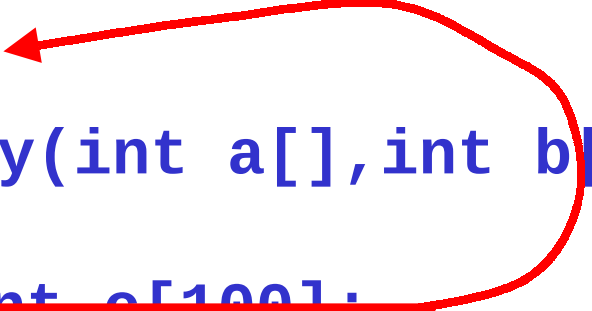
Review: Program memory layout



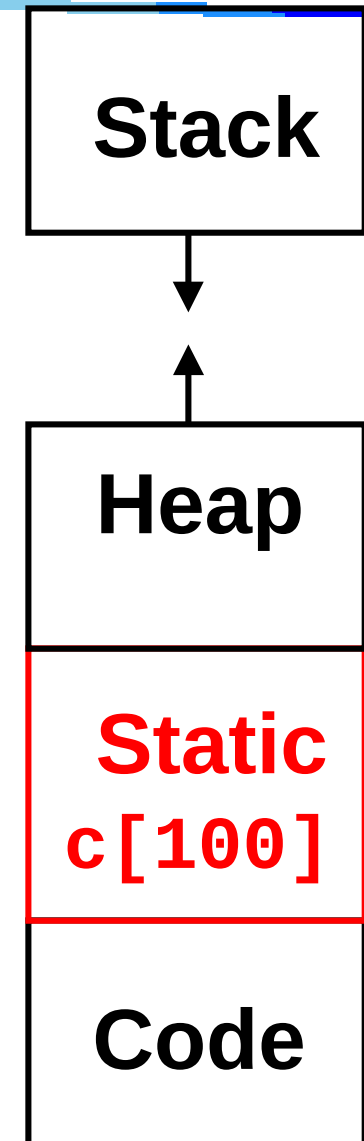
Alternate Static allocation (scope: public to everyone)

- 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

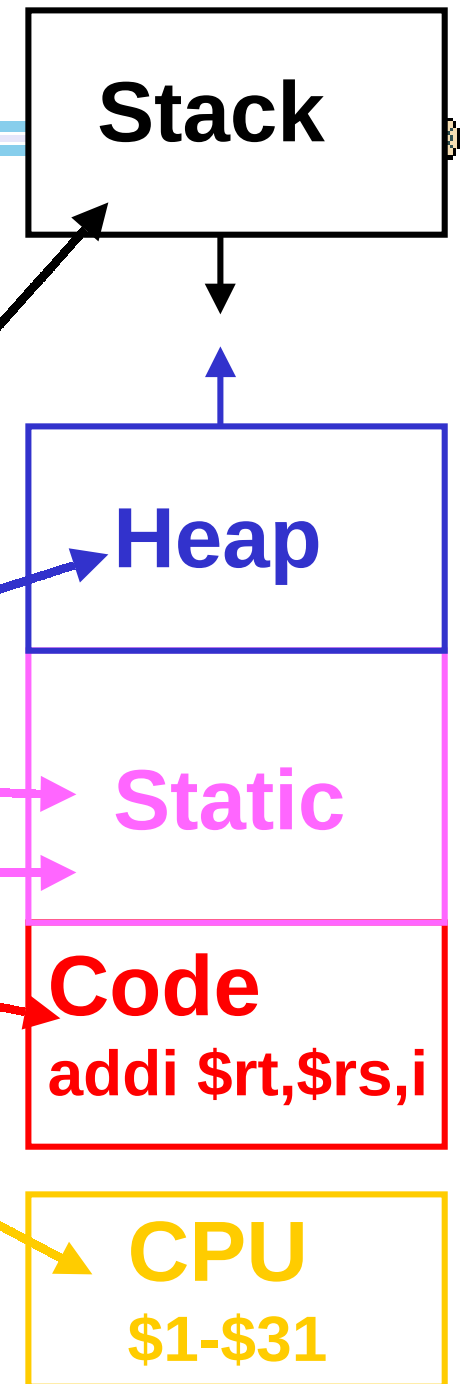


Review: memory allocation model



```
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)	• MIPS assembler
char a;	.data
char b[] = "hello\n"	a: .byte 0
char c[6] = { -1, 20, 0xf };	b: .asciiz "hello\n"
short d;	c: .byte -1,20,0xf,0,0,0
int e;	d: .half 0
int f = 0x1f;	e: .word 0
int *g;	f: .word 0x1f
int *****h;	g: .word 0
int *i[5];	h: .word 0
int (*j)(int x, int y);	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
function() {

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      #y2
```

Unaligned structures: performance issue

[updated]

```
struct {  
    char c;  
    int  x, y;  
} t;
```

• MIPS Assembler

```
0x4000  t: .byte 0      #c
```

```
0x4001      .word 0      #x unaligned
```

```
0x4005      .word 0      #y
```

```
register int f;
```

```
f = t.x;
```

```
# $s1=f
```

```
la    $s0,t          # $s1=&t
```

```
lw    $s1,0($s0)     # $s1=cx3x2x1
```

```
sll   $s1,$s1,8       # $s1=x3x2x10
```

```
lbu   $at,4($s0)      # $t1=000x0
```

```
or    $s1,$s1,$at     # $s1=x3x2x1x0
```

Unaligned words increase the code size and time by a factor of 4 at least!

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:
desti-
nation
1st v.last!
(common
error)**

Numbers: Review



- Number Base B $\Rightarrow$ B symbols per digit:

–Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Base 2 (Binary): 0, 1

- Number representation: $d_4d_3d_2d_1d_0$

$$-d_4 \times B^4 + d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$$

$$\begin{aligned} -10010_{\text{ten}} &= 1 \times 10^4 + 0 \times 10^3 + 0 \times 10^2 + 1 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 10000 + 0 \times 1000 + 0 \times 100 + 1 \times 10 + 0 \times 1 \\ &= 10000 + 0 + 0 + 10 + 0 \\ &= 10010_{\text{ten}} \end{aligned}$$

$$\begin{aligned} -10010_{\text{two}} &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 16_{\text{ten}} + 0_{\text{ten}} + 0_{\text{ten}} + 2_{\text{ten}} + 0_{\text{ten}} \\ &= 18_{\text{ten}} \end{aligned}$$

Numbers: Decimal, Binary, Octal, Hex

[updated]

	<u>Dec</u>	<u>Bin</u>	<u>Hex</u>		<u>Dec</u>	<u>Bin</u>	<u>Hex</u>
base 2: Binary							
base 8: Octal	00	000000	00		16	100000	10
base 10: Decimal	01	000001	01		17	100001	11
base 16: Hex	02	000010	02		18	100010	12
<u>Octal (every 3 bits):</u>	03	000011	03		19	100011	13
= 1001111101	04	000100	04		20	100100	14
= 1, 001, 111, 101	05	000101	05		21	100101	15
= 1,1,1,7,5	06	000110	06		22	100110	16
= 011175 (in C/C++)	07	000111	07		23	100111	17
<u>Hex (every 4 bits):</u>	08	001000	08		24	101000	18
= 1001111101	09	001001	09		25	101001	19
= 10, 0111, 1101	10	001010	0a		26	101010	1a
= 2, 7, d	11	001011	0b		27	101011	1b
= 0x27d (in C/C++)	12	001100	0c		28	101100	1c
<u>Decimal:</u>	13	001101	0d		29	101101	1d
= 1001111101 = 0x27d	14	001110	0e		30	101110	1e
= $2 \cdot 16^2 + 7 \cdot 16^1 + 13 \cdot 16^0$	15	001111	0f		31	101111	1f
= $2 \cdot 256 + 7 \cdot 16 + 13 \cdot 1$							
= 637 (in C/C++)							

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

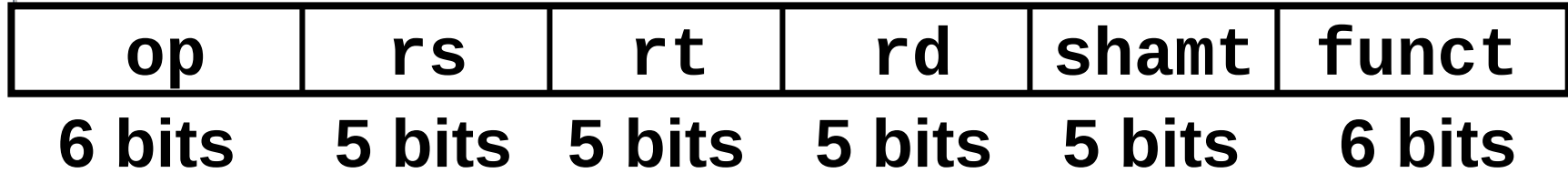


–Fields have names:

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

- 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

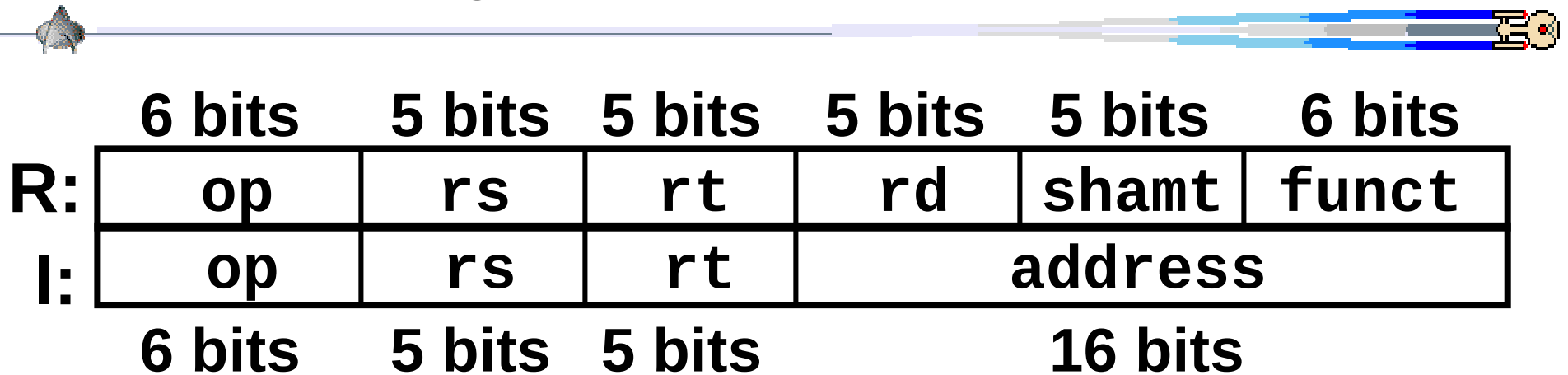


- What if want longer fields? e.g,
 - 5 bits $\Rightarrow$ address of 2^5 or 32 $\Rightarrow$ too small
 - But want all instructions same length!
- Principle: Good design demands good compromises
 - Add 2nd format with larger address



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

Notes about Register and Imm. Formats



- 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`

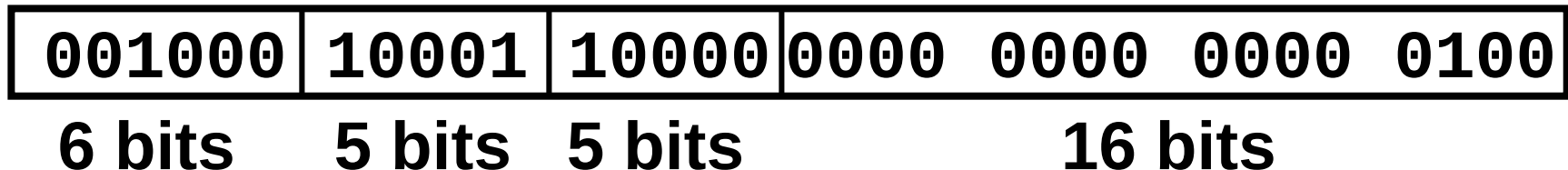
- **Format:**



- **Decimal representation:**



- **Binary representation:**



MIPS instructions



ALU

alu \$rd,\$rs,\$rt

\$rd = \$rs <alu> \$rt

ALUi

alui \$rd,\$rs,const16

\$rd = \$rs <alu> const16

**Data
Transfer**

lw \$rt,byteoffset(\$rs)

\$rt = Mem[\$rs + offset]

sw \$rt,byteoffset(\$rs)

Mem[\$rs + offset] = \$rt

lw, sw: Memory access must on word aligned: 0,4,8,c,10,14,18,1c,...

lh, lhu, sh: Memory access must on halfword aligned: 0,2,4,6,8,a,c,e,10,12, ...

lb, lbu, sb: Memory access is already on a byte address: 0,1,2,3,4,5,6,8, ...

Branch

beq \$rs,\$rt,wordoffset

\$pc = (\$rd == \$rs)? (pc+4+offset<<2):(pc+4);

All instructions occur on word aligned (i.e. 0,4,8, ...) don't need last 2 bits.

Jump

j wordaddress

pc = wordaddress << 2

MIPS fixed sized instruction formats



R - Format

op	rs	rt	rd	shamt	func
----	----	----	----	-------	------

ALU

alu \$rd,\$rs,\$rt

I - Format

op	rs	rt	const16 or byteoffset or wordoffset
----	----	----	---

ALUi

alui \$rt,\$rs,const16

Data
Transfer

lw \$rt,byteoffset(\$rs)
sw \$rt,byteoffset(\$rs)

Branch

beq \$rs,\$rt,wordoffset

J - Format

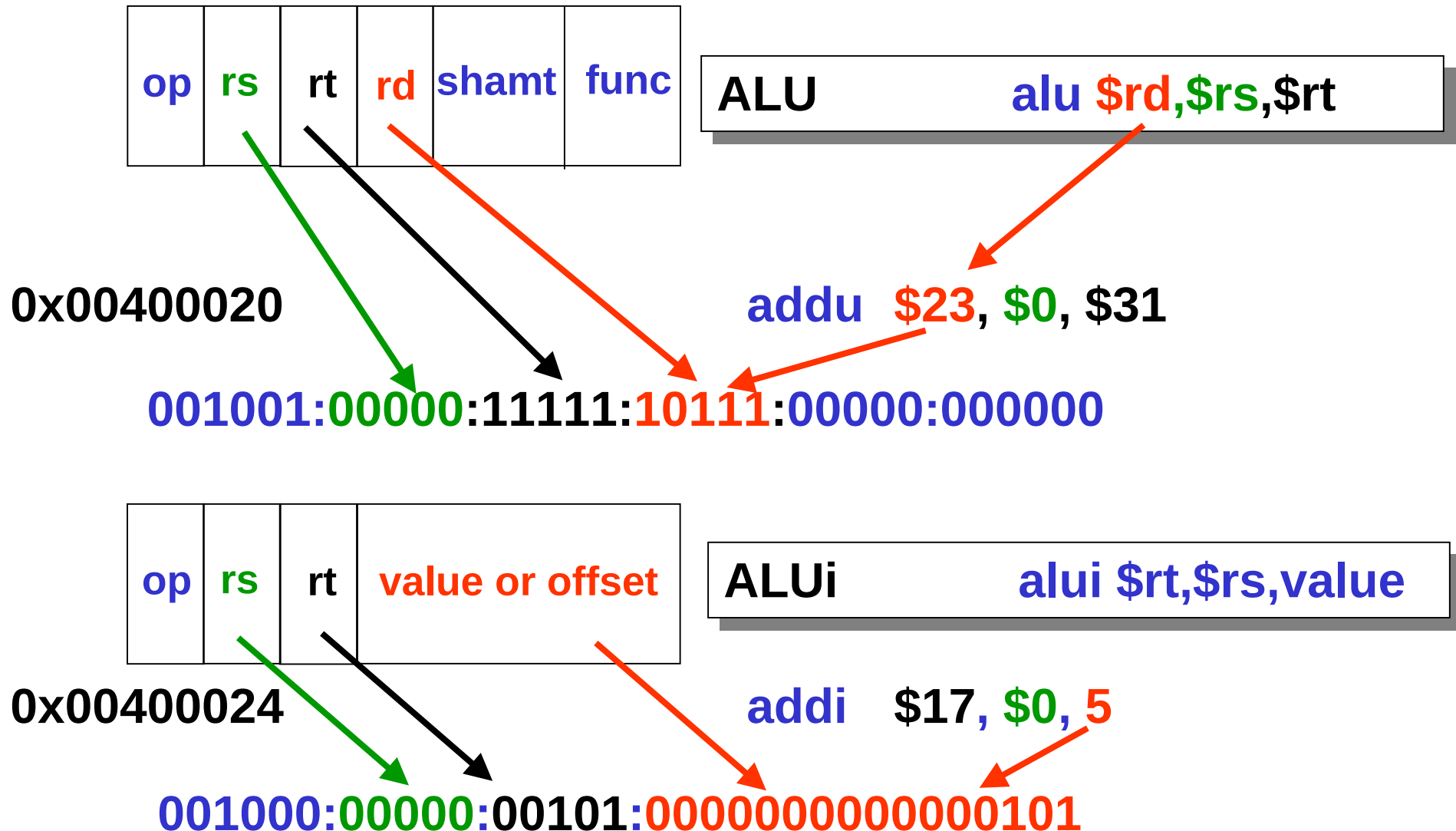
op	absolute address
----	------------------

Jump

j wordoffset

Assembling Instructions

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



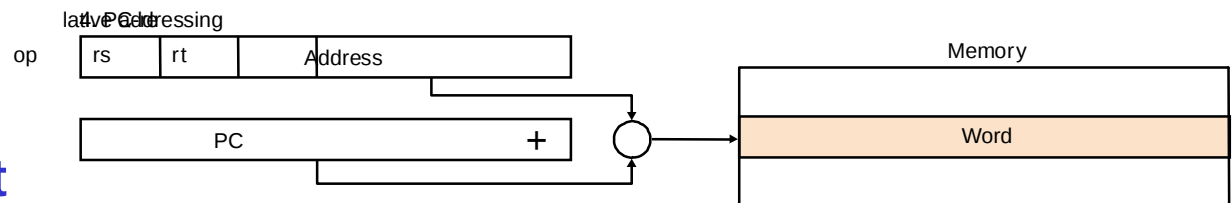
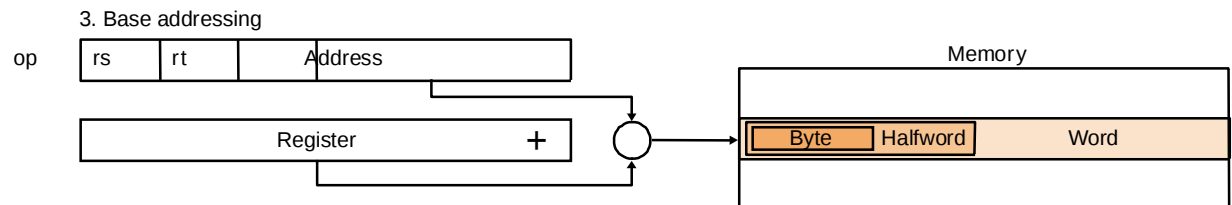
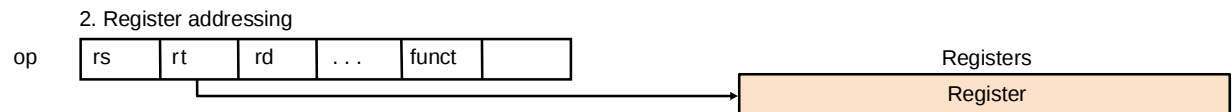
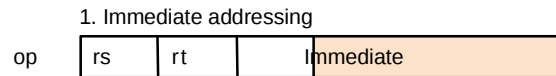
MIPS instruction formats



Arithmetic

`addi $rt, $rs, const16`

`add $rd, $rs, $rt`



Data Transfer

`lw $rt, byteoffset($rs)`

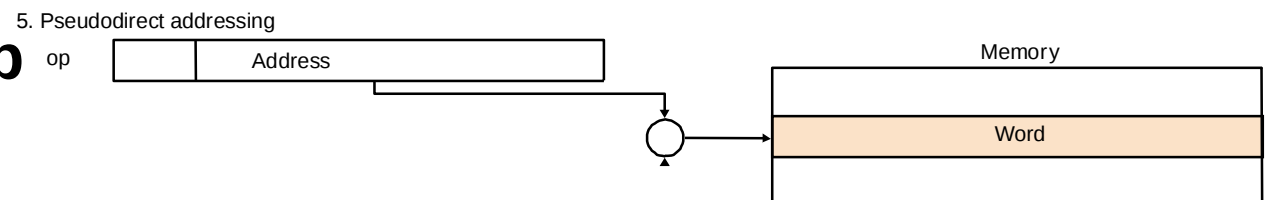
`sw $rt, byteoffset($rs)`

Conditional branch

`beq $rs, $rt, wordoffset`

Unconditional jump

`j wordaddress`



C function to MIPS Assembly Language



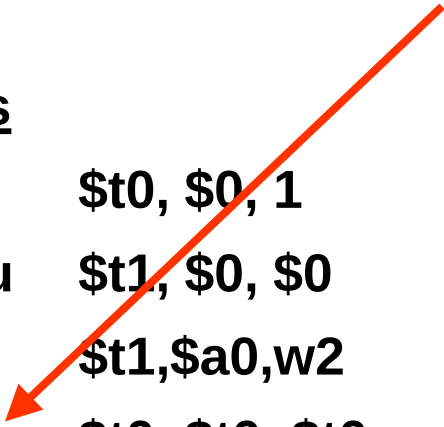
```
int power_2(int y) { /* compute  $x=2^y$ ; */  
    register int x, i; x=1; i=0; while(i<y) { x=x*2; i=i+1; }  
    return x;  
}
```

Exit condition of a while loop is
if ($i \geq y$) then goto w2

Assembler .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 # addu \$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 \$2	\$a0 \$4	\$t0 \$8	\$t1 \$9	\$ra \$31	
00400020	?	0	?	?	700018	<code>addi \$t0, \$0, 1</code>
00400024	?	0	1	?	700018	<code>addu \$t1, \$0, \$0</code>
00400028	?	0	1	0	700018	<code>bge \$t1,\$a0,w2</code>
00400038	?	0	1	0	700018	<code>add \$v0,\$0,\$t0</code>
0040003c	1	0	1	0	700018	<code>jr \$ra</code>
00700018	?	0	1	0	700018	...