MIPS Instruction Set Architecture Review

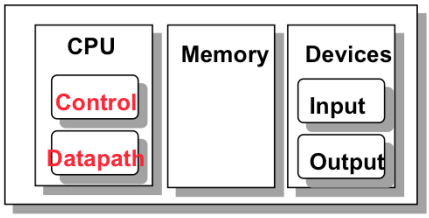
[Adapted from Mary Jane Irwin for
Computer Organization and Design,
Patterson & Hennessy, © 2005, UCB]

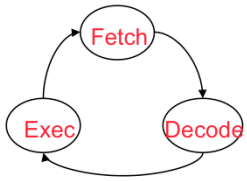
MIPS = Microprocessor without interlocked Pipeline Stages
Primarily used in Embedded Systems such as Windows CE, video game consoles

Other handouts
HW#1
To handout next time

(vonNeumann) Processor Organization

- Control needs to
  1. input instructions from Memory
  2. issue signals to control the information flow between the Datapath components and to control what operations they perform
  3. control instruction sequencing

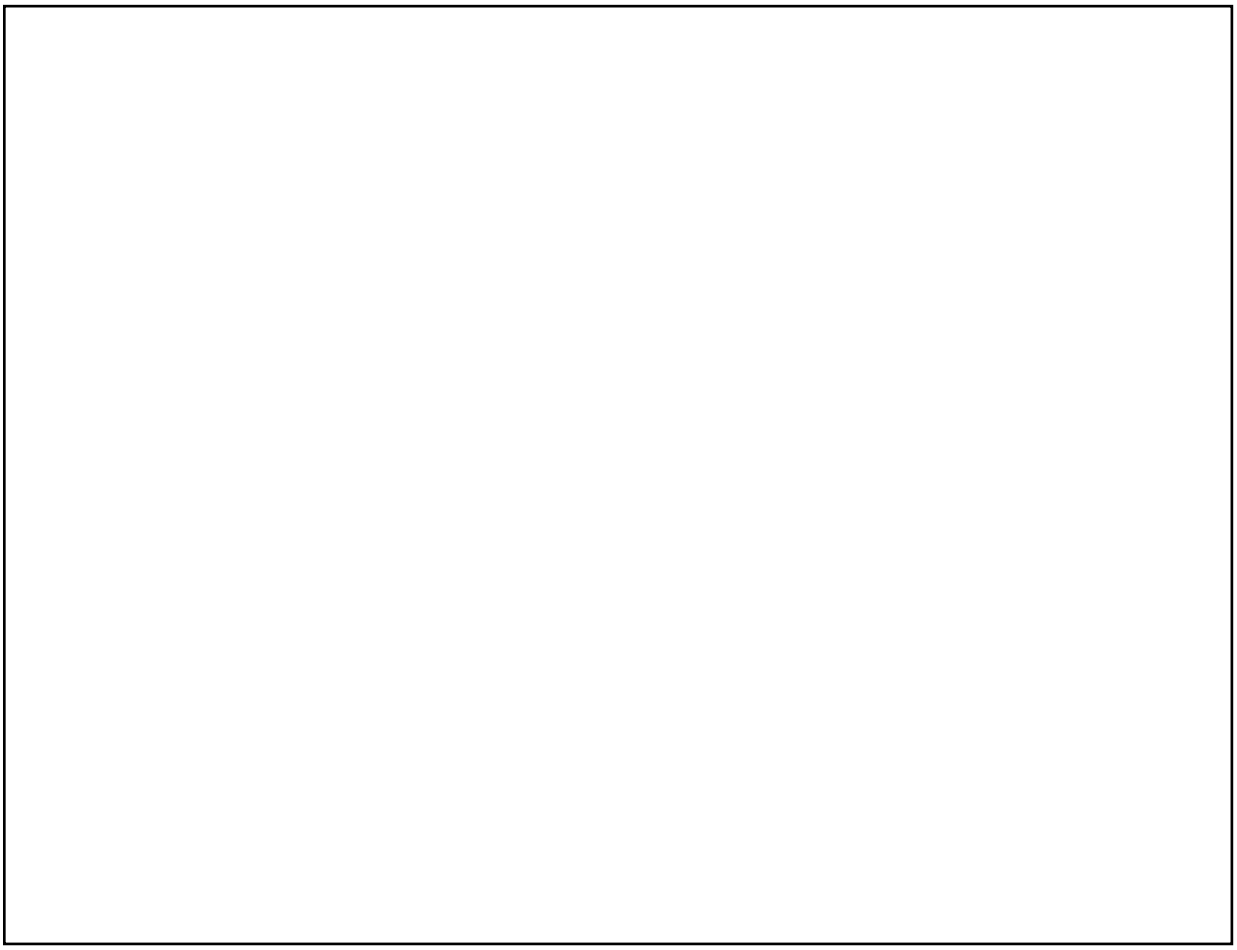




- Datapath needs to have the
  - components – the functional units and storage (e.g., register file) needed to execute instructions
  - interconnects - components connected so that the instructions can be accomplished and so that data can be loaded from and stored to Memory

For a given level of function, however, that system is best in which one can specify things with the most simplicity and straightforwardness. … Simplicity and straightforwardness proceed from conceptual integrity. … Ease of use, then, dictates unity of design, conceptual integrity.

The Mythical Man-Month, Brooks, pg 44



RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, …
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

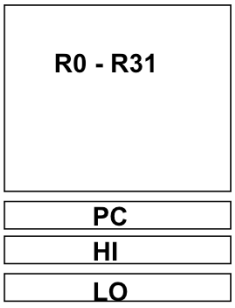
Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

As opposed to CISC – Complicated Instruction Set Architecture (ala the x86)

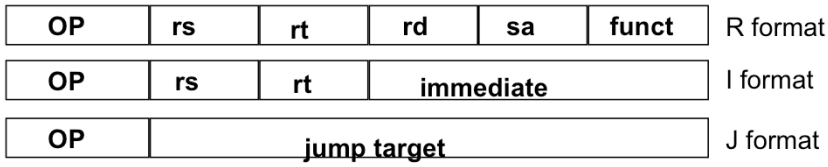
MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers



3 Instruction Formats: all 32 bits wide



R: Register

I: Immediate (immediate ist Wert oder Adresse, 16 bit)

J: Jump

Sa: Shift amout (für Shift befehle)

Funct (Funktionscode für Varianten von OP)

OP 6 bit, r je 5 bit. Sa 5 bit, funct, 6 bit

Memory: load hi/lo into registers (example: used after multiplication)

Special: NOP, break (debugging), syscall…

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $

For lecture

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

Function oriented

For example call to a subroutine via JAL

Then we have registers $a for the arguments passed to the function

Registers $t must be saved before a call to a function if used in the function

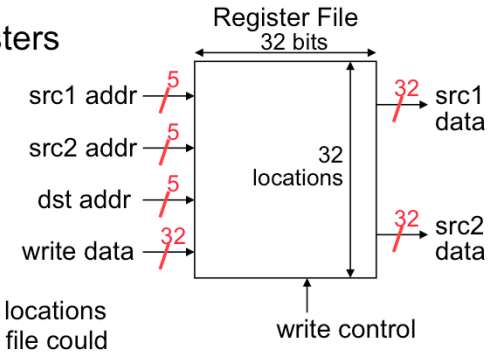
Register $ra stores the return address for continuing execution after function execution is complete

Registers $sp and $fp allocate and deallocate memory local to the function

Registers $v return values computed by the function

MIPS Register File

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port



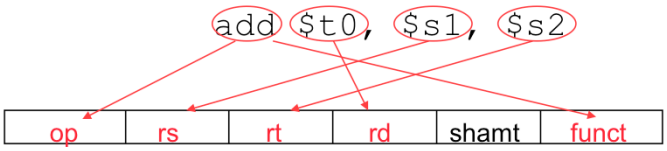
- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed quadratically
  - Easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since registers are accessed with fewer bits than a memory location)

All machines (since 1975) have used general purpose registers

How many bits wide are each of the lines going into/out of the register file?

Machine Language - Add Instruction

- Instructions, like registers and words of data, are 32 bits long
- Arithmetic Instruction Format (R format):



op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

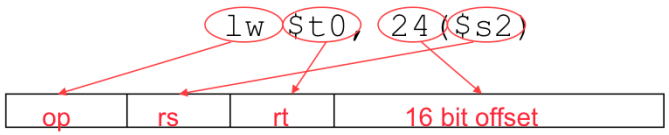
```
lw    $t0, 4($s3)   #load word from memory

sw    $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

What is the memory address of the given load and store instructions if $s3 contains the address 24?

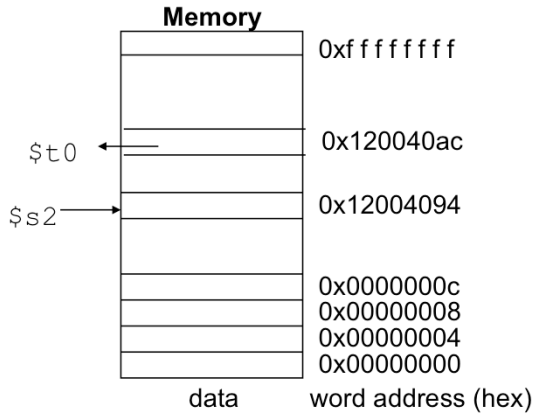
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



24_{10} + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



destination address no longer in the rd field - now in the rt field

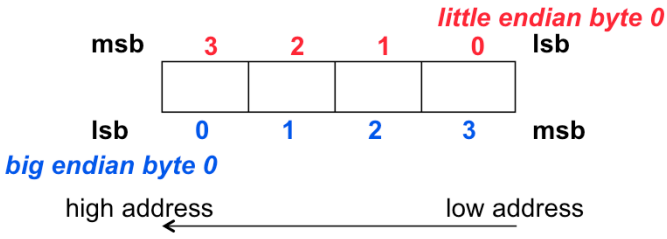
offset limited to 16 bits - so can't get to every location in memory (with a fixed base address)

Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha



Talk about the reasons (performance) for the word/byte alignment restriction

When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (e.g. 4 byte chunks on a 32-bit system) or larger.

Data alignment means putting the data at a memory address equal to some multiple of the word size, which increases the system's performance due

to the way the CPU handles memory. To align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure

and the start of the next, which is data structure padding.

the data starts at address 14 instead of 16, then the computer has to read two or

more 4 byte chunks and do some calculation before the requested data has been

read, or it may generate an alignment fault.

RISC permits only data alignment

CISC allows data misalignment.

Little endian = increasing address go with increasing byte importance

Test

```
#define LITTLE_ENDIAN 0
#define BIG_ENDIAN 1
int endian() {
   int i = 1;
   char *p = (char *)&i;
   if (p[0] == 1)
        return LITTLE_ENDIAN;
   else
        return BIG_ENDIAN;
}
```


Talk about the reasons (performance) for the word/byte alignment restriction

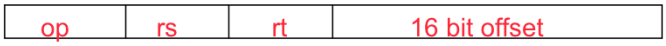
Bei RISC nur Data Alignment erlaubt

Bei CICS auch Data Misalignment. Viel Verschieben Notwendig. Dafür auch Busbreite möglich, die zu klein ist.

Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lbu   $t0, 1($s3)   #load byte from memory
sb    $t0, 6($s3)   #store byte to  memory
```



- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

load byte takes the contents of the byte at the memory address specified, zero-extends it, and loads it into the register

It leaves the other bits in the memory word intact.

Store lässt Speicher intakt, d.h. keine 0 Erweiterung.

Byte Befehle wichtig für Zeichenketten (strings).

Lb statt lbu macht signed extension!

Our First Example

- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
   muli $2, $5, 4
   add $2, $4, $2
   lw $14, 0($2)
   lw $15, 4($2)
   sw $15, 0($2)
   sw $14, 4($2)
   jr $31
```


Argumente werden in $4-$7 uebergeben!

$4 v

$5 k

$14,$15 temporaries

MIPS Control Flow Instructions

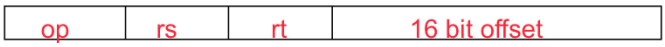
- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

  - Ex:

```
        if (i==j) h = i + j;

        bne $s0, $s1, Lbl1
        add $s3, $s0, $s1
Lbl1:   ...
```

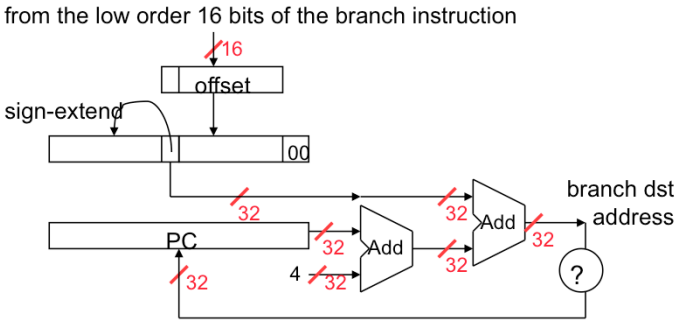
- Instruction Format (I format):



- How is the branch destination address specified?

Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to -2^{15} to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



Note that two low order 0's are concatenated to the 16 bit field giving an eighteen bit address

0111111111111111 00 = 2**15 –1 words (2**17 – 1 bytes)

More Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```

- Instruction format (R format):

More Branch Instructions - Continued

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt  $at, $s1, $s2      #$at set to 1 if
bne  $at, $zero, Label  #  $s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - It's why the assembler needs a reserved register (`$at`)

For lecture

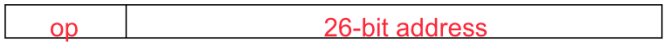
```
blt     $s1,$s2 Label =

slt     $at,$s1,$s2          #at gets 1 if $s1 < $s2
bne     $at,$zero,Label
```

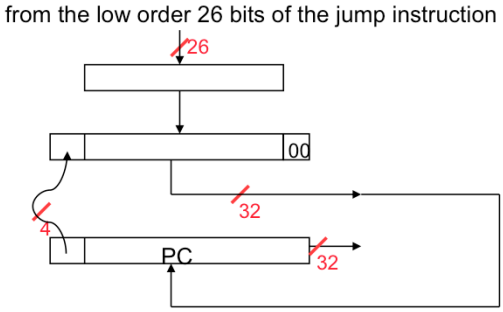
Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
j   label        #go to label
```

- Instruction Format (J Format):





If-then-else code compilation

Aside: Branching Far Away

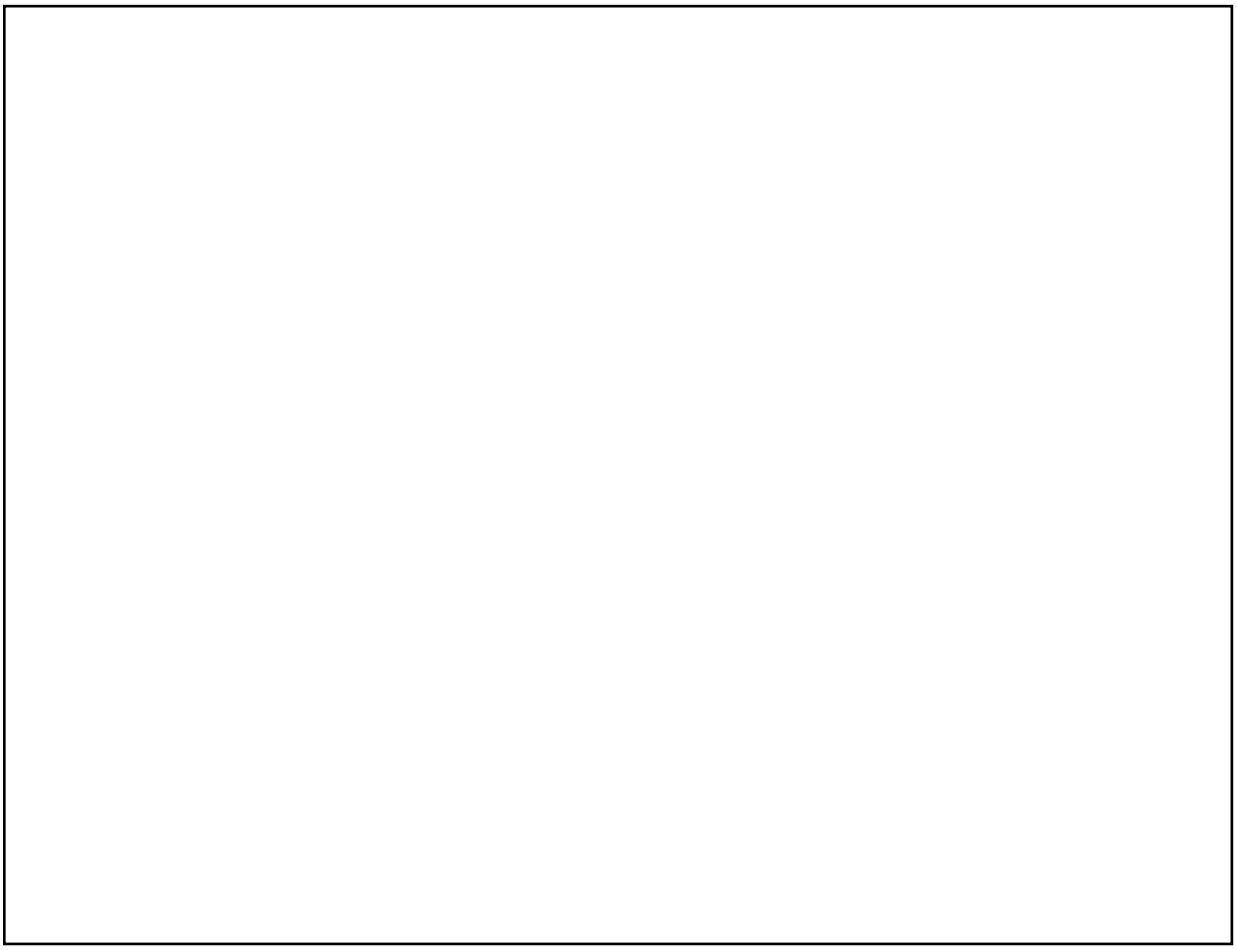
- What if the branch destination is further away than can be captured in 16 bits?

- The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
        beq  $s0, $s1, L1
```

becomes

```
        bne  $s0, $s1, L2
        j    L1
    L2:
```

Instructions for Accessing Procedures

- MIPS procedure call instruction:

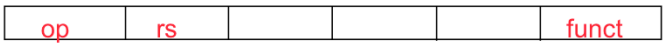
```
jal   ProcedureAddress     #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

- Then can do procedure return with a

```
jr    $ra                  #return
```

- Instruction format (R format):



jr jumps to the address stored in the register – in this case $ra – which is just what we want.

Example: Procedure

```
int leaf_ex
    (int g, int h, int i, int j)
{
  int f;
  f = (g+h) - (i+j);
  return f;
}
```

Hier werden alle Register gerettet. Konvention ist, dass nur die $s0-$s7 gerettet werden müssen.

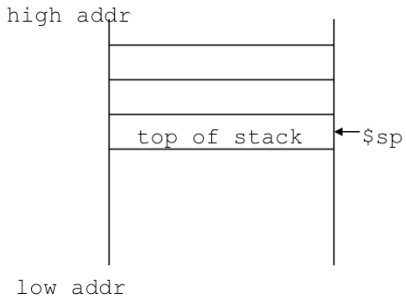
Die Argumente werden in $a0-$a3 übergeben,
Resultate in $v0,$v1 zurückgegeben.

```
leaf_ex:
  addi $sp,$sp,-12
  sw $t1, 8($sp)
  sw $t0, 4($sp)
  sw $s0, 0($sp)
  add $t0,$a0,$a1
  add $t1,$a2,$a3
  sub $s0,$t0,$t1
  add $v0,$s0,$zero
  lw $s0, 0($sp)
  lw $t0, 4($sp)
  lw $t1, 8($sp)
  addi $sp,$sp,12
  jr $ra
```


All registers are saved

Aside: Spilling Registers

- What if the callee needs more registers? What if the procedure is recursive?
  - uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)



- One of the general registers, $sp, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push

    $sp = $sp – 4
    data on stack at new $sp
  - remove data from the stack – pop

    data from stack at $sp
    $sp = $sp + 4

Spill: ueberlaufen

Note that temporary registers $t0 through $t9 can also be used as by MIPS convention they are not preserved by the callee (kept intact) across subroutine boundaries

However, saved registers $s0 through $s7 must be preserved by the calle - I.e., if the callee uses one, it must first save it and then restore it to its old value before returning control to the caller

Example: Recursive Procedure

```
int fact( int n )
{
  if (n < 1)
    return (1);
  else
    return ( n * fact(n-1));
}
```

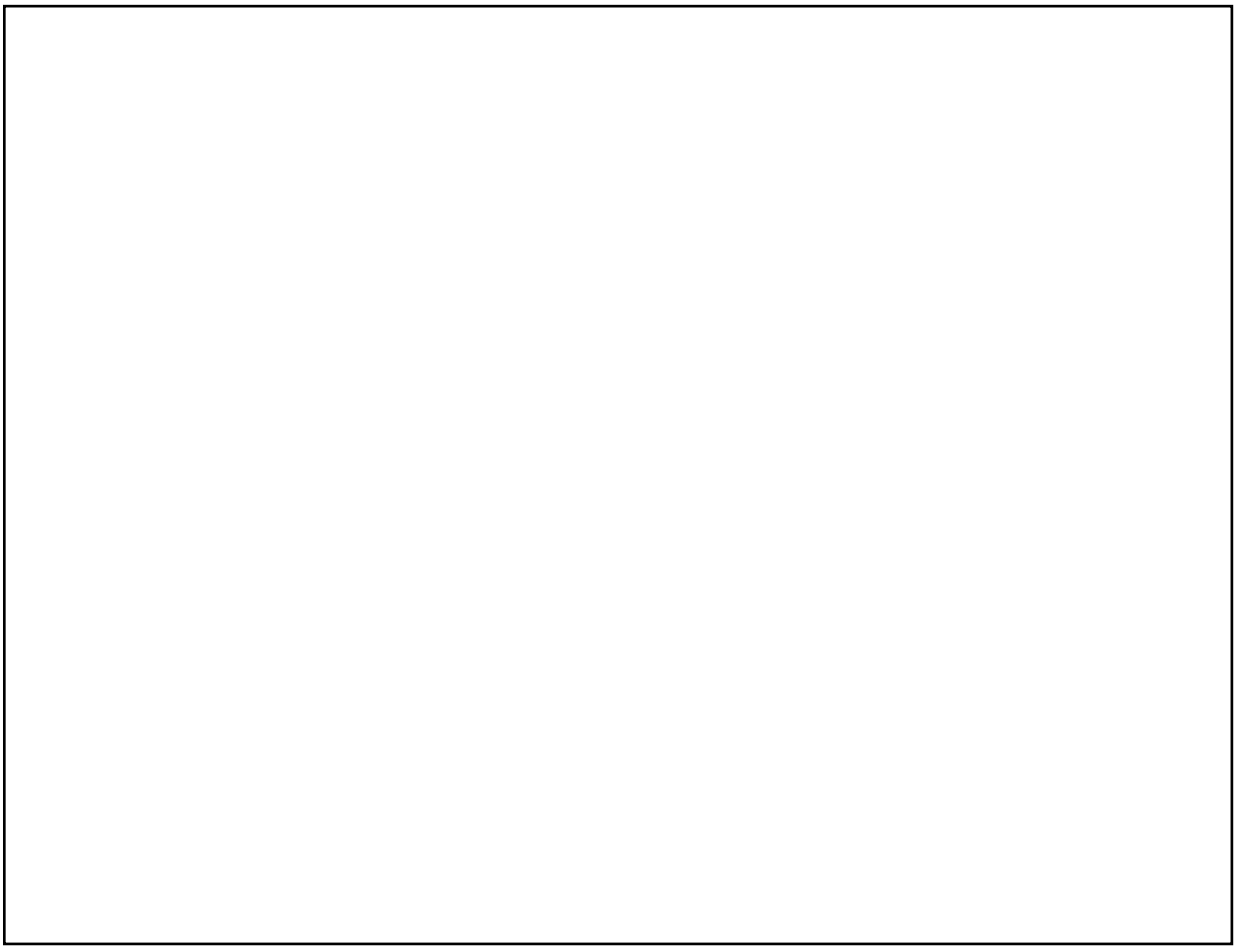
$ra und $a0 müssen auf den Stack gerettet werden, damit sie nach dem Funktionsaufruf wieder verfügbar sind.

```
fact: addi $sp,$sp,-8
      sw $ra, 4($sp)
      sw $a0, 0($sp)
      slti $t0, $a0, 1
      beq $t0,$zero,L1
      addi $v0,$zero,1
      addi $sp,$sp,8
      jr $ra
L1:   addi $a0,$a0,-1
      jal fact
      lw $a0,0($sp)
      lw $ra, 4($sp)
      addi $sp,$sp,8
      mul $v0,$a0,$v0
      jr $ra
```


Save ra and a0 so that they are available after the function call

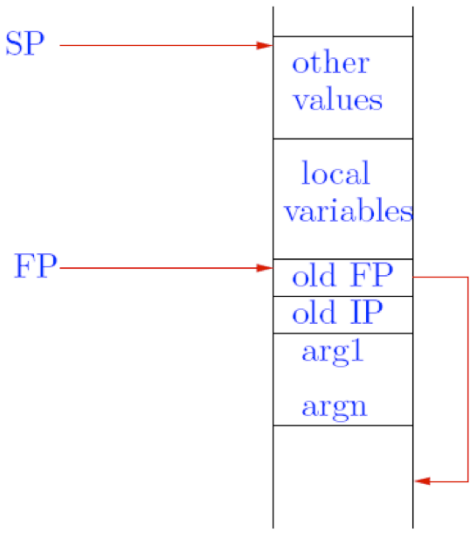
Function calls and stack frames

- Each time a function is called, space must be allocated for the local variables of the function. This region of the stack is called the stack frame for this function call.
- Use a Frame Pointer (FP) to indicate the location of the current frame. This allows easy access to the local variables at runtime.
- On return from a function call, execution must continue from the next instruction after the function call.
- Store the old instruction pointer (PC) in the stack frame.



Examples: Stack

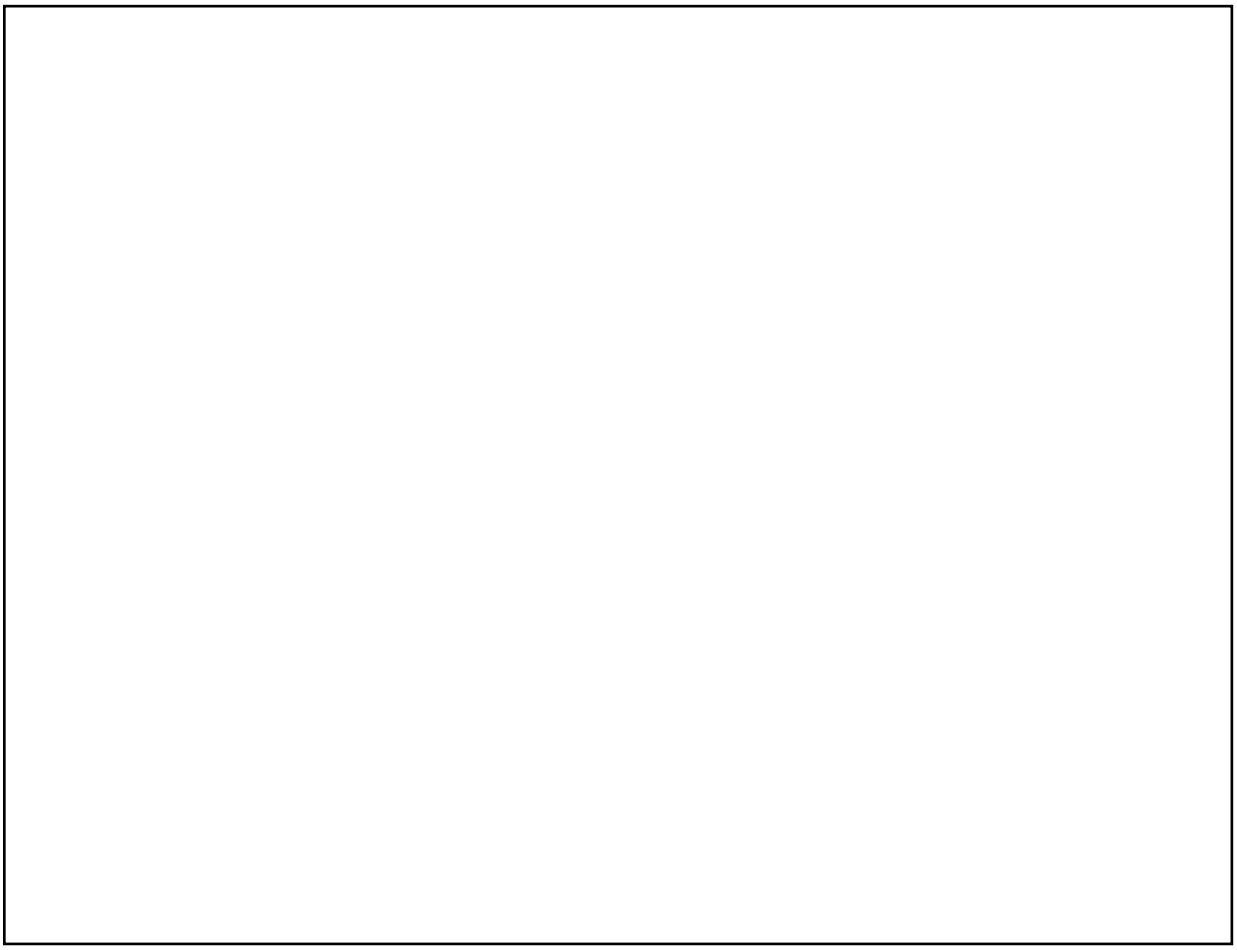
- On return from a function, the current stack frame is popped out and execution continues with the previous stack frame.
- Store the old FP on the stack.



A simple example of function call

```
/* function.c */
void f (int x, int y) {
  int a,b,c;
}

int main () {
  f (10, 20);
}
```

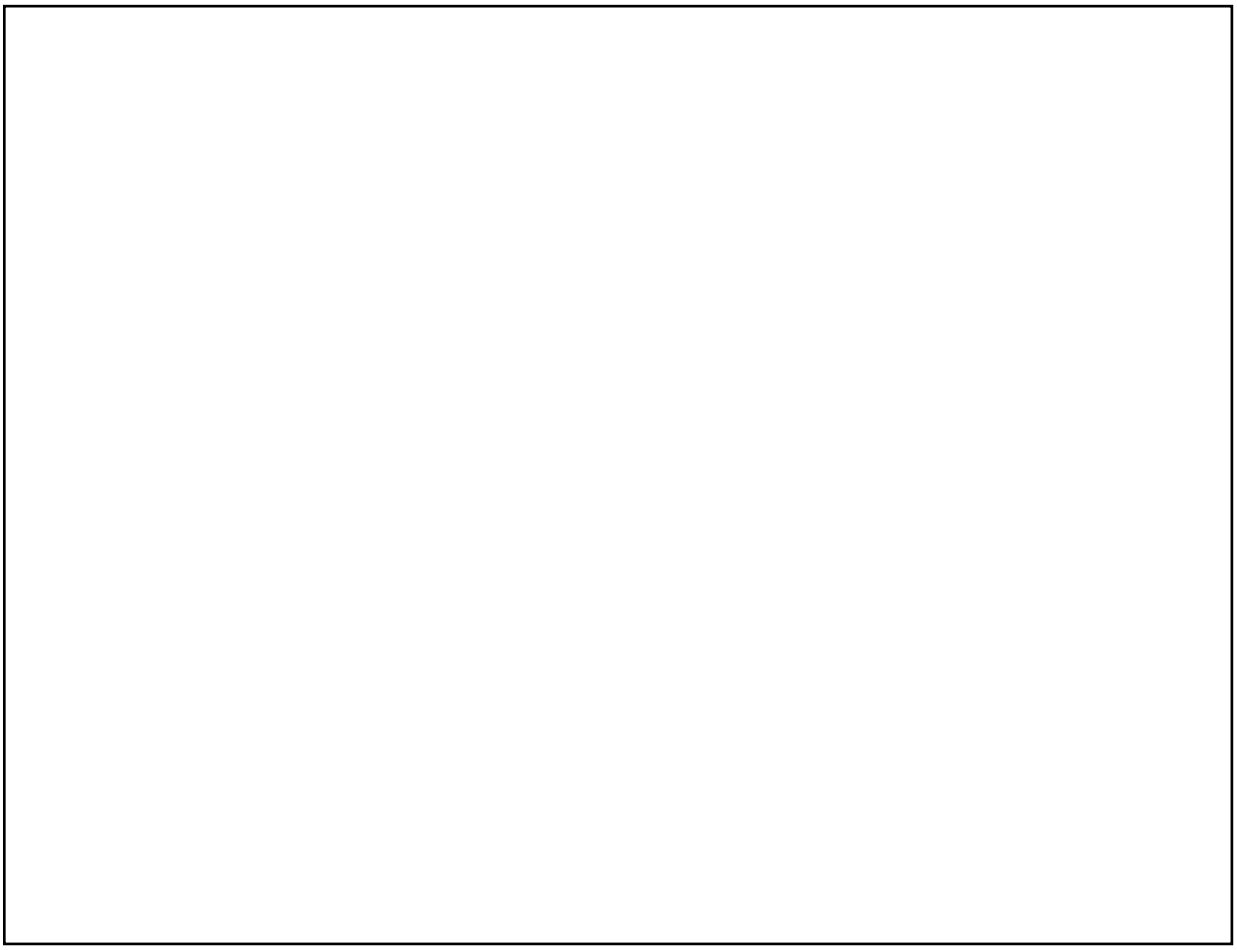
Let's see the compiled code produced.



The caller

```
(gdb) disassemble main
 ...
0x804832f <main+19>: push   $0x14
0x8048331 <main+21>: push   $0xa
0x8048333 <main+23>: call   0x8048314 <f>
 ...
```

The arguments are pushed on to the stack and the function is called.



And the callee

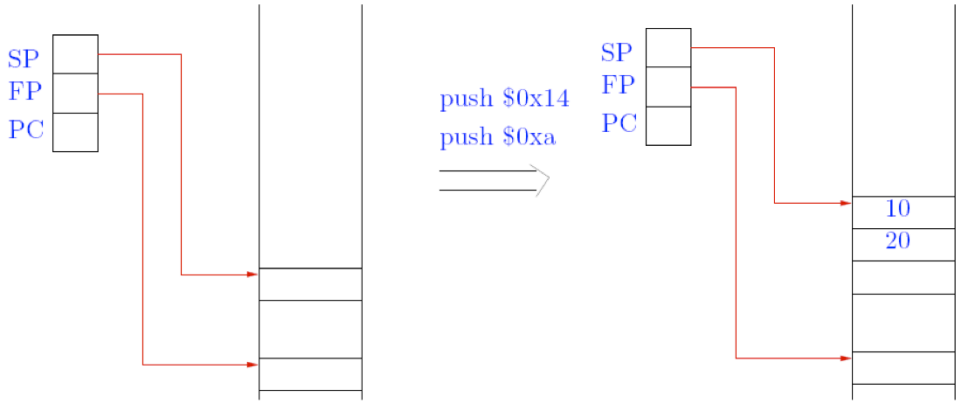
```
0x8048314 <f>:      push   %ebp
0x8048315 <f+1>:    mov    %esp,%ebp
0x8048317 <f+3>:    sub    $0xc,%esp
0x804831a <f+6>:    leave
0x804831b <f+7>:    ret
```

- Save old FP, update FP
- Allocate space for local variables, do computations
- Restore FP, pop saved FP from stack
- Return (restore PC, pop saved PC from stack)

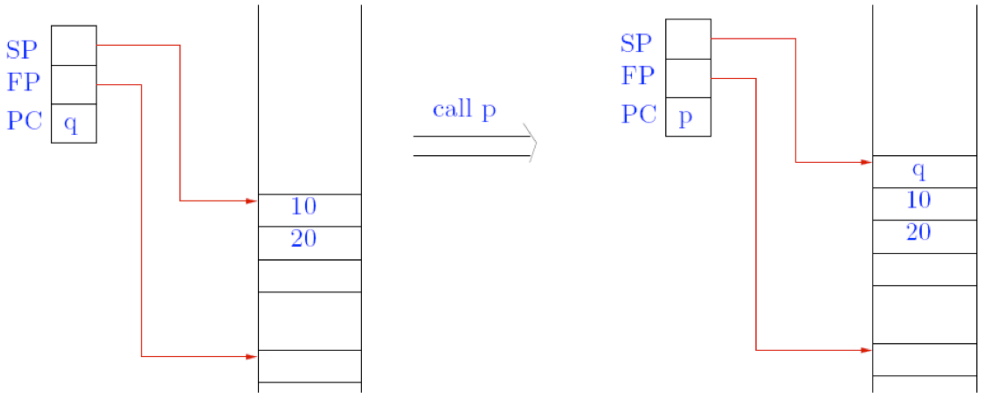
EBP is a stack frame pointer in x86 (CISC)

Intro operations can be substituted by
enter
and exiting operations by
leave

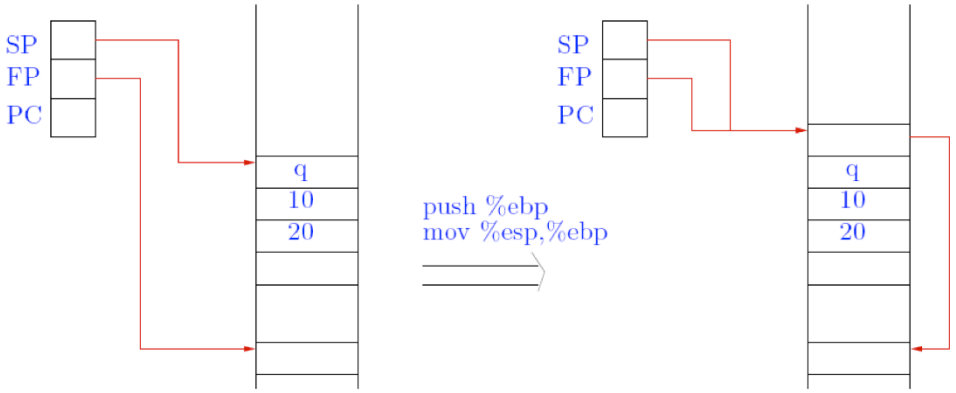
At run time: pushing arguments



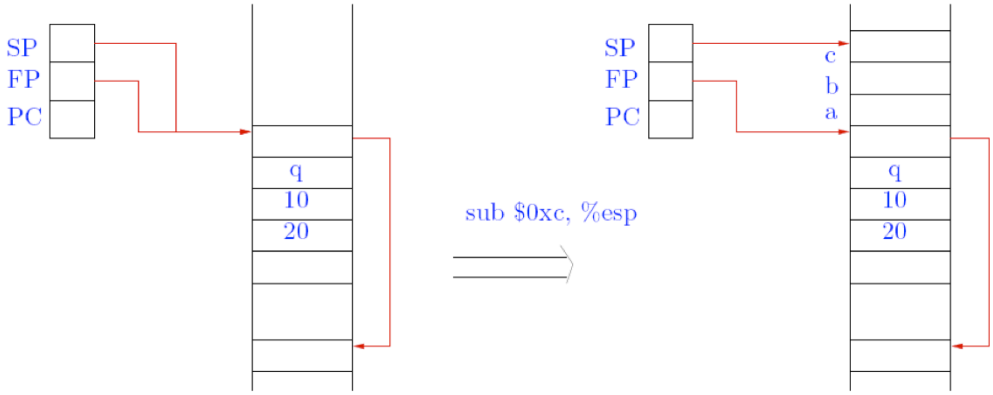
Calling function: saving PC and updating PC



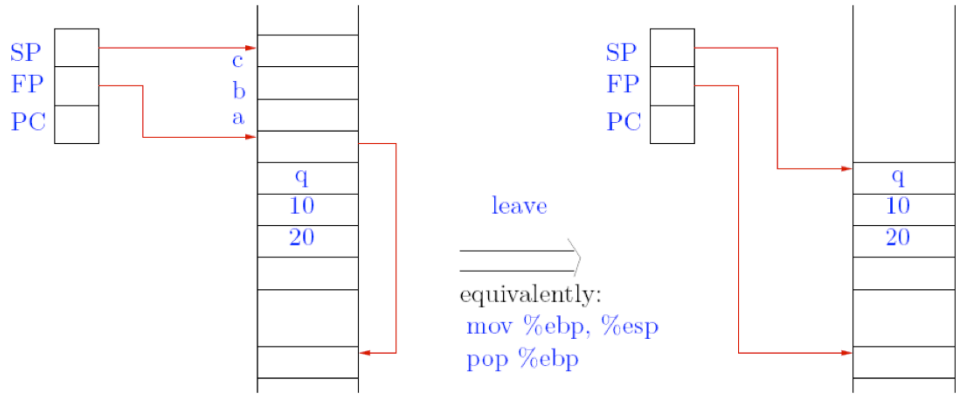
Inside callee: saving FP and updating FP



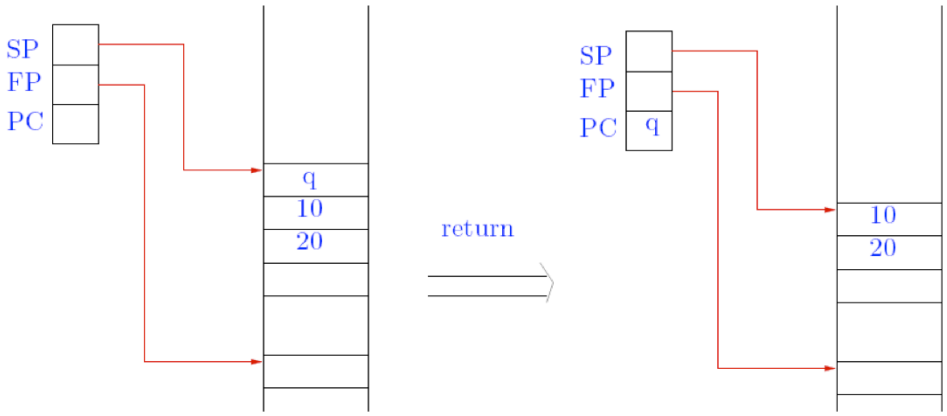
Allocating space for local variables



End of callee: restoring FP and popping saved FP



Returning: restoring PC and popping saved PC



Stack overflow

The return address is stored on the stack.
=> It can also be overwritten to point to arbitrary code!!!

```
void f () {
int a[10];
a[15] += 7;
}
```

```
main () {
  int x = 10;
  f ();
  x = 20;
  printf ("x=%d!\n",x);
}
```

Output: x=10!

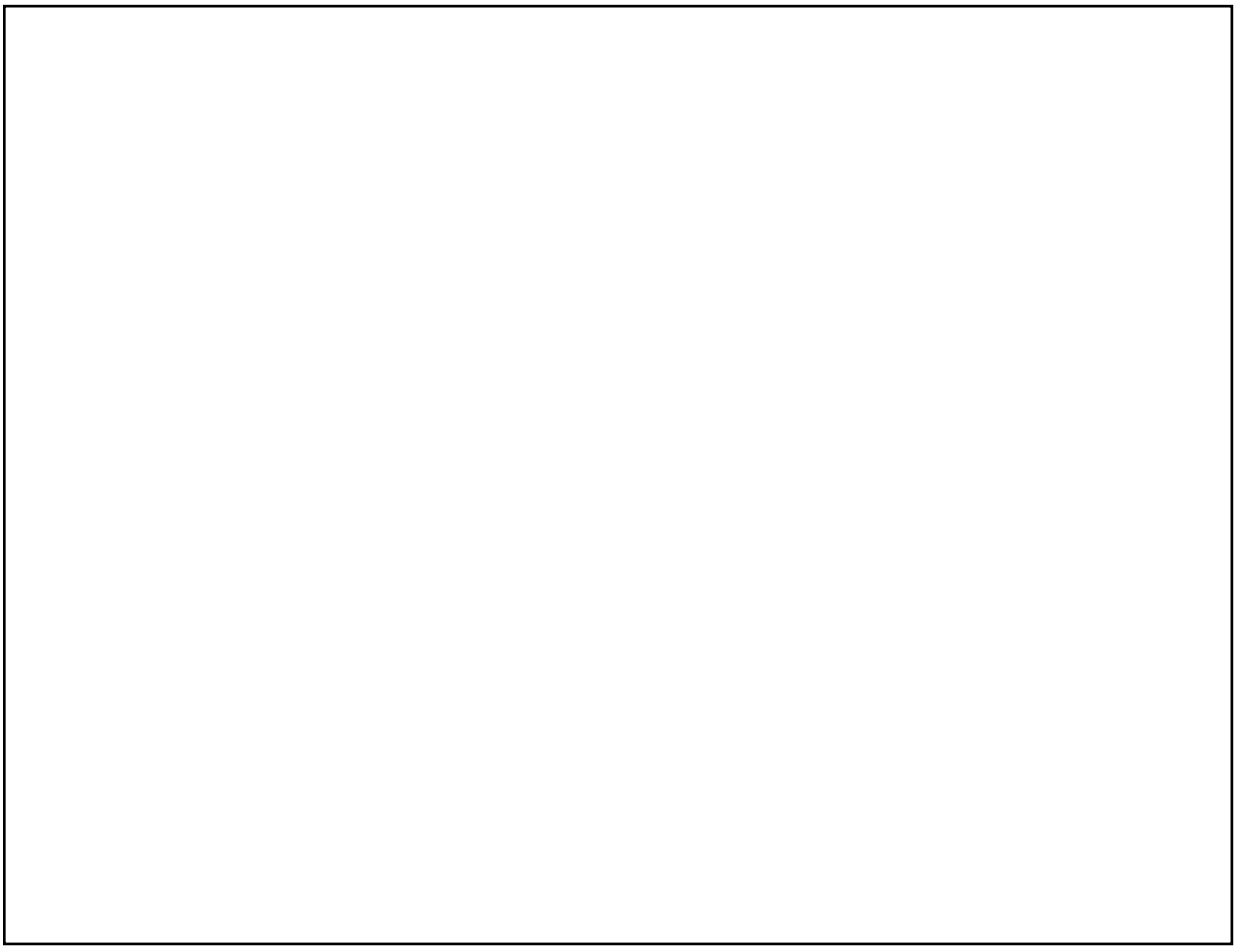
- We have skipped the instruction x = 20; !
- Where is the return address stored (a[15])?
- What should be the new return address (increment by 7)?

Organization of the stack

- Organization of the stack: a[0], . . . , a[9], old FP, old PC
- Hence the return address is at the location a[11].
- Not always!! Compiler optimizations may create blank spaces between array and the following data.
- Look at the compiled code.

```
0x8048344 <f>:      push   %ebp
0x8048345 <f+1>:    mov    %esp,%ebp
0x8048347 <f+3>:    sub    $0x38,%esp
...
```

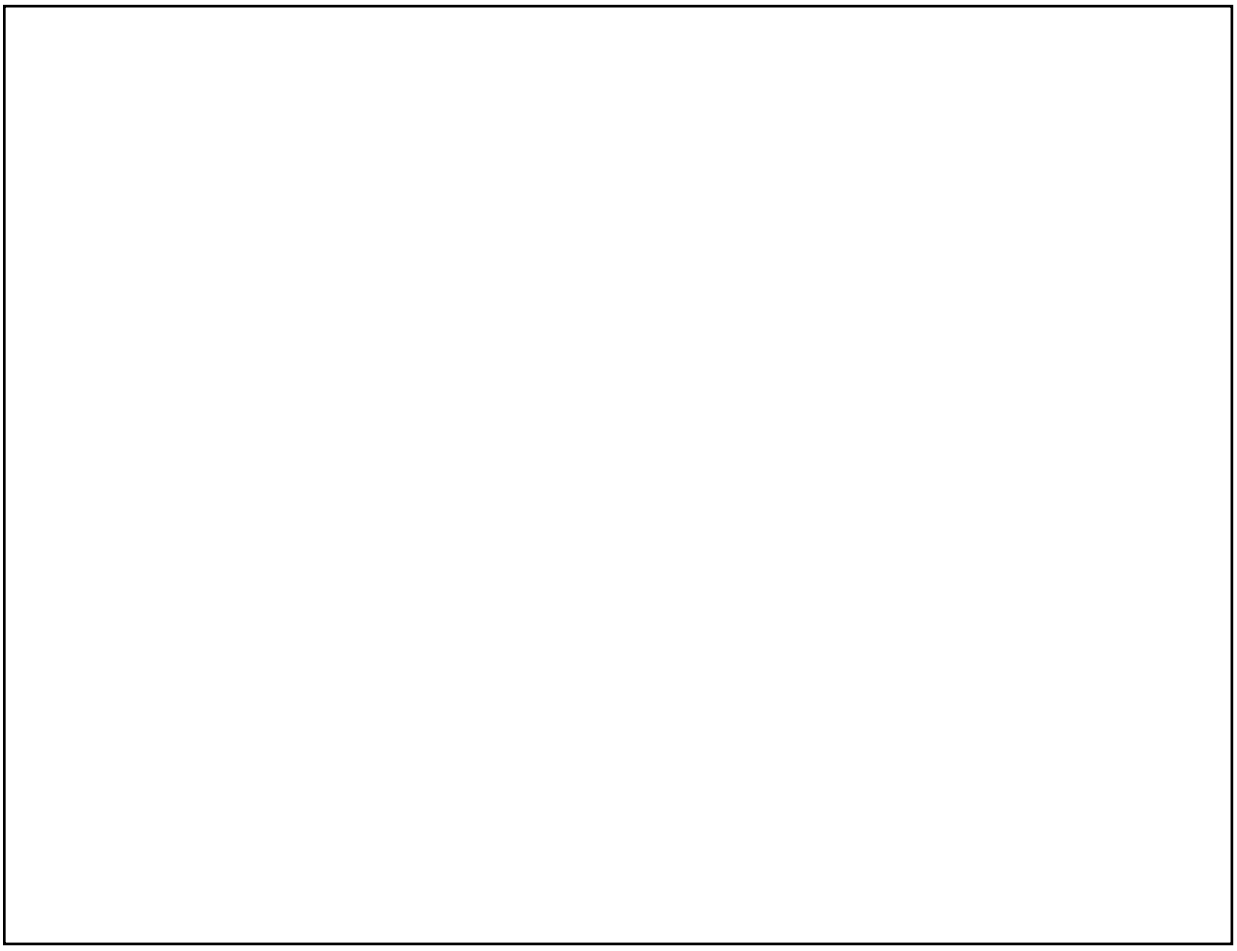
- Space allocated after old FP is 0x38 = 56 = 4*14 bytes.
- Hence return address is at address a[15]



a[15]+=7

```
...
0x8048369 <main+23>: call   0x8048344 <f>
0x804836e <main+28>: movl   $0x14,0 xfffffffc (%ebp)
0x8048375 <main+35>: sub    $0x8,%esp
...
```

- Instruction x = 20; requires 35 - 28 = 7 bytes.
- Hence we put a[15] +=7 in the function f in order to skip execution of this instruction.
- Besides modifying data, we may cause arbitrary code to be executed!

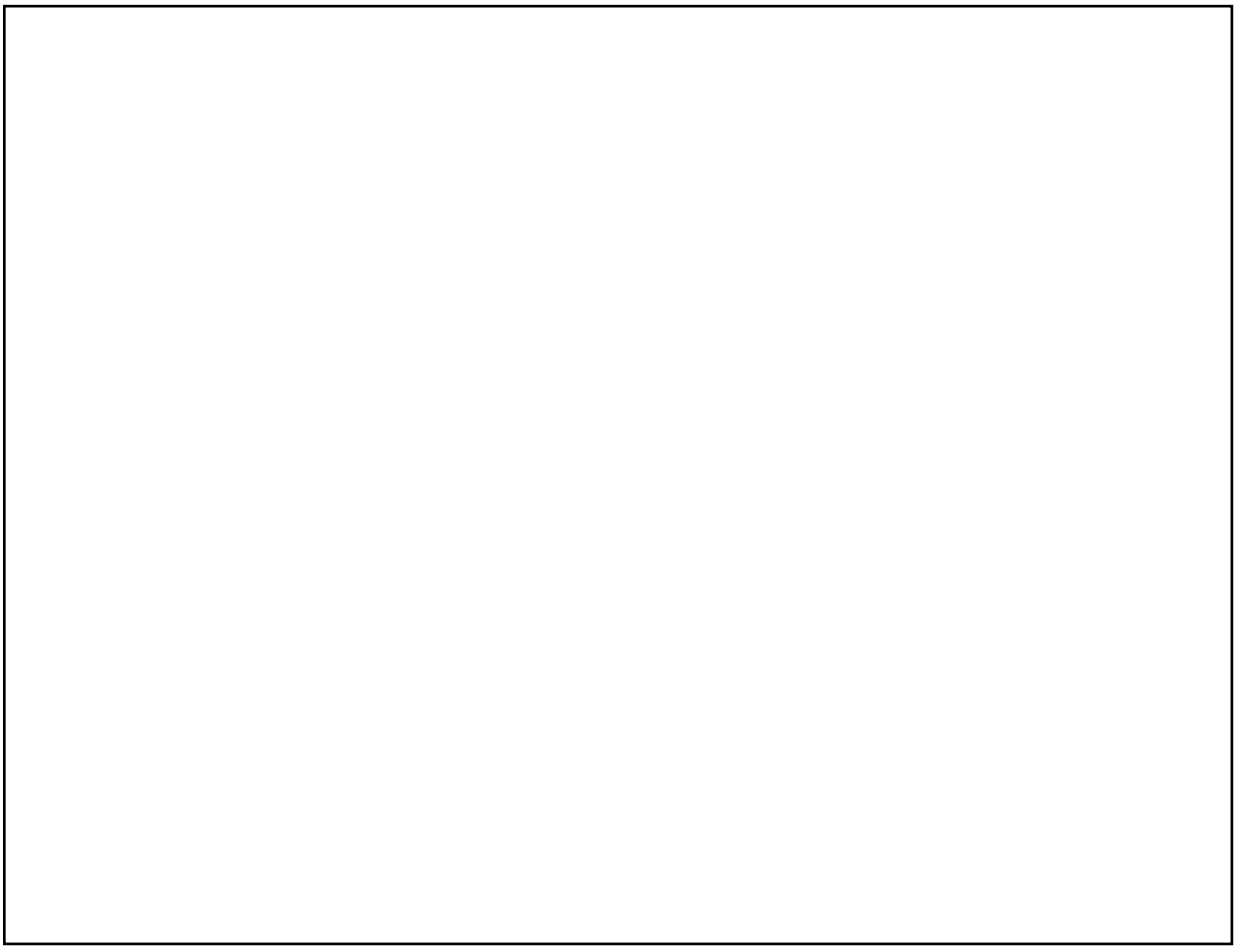


Supplying appropriate inputs

Weaknesses can be exploited by users by supplying appropriate inputs.

```
int main (int argc, char *argv[]) {
  char s[1024];
  strcpy(s,argv[1]);
  ...
}
```

- An appropriate input is given to overwrite the return address,
- At the minimum, the program may abort abruptly.
- An ingenious attacker may get some desired code to be executed (shellcode)
- by providing it as a part of the input string!

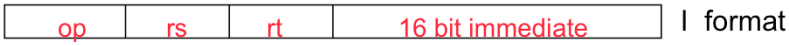


MIPS Immediate Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi  $sp, $sp, 4      #$sp = $sp + 4
slti  $t0, $s2, 15     #$t0 = 1 if $s2<15
```

- Machine format (I format):



I format

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

in gcc 52% of arithmetic operations involve constants – in spice it's 69%

have the students answer why not – speed and limited number of registers

much faster than if loaded from memory

addi and slti do sign extend of immediate operand into the leftmost bits of the destination register (ie., copies the leftmost bit of the 16-bit immediate value into the upper 16 bits)

by contrast, ori and andi loads zero's into the upper 16 bits and so is usually used (instead of the addi) to build 32 bit constants

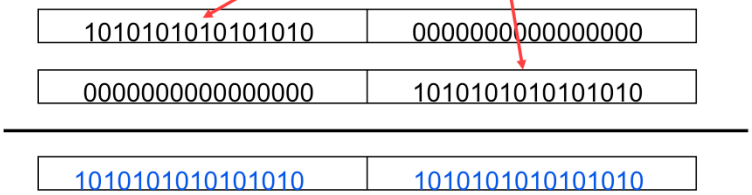
Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

- Then, to get the lower order bits right, use (logical OR)

```
ori $t0, $t0, 1010101010101010
```



For lecture

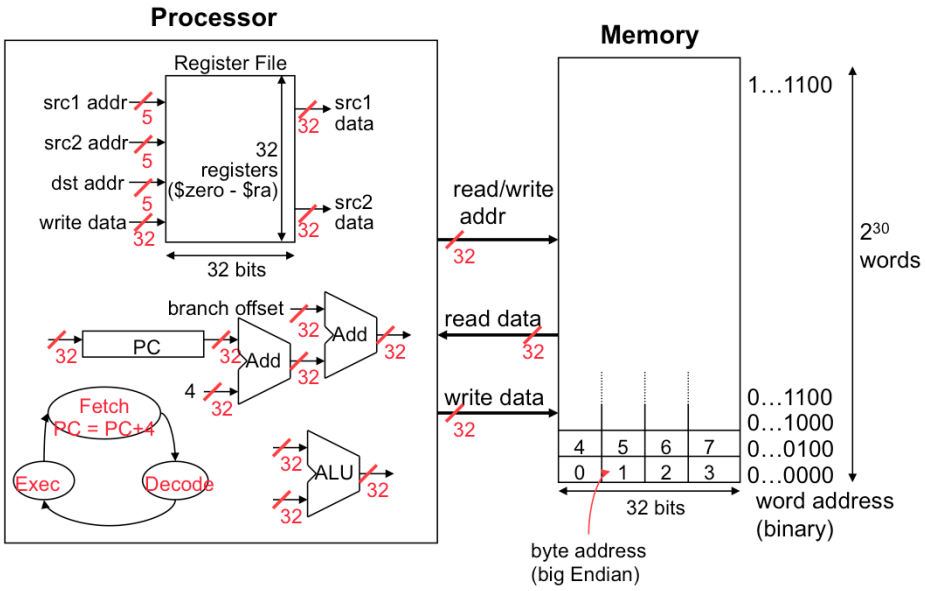
sets the upper 16 bits of a constant in a register, allowing the next instruction to specify the lower 16 bits of the constant

note that lui fills the lower 16 bits of the register with zero's

why can't addi be used as the second instruction for this 32 bit constant?

also note – the assembler lets me specify constants larger than 16 bits in one instruction and then expands the code into two instructions (lui and ori) automatically

MIPS Organization So Far



Big Endian: High to low as memory location increases (reversed)

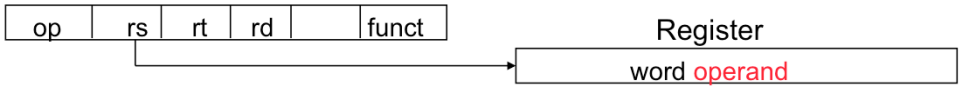
MIPS ISA So Far

Category	Instr	Op Code/ Funct	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer (I format)	load word	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	15	lui $s1, 6	$s1 = 6 * 2^{16}
Cond. Branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

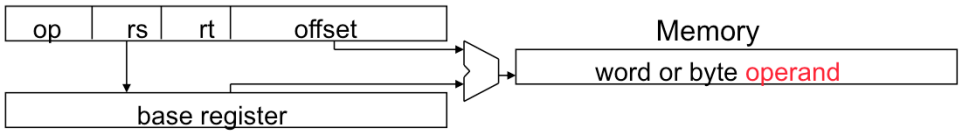
similarity of the binary representation of related instructions simplifies the hardware design

Review of MIPS Operand Addressing Modes

- Register addressing – operand is in a register

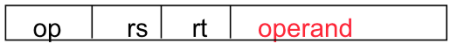


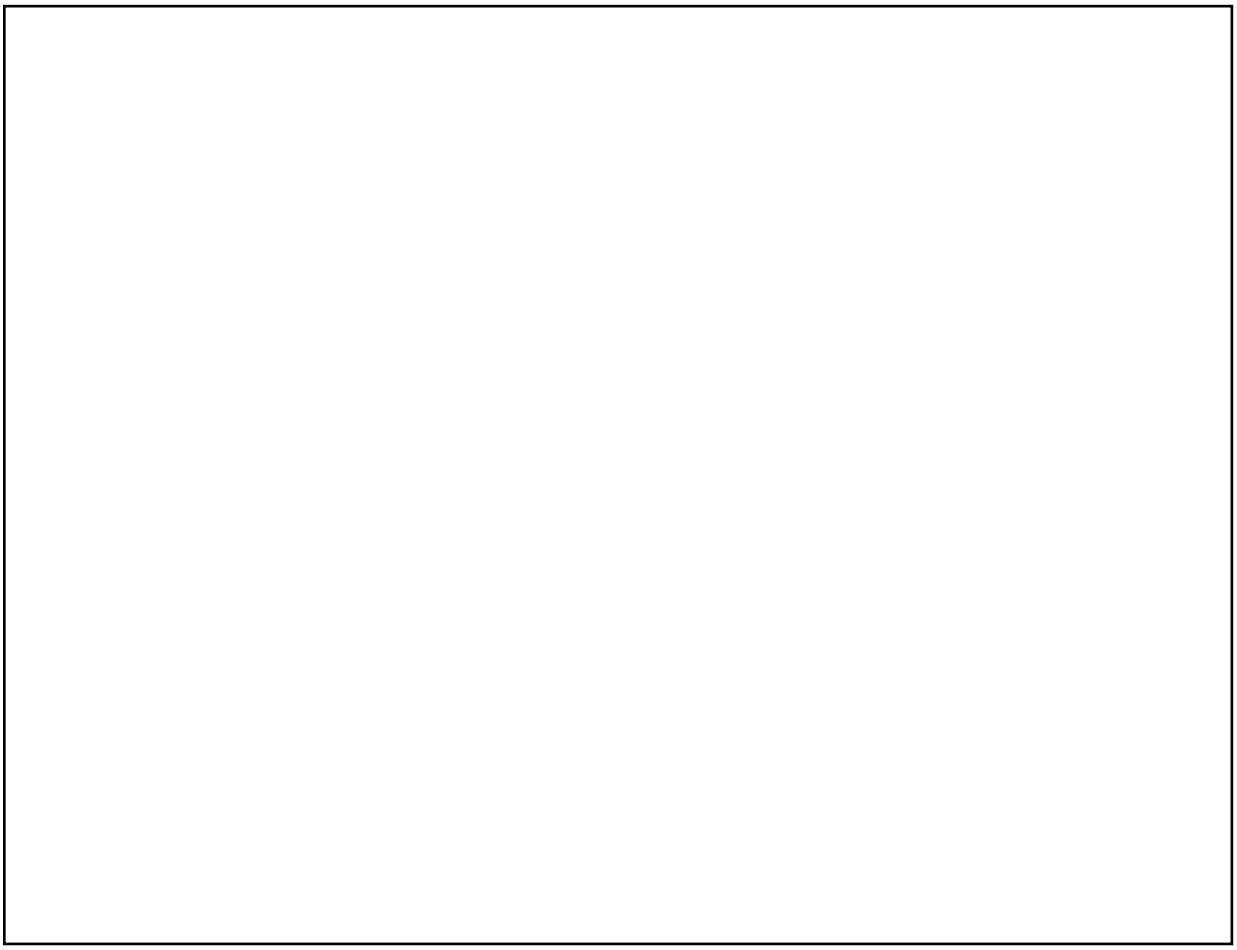
- Base (displacement) addressing – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction



  - Register relative (indirect) with 0($a0)
  - Pseudo-direct with addr($zero)

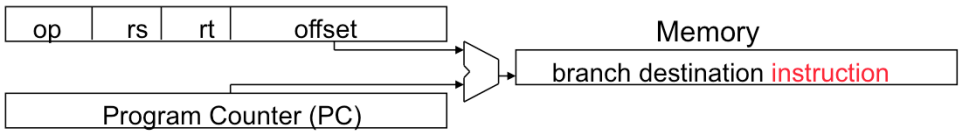
- Immediate addressing – operand is a 16-bit constant contained within the instruction



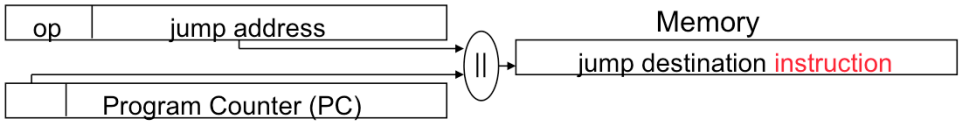


Review of MIPS Instruction Addressing Modes

- PC-relative addressing –instruction address is the sum of the PC and a 16-bit constant contained within the instruction

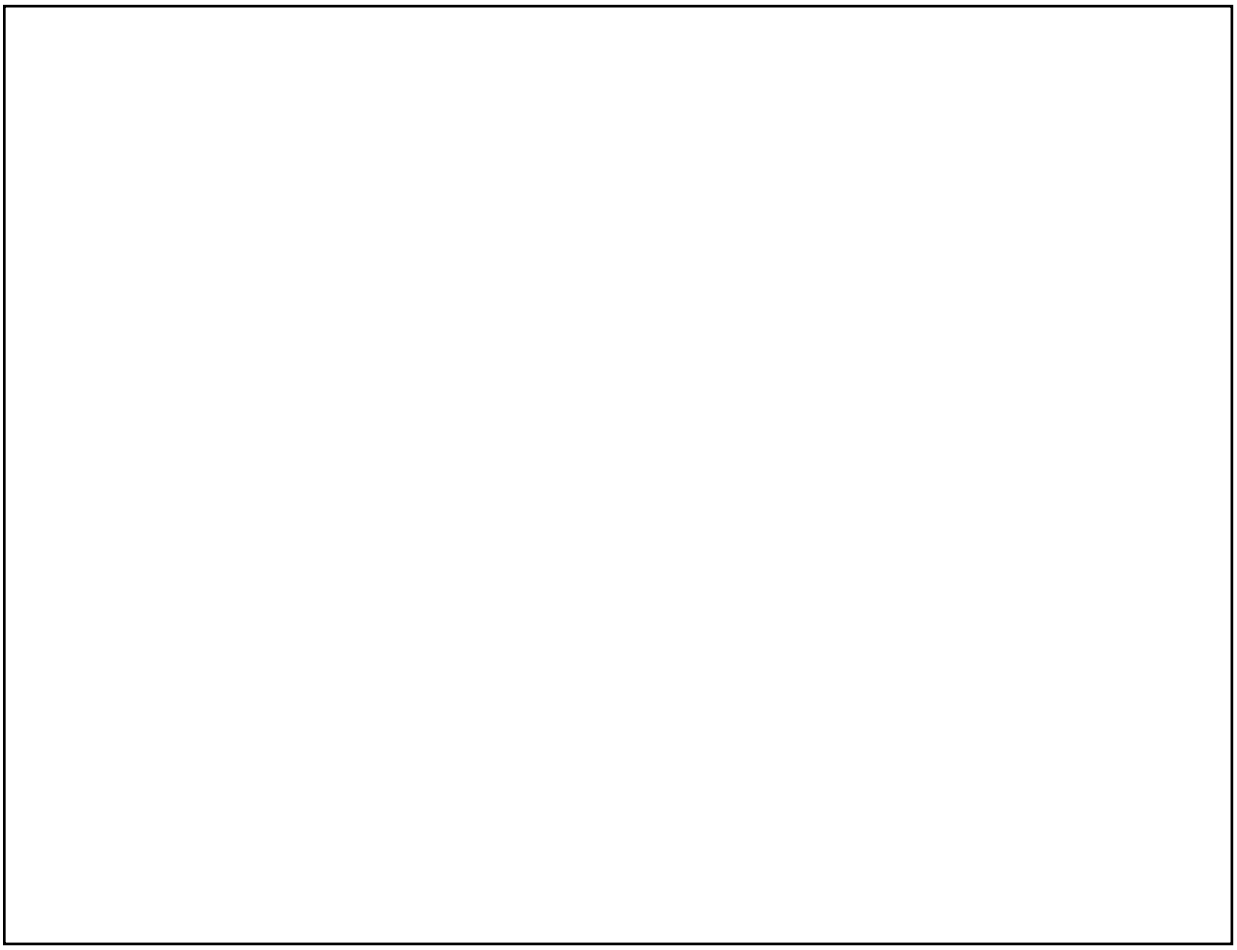


- Pseudo-direct addressing – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC



MIPS (RISC) Design Principles

- Simplicity favors regularity
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- Good design demands good compromises
  - three instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands



Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
    - *"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"*
- Let's look (briefly) at IA-32

CPI = clock cycles per instruction = instructions for a program X average clock cycles per instruction

IA - 32

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits,widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

"This history illustrates the impact of the "golden handcuffs" of compatibility"
"adding new features as someone might add clothing to a packed bag"
"an architecture that is difficult to explain and impossible to love"

IA-32 = Intel Architecture 32 bits

MMX works on FP calculations and SIMD (single instruction multiple data –

Instructions that perform the same operation on multiple data points simultaneously)

but blocks CPU

SSE leaves CPU free

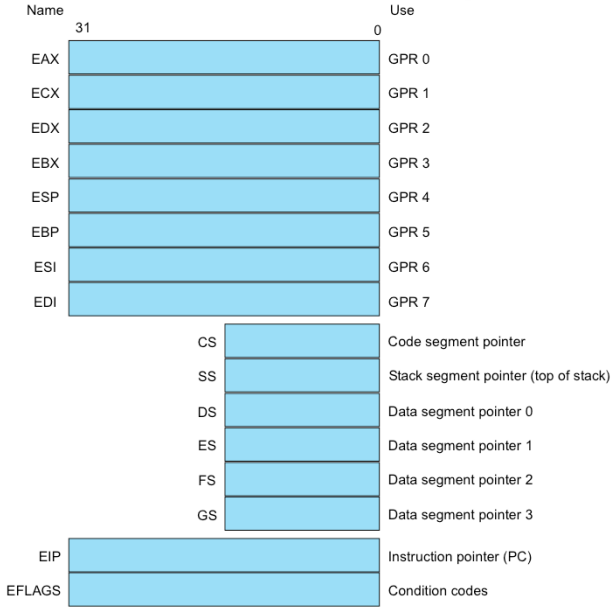
IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



GPR = general purpose processor registers

IA32 memory is divided into segments, pointed by segment registers

Modern operating system and applications use the (unsegmented) memory mode: all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.

EFLAG contains interrupt flags, alignment, I/O privilege, overflow, sign, zero, parity, carry etc

IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)` `#` ≤16-bit displacement
Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)`
Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0)` `#` ≤16-bit displacement

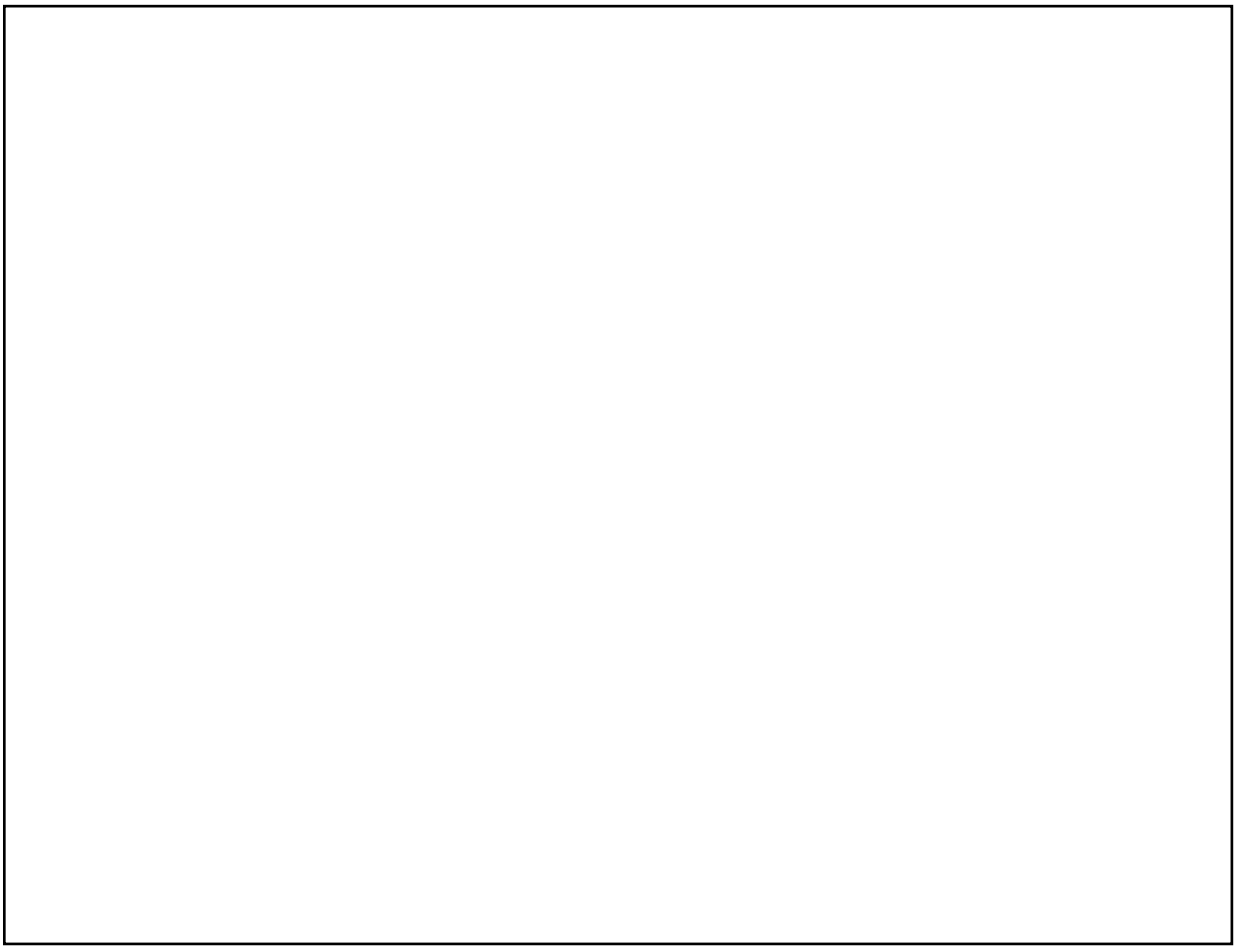
FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

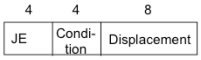
FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)



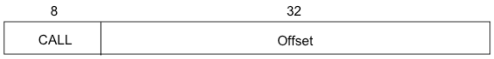
IA-32 instruction Formats

- Typical formats: (notice the different lengths)

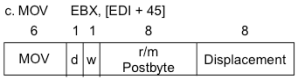
a. JE EIP + displacement



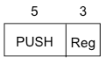
b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42