

Computation

5JJ70

'Implementatie van rekenprocessen'

Translate *C* into MIPS assembly

Henk Corporaal

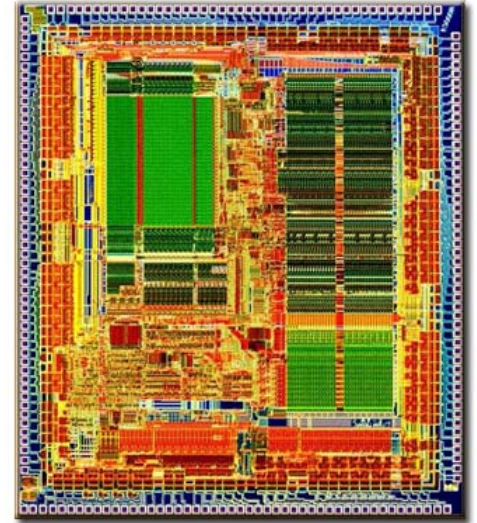
December 2009

Welcome

- So far we covered almost all of C,
- and a large part of C++
- We also introduced the MIPS processor and its instructions
- C++ topics not treated:
 - ◆ copy constructor
 - ◆ abstract classes
 - ◆ virtual destructors
 - ◆ define your own operators (overloading existing ones)
 - ◆ exceptions, using catch and throw
 - ◆ formatted stream i/o
 - ◆ templates (we'll come back on this)
- Refer to the online manuals and tutorials, or the C++ bible

Topics of today

- Final lecture of first semester
- Recap of MIPS instruction set and formats
- MIPS addressing modes
- Register allocation
 - ◆ graph coloring
 - ◆ spilling
- Translating *C* statements into Assembler
 - ◆ if statement
 - ◆ while statement
 - ◆ switch statement
 - ◆ procedure / function (leaf and non-leaf)
 - ◆ stack save and restore mechanism



MIPS_3000

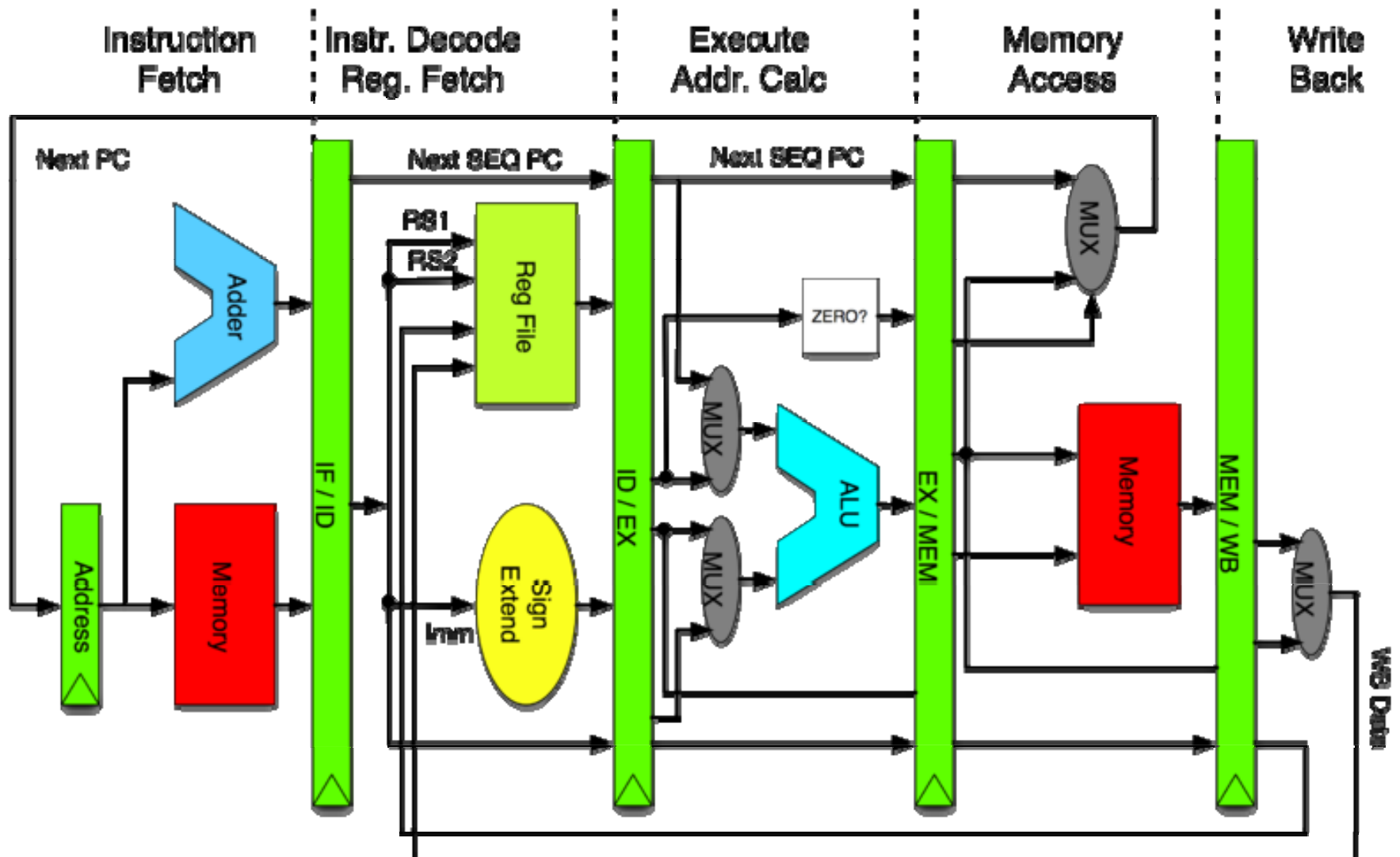
Actions to be performed for an instruction

let's take a load instruction at address 0x800:

```
0x800  lw $s1, 8($s2)
```

1. Cycle 1
 - Fetch instruction to which PC points
 - New PC = PC+4
2. Cycle 2
 - Decode instruction
 - Fetch \$s2 from register file
3. Cycle 3
 - Execution: Add 8 to \$s2
4. Cycle 4
 - Fetch data from address (8+\$s2)
5. Cycle 5
 - Write data back to register file location \$s1

Pipelined MIPS



Main Types of Instructions

- Arithmetic
 - ◆ Integer
 - ◆ Floating Point
- Memory access instructions
 - ◆ Load & Store
- Control flow
 - ◆ Jump
 - ◆ Conditional Branch
 - ◆ Call & Return

MIPS arithmetic

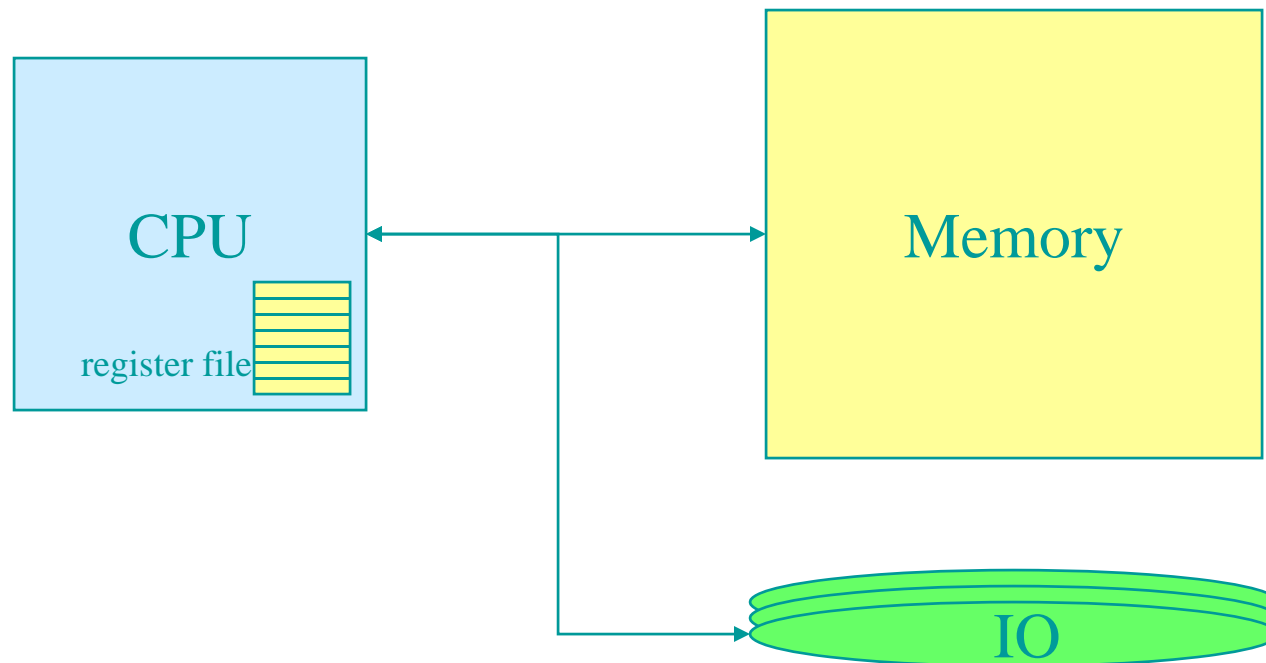
C code: $A = B + C + D;$
 $E = F - A;$

MIPS code: `add $t0, $s1, $s2`
 `add $s0, $t0, $s3`
 `sub $s4, $s5, $s0`

- Operands must be registers, only 32 registers provided
- Design Principle: *smaller is faster*. Why?

Registers vs. Memory

- Arithmetic instructions operands must be registers,
— only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables ?



Register allocation

- Compiler tries to keep as many variables in registers as possible: graph coloring
- Some variables can not be allocated
 - ◆ large arrays (too few registers)
 - ◆ aliased variables (variables accessible through pointers in C)
 - ◆ dynamic allocated variables
 - ☞ heap
 - ☞ stack
- Compiler may run out of registers => *spilling*

Register allocation using graph coloring

Example:

Program:

Live Ranges

a b c d

a define
(def)

a :=

c :=

b :=

:= b

d :=

:= a

:= c

:= d

a consume
(cons)

Register allocation using graph coloring

Inference Graph

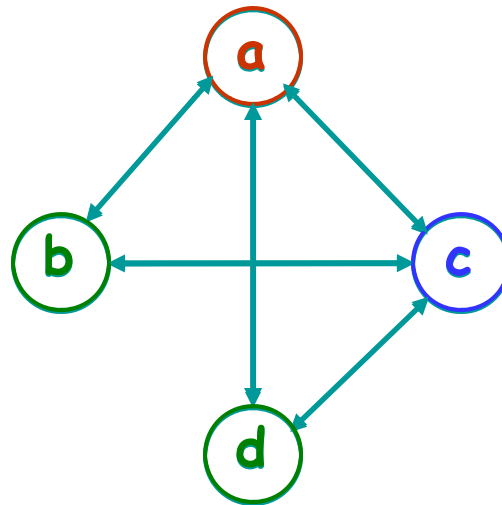
Coloring:

a = red

b = green

c = blue

d = green



Graph needs 3 colors => program needs 3 registers

Register allocation: spilling

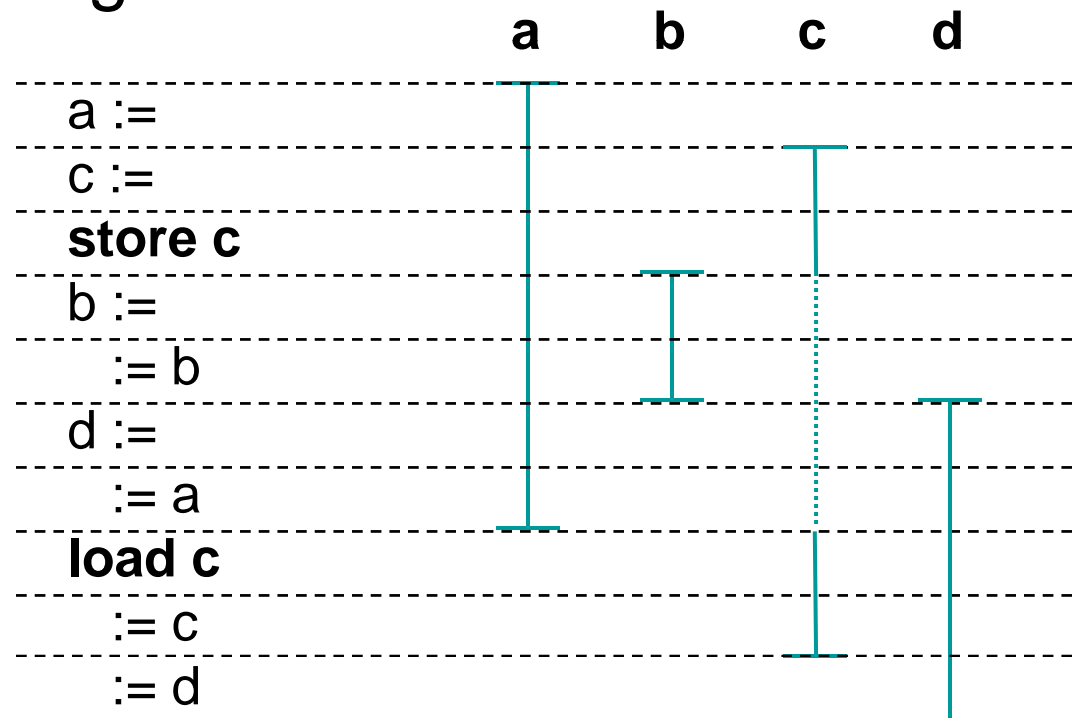
Spill/ Reload code

Spill/ Reload code is needed when there are not enough colors (registers) to color the interference graph

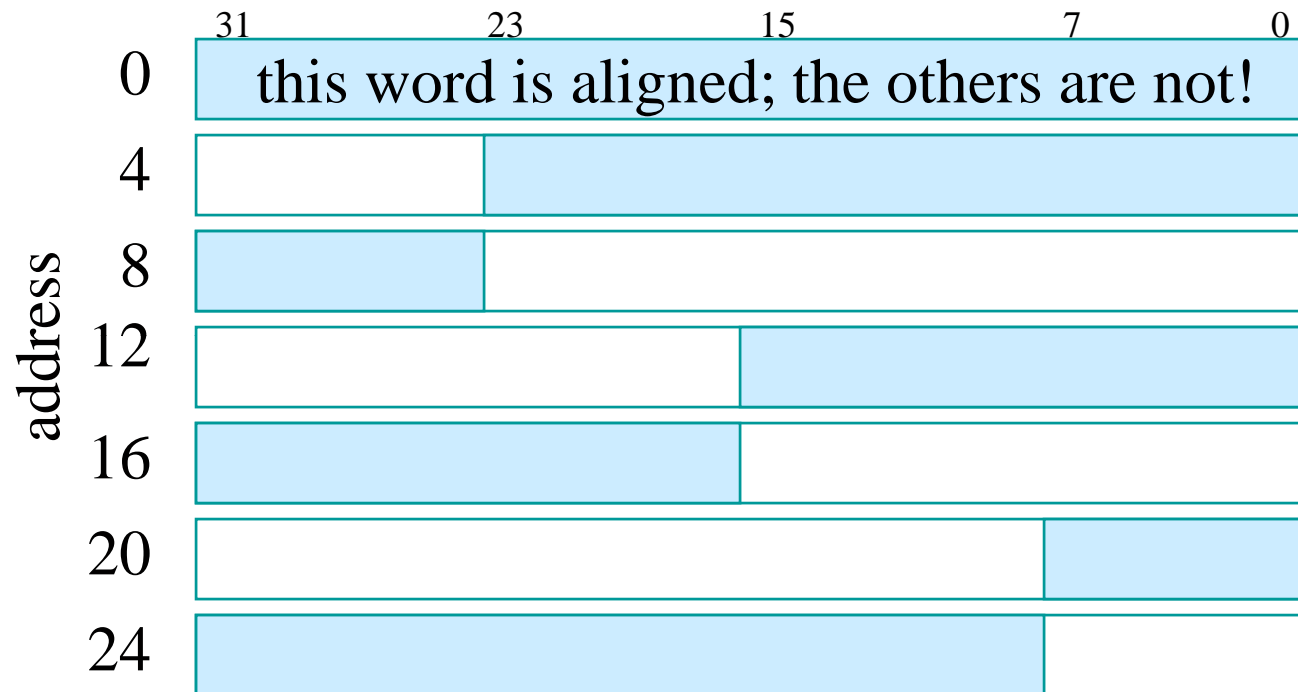
Example:
What if only **two**
registers available ?

Program:

Live Ranges



Memory layout: Alignment



- Question:
If words are aligned, what are then the least 2 significant bits of a word address?

Instructions: Load and store

- Example:

C code: `A[8] = h + A[8];`

MIPS code: `lw $t0, 32($s3)`
 `add $t0, $s2, $t0`
 `sw $t0, 32($s3)`

- Store word operation has no destination (reg) operand
- Remember arithmetic operands are registers, not memory!

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
 - ◆ Example: `add $t0, $s1, $s2`
 - ◆ Registers have numbers: `$t0=9, $s1=17, $s2=18`

- Instruction Format:

<i>op</i>	<i>rs</i>	<i>rt</i>	<i>rd</i>	<i>shamt</i>	<i>funct</i>
000000	10001	10010	01001	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- *Question:*
Can you guess what the field names stand for?

Machine Language

- Consider the load-word and store-word instructions,
 - ◆ What would the **regularity principle** have us do?
 - ◆ New principle: *Good design demands a compromise*
- Introduce a new type of instruction format
 - ◆ I-type for data transfer instructions
 - ◆ other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32
op	rs	rt	16 bit number

- *Where's the compromise?*

Control

- Decision making instructions
 - ◆ alter the control flow,
 - ◆ i.e., change the "next" instruction to be executed

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label  
beq $t0, $t1, Label
```

- Example: if ($i=j$) $h = i + j$;

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

Control

- MIPS unconditional branch instructions:

j label

- Example:

```
if (i!=j)
    h=i+j;
else
    h=i-j;
```



```
beq $s4, $s5, Lab1
add $s3, $s4, $s5
j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

So far:

■ Instruction

Meaning

add \$s1,\$s2,\$s3	\$s1 = \$s2 + \$s3
sub \$s1,\$s2,\$s3	\$s1 = \$s2 - \$s3
lw \$s1,100(\$s2)	\$s1 = Memory[\$s2+100]
sw \$s1,100(\$s2)	Memory[\$s2+100] = \$s1
bne \$s4,\$s5,L	Next instr. is at Label if \$s4 \neq \$s5
beq \$s4,\$s5,L	Next instr. is at Label if \$s4 = \$s5
j Label	Next instr. is at Label

■ Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Used MIPS compiler conventions

Name	Register number	Usage
\$zero	0	the constant value 0
\$v0-\$v1	2-3	values for results and expression evaluation
\$a0-\$a3	4-7	arguments
\$t0-\$t7	8-15	temporaries
\$s0-\$s7	16-23	saved (by callee)
\$t8-\$t9	24-25	more temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address

Constants

- Small constants are used quite frequently (50% of operands) e.g.,
 $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$
- Solutions? Why not?
 - ◆ put 'typical constants' in memory and load them?
 - ◆ create hard-wired registers (like \$zero) for small constants?
 - ◆ ...???

- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori  $29, $29, 4
```

- How do we make this work?

How about larger constants?

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

lui \$t0, 10101010101010101010 filled with zeros

1010101010101010	0000000000000000
------------------	------------------

Then must get the lower order bits right, i.e.,

ori \$t0, \$t0, 1010101010101010

ori

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

Addresses in Branches and Jumps

- Instructions:

bne \$t4,\$t5,Label	Next instruction is at Label if $\$t4 \neq \$t5$
beq \$t4,\$t5,Label	Next instruction is at Label if $\$t4 = \$t5$
j Label	Next instruction is at Label

- Formats:

I	op	rs	rt	16 bit address
J	op	26 bit address		

- Questions:

Above addresses are not 32 bits !!

- ◆ How do we extend them to 32 bits?
- ◆ How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

`bne $t4,$t5,Label`

Next instruction is at Label if $\$t4 \neq \$t5$

`beq $t4,$t5,Label`

Next instruction is at Label if $\$t4 = \$t5$

- Formats:

I	op	rs	rt	16 bit address
---	----	----	----	----------------

- Could specify a register (like `lw` and `sw`) and add it to address

- ◆ use Instruction Address Register (PC = program counter)
- ◆ most branches are local (principle of locality)

- Jump instructions just use high order (4) bits of PC

- ◆ 32-bit Jump address = $PC[31..28] + Instr[25..0] + [00]$
- ◆ Address boundaries of 256 MB

Overview of MIPS

- simple instructions, all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

To summarize:

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	$\$s1 = \$s2 + 100$	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Word from memory to register
	store word	sw \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	$\$s1 = \text{Memory}[\$s2 + 100]$	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	$\text{Memory}[\$s2 + 100] = \$s1$	Byte from register to memory
	load upper immediate	lui \$s1, 100	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq \$s1, \$s2, 25	if ($\$s1 == \$s2$) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne \$s1, \$s2, 25	if ($\$s1 \neq \$s2$) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt \$s1, \$s2, \$s3	if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than; for beq, bne
	set less than immediate	slti \$s1, \$s2, 100	if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr \$ra	go to \$ra	For switch, procedure return
	jump and link	jal 2500	$\$ra = PC + 4$; go to 10000	For procedure call

Floating point instructions

- Mips has separate floating point registers \$f0, \$f1, \$f2, ...
- And special instructions that operate on them:

add.s
sub.s
mul.s
div.s

add.d
sub.d
mul.d
div.d

single (.s) or double (.d)

c.x.s
bc1t

c.x.d
bc1f

Compare two registers

Branch if true, branch if false

lwc1
swc1

Load and save fp words

Floating point instructions examples

```
add.s $f2, $f3, $f4
```

```
add.d $f2, $f4, $f6      // adds pairs of float registers
```

```
lwc1 $f1, 100($s2)       // $f1 = Memory[$s2+100]
```

```
c.lt.s $f2, $f3          // if ($f2 < $f3) cond = 1; else cond = 0
```

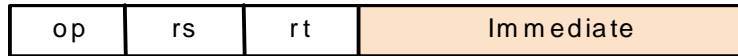
```
bclt 25                  // if (cond==1) goto PC+4+100
```

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
 - ◆ much easier than writing down numbers
 - ◆ e.g., destination register first
- Machine language is the underlying reality
 - ◆ e.g., destination register is no longer first
- Assembly can provide '*pseudoinstructions*'
 - ◆ e.g., "move \$t0, \$t1" exists only in Assembly
 - ◆ would be implemented using "add \$t0, \$t1, \$zero"
- When considering performance you should count real instructions!

MIPS addressing modes

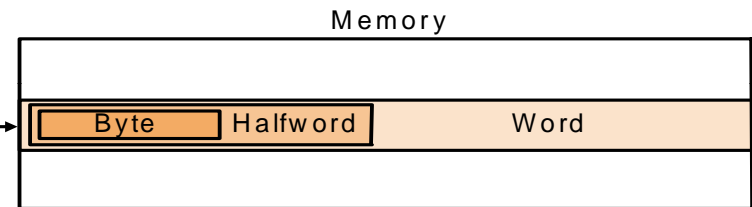
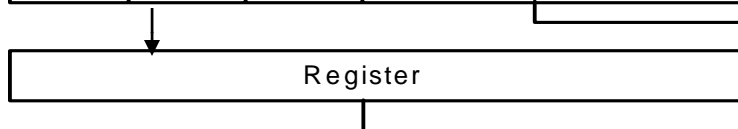
1. Immediate addressing



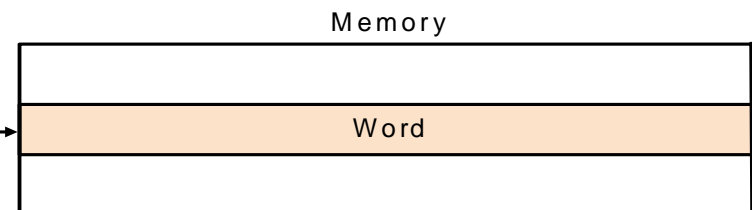
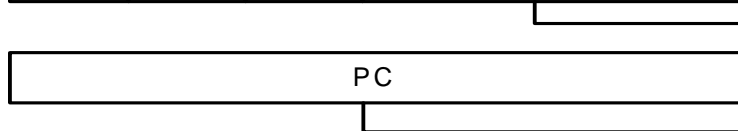
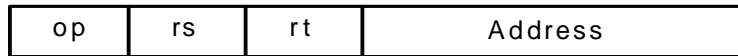
2. Register addressing



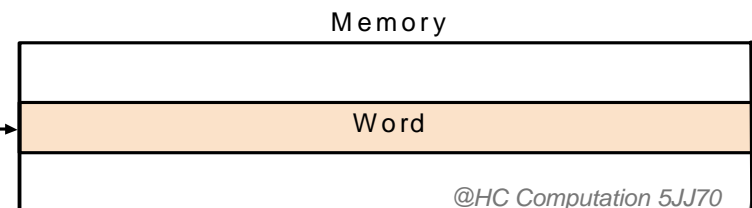
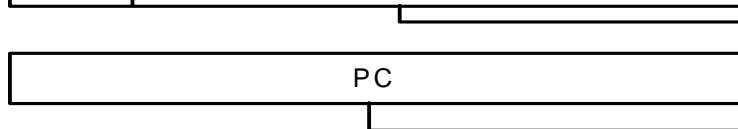
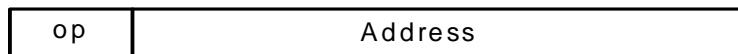
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



More complex stuff

- Inequalities
- While statement
- Case/Switch statement
- Procedure
 - ◆ leaf
 - ◆ non-leaf / recursive
- Stack
- Memory layout
- Characters, Strings
- Arrays versus Pointers

Inequalities

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
```

- Can use this instruction to build "blt \$s1, \$s2, Label"
 - ◆ blt is pseudo instruction
 - ◆ you can now build general control structures
- Note that the assembler needs a register to do this,
 - use conventions for registers

While statement

```
while (save[i] == k)
    i=i+j;
```

Registers allocation:

- i \$s3
- base of save[] \$s6
- k \$s5

```
Loop: muli $t1,$s3,4
      add $t1,$t1,$s6
      lw  $t0,0($t1)
      bne $t0,$s5,Exit
      add $s3,$s3,$s4
      j   Loop
```

```
Exit:
```

*# calculate address of
save[i]*

sll \$t1,\$s3,2

Faster alternative

Case/Switch statement

C Code

```
switch (k) {  
    case 0: f=i+j; ... break;  
    case 1: .....;  
    case 2: .....;  
    case 3: .....;  
}
```

Assembler Code (see book CD for real MIPS code):

1. test if k inside 0-3
2. calculate address of jump table location
3. fetch jump address and jump
4. code for all different cases (with labels L0-L3)

jump table

address L0
address L1
address L2
address L3

Note: earlier we showed a different solution for a Pentium/AMD

MIPS machine code for function calls

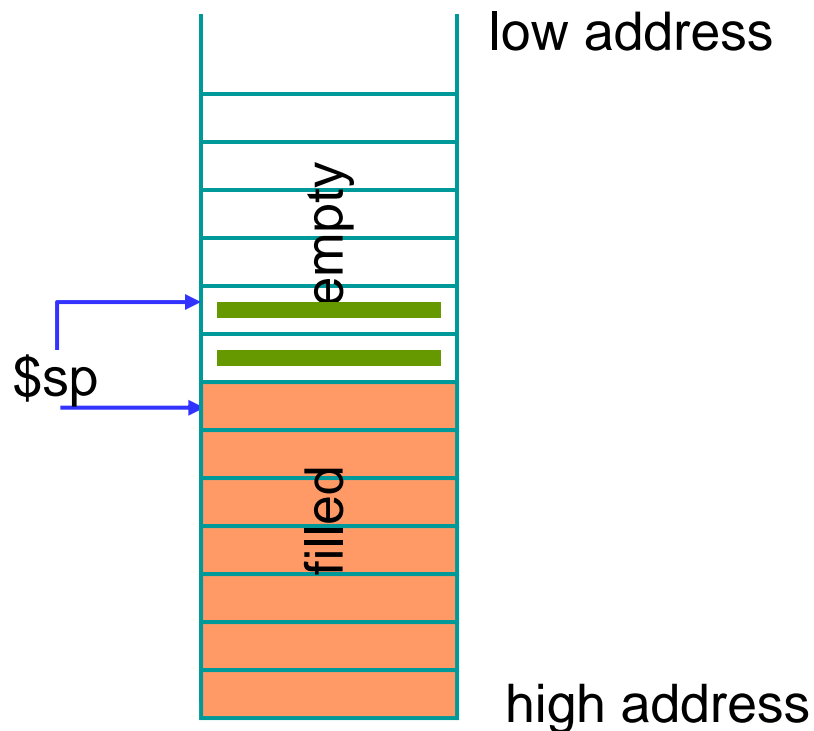
- `$sp` is one of the 32 registers. It is designated to be the stack pointer. It contains the memory address of the top of the stack of used addresses.
- The stack is inverted: it grows from high to low.
- Before a subroutine call, the arguments, and registers that need to be saved are *pushed* on the stack:

```
sub $sp, $sp, 12    # adjust the stack to make room for 3 words
sw  $t1, 8($sp)     # copy the argument in register $t1 to stack
sw  $t0, 4($sp)     # copy the argument in register $t0 to stack
sw  $s0, 0($sp)     # copy the argument in register $s0 to stack
```

- The special instruction `jal` jumps to the subroutine (function), and at the same time stores the return address in register `$ra`
- The return address in `$ra` is the current value of the program counter + 4.

```
jal my_function     # store PC+4 in $a, jump to routine my_function
j   $ra             # return from function call
add $sp, $sp, 12    # adjust the stack pointer, freeing memory.
```

Using a Stack



Save \$s0 and \$s1:

```
subi  $sp, $sp, 8
sw     $s0, 4($sp)
sw     $s1, 0($sp)
```

Restore \$s0 and \$s1:

```
lw     $s0, 4($sp)
lw     $s1, 0($sp)
addi   $sp, $sp, 8
```

Convention: \$ti registers do not have to be saved and restored by callee;
They are scratch registers.

Compiling a leaf Procedure

C code

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

Assembler code:

```
leaf_example: - save registers changed by callee
               - code for expression 'f = ....'
                 (g is in $a0, h in $a1, etc.)
               - put return value in $v0
               - restore saved registers
               - return: jr $ra
```

Compiling a non-leaf procedure

For non-leaf procedure the **callee** should:

- save arguments registers (if used)
- save return address (\$ra)
- save callee used registers (from \$s0-\$s7 set)
- create stack space for local arrays and structures (if any)
- restore registers, saved at beginning, before return
(jr \$ra)

The **caller** should:

- save and restore caller life registers (from \$t0-\$t9)
around function call (jal label)

Compiling a non-leaf procedure

Factorial: $n! = n * (n-1)!$

$0! = 1$

C code of 'recursive' factorial:

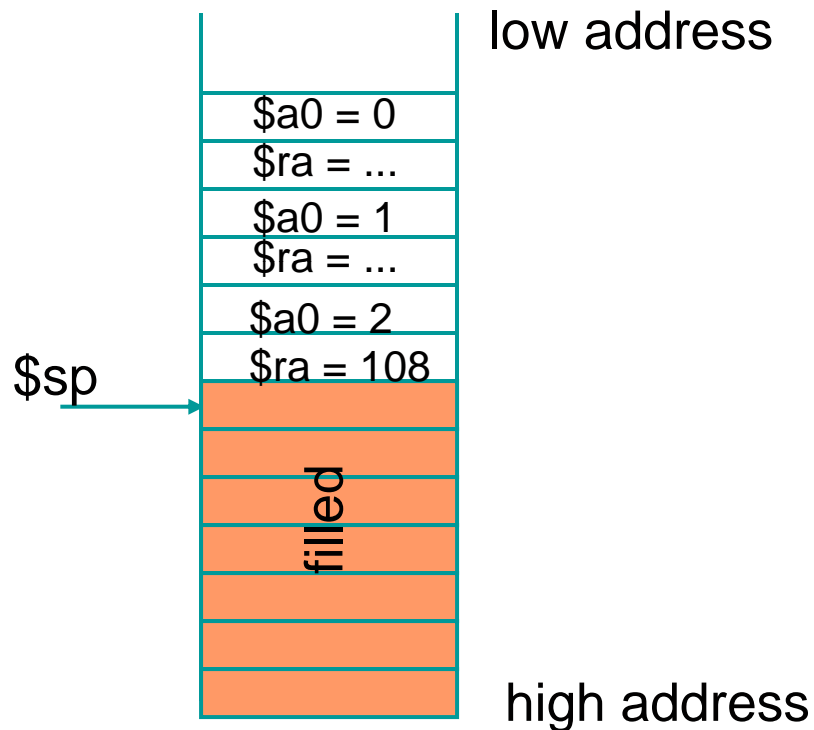
```
int fact (int n)
{
    if (n<1)
        return 1;
    else
        return (n*fact(n-1));
}
```

Compiling a non-leaf procedure

Assembler (callee) code for 'fact'

```
fact: subi $sp,$sp,8      # save return address
      sw   $ra,4($sp)     # and arg.register a0
      sw   $a0,0($sp)
      slti $t0,$a0,1      # test for n<1
      beq  $t0,$zero,L1   # if n>= 1 goto L1
      addi $v0,$zero,1     # return 1
      addi $sp,$sp,8      #
      jr   $ra
L1:   subi $a0,$a0,1
      jal  fact           # call fact with (n-1)
      lw   $a0,0($sp)     # restore return address
      lw   $ra,4($sp)     # and a0 (in right order!)
      addi $sp,$sp,8
      mul  $v0,$a0,$v0     # return n*fact(n-1)
      jr   $ra
```


How does the stack look for fact(2) ?



Caller:

```
100 addi $a0,$zero,2
104 jal fact
108 ....
```

Note: no caller save regs (`$ti`) are used

Beyond numbers: Characters

- Characters are often represented using the ASCII standard
- ASCII = American Standard COde for Information Interchange
- Note: $\text{value}(a) - \text{value}(A) = 32$
 $\text{value}(z) - \text{value}(Z) = 32$

Beyond numbers: Strings

- A string is a sequence of characters
- Representation alternatives for "aap":
 - ◆ including length field: 3'a"a"p'
 - ◆ separate length field
 - ◆ delimiter at the end: 'a"a"p'0 (Choice of language C !!)

Example: procedure 'strcpy':

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while ((x[i]=y[i]) != 0)    /* copy and test byte */
        i=i+1;
}
```

strcpy: MIPS assembly

```
strcpy: subi $sp,$sp,4
        sw   $s0,0($sp)
        add  $s0,$zero,$zero    # i=0
L1:      add  $t1,$a1,$s0        # address of y[i]
        lb   $t2,0($t1)         # load byte y[i] in $t2
        add  $t3,$a0,$s0        # similar address for x[i]
        sb   $t2,0($t3)         # store byte y[i] into x[i]
        addi $s0,$s0,1
        bne  $t2,$zero,L1       # if y[i]!=0 go to L1
        lw   $s0,0($sp)         # restore old $s0
        addl $sp,$sp,4
        jr   $ra
```

Note: strcpy is a leaf-procedure;

- no saving of args and return address required

Arrays versus pointers

Array version (initializing array to 0):

```
clear1 (int array[], int size)
{
    int i;
    for (i=0; i<size; i=i+1)
        array[i]=0;
}
```

Pointer version:

```
clear2 (int *array, int size)
{
    int *p;
    for (p=&array[0]; p<&array[size]; p=p+1)
        *p=0;
}
```

Arrays versus pointers

- Compare the assembly results in your book
- Note the size of the loop body:
 - ◆ Array version: 7 instructions
 - ◆ Pointer version: 4 instructions
- Pointer version much faster !
- Clever compilers perform pointer conversion themselves