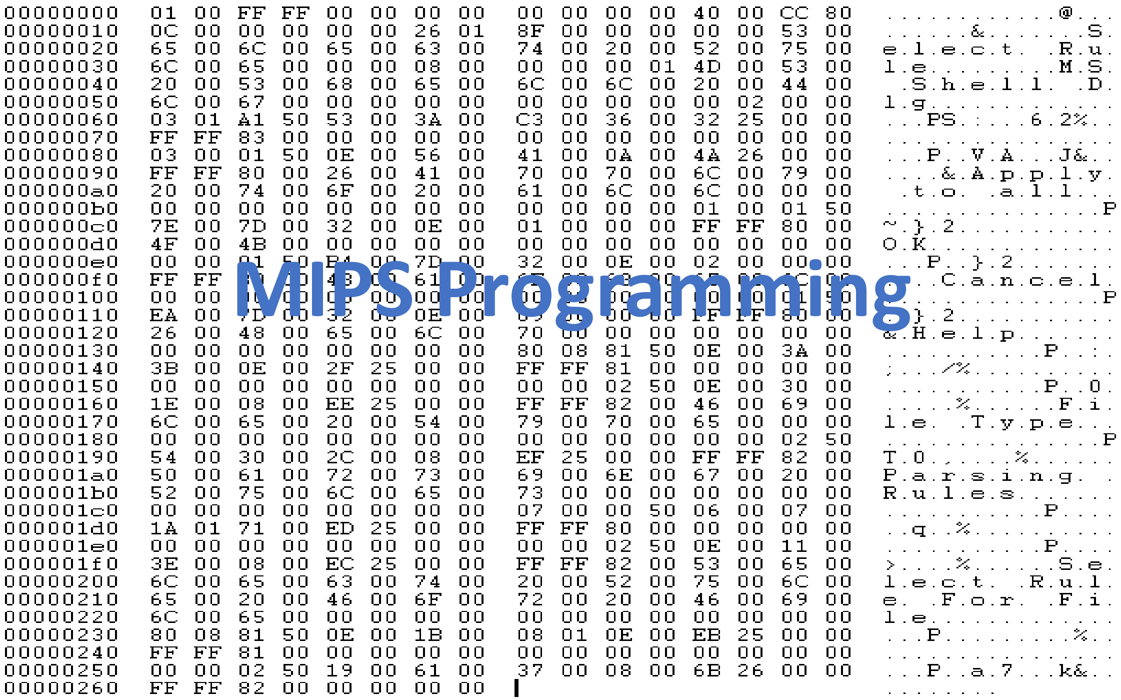
Αρχιτεκτονική Υπολογιστών

Τμήμα I (Α –ΚΑΣ)



MIPS Programming

Μάριος Κόνιαρης

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)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	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 != $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

Instruction Formats

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

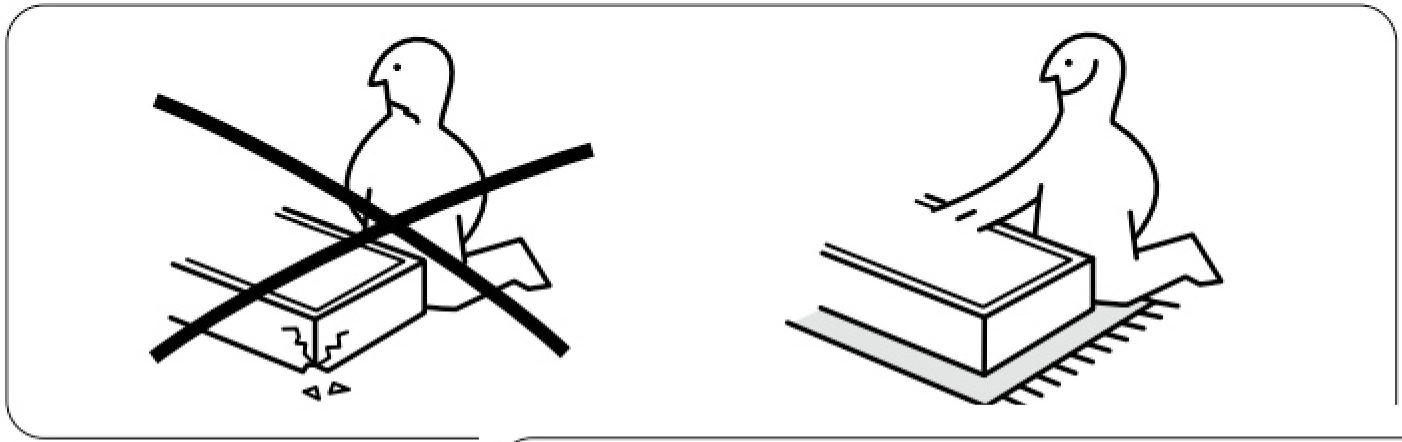
J-Type

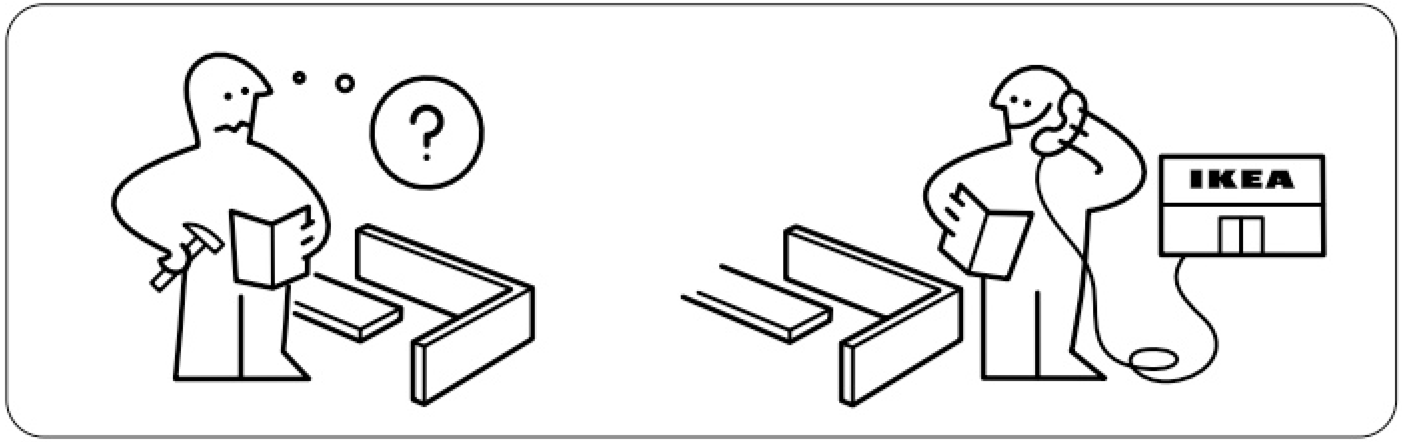
op	addr
6 bits	26 bits

The MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

Instructions





Let's try

```
# $a0 contains an input variable / initialized to an integer n
# $v0 stores the output
            addi  $t0, $zero, 0        #
            addi  $t1, $zero, 2        #
loop:       slt   $t2, $t1, $a0        #
            beq   $t2, $zero, done     #
            add   $t0, $t0, $t1        #
            addi  $t1, $t1, 2          #
            j     loop                 #
done:       add   $v0, $t0, $zero      #
```

Add an appropriate comment to each line of code given

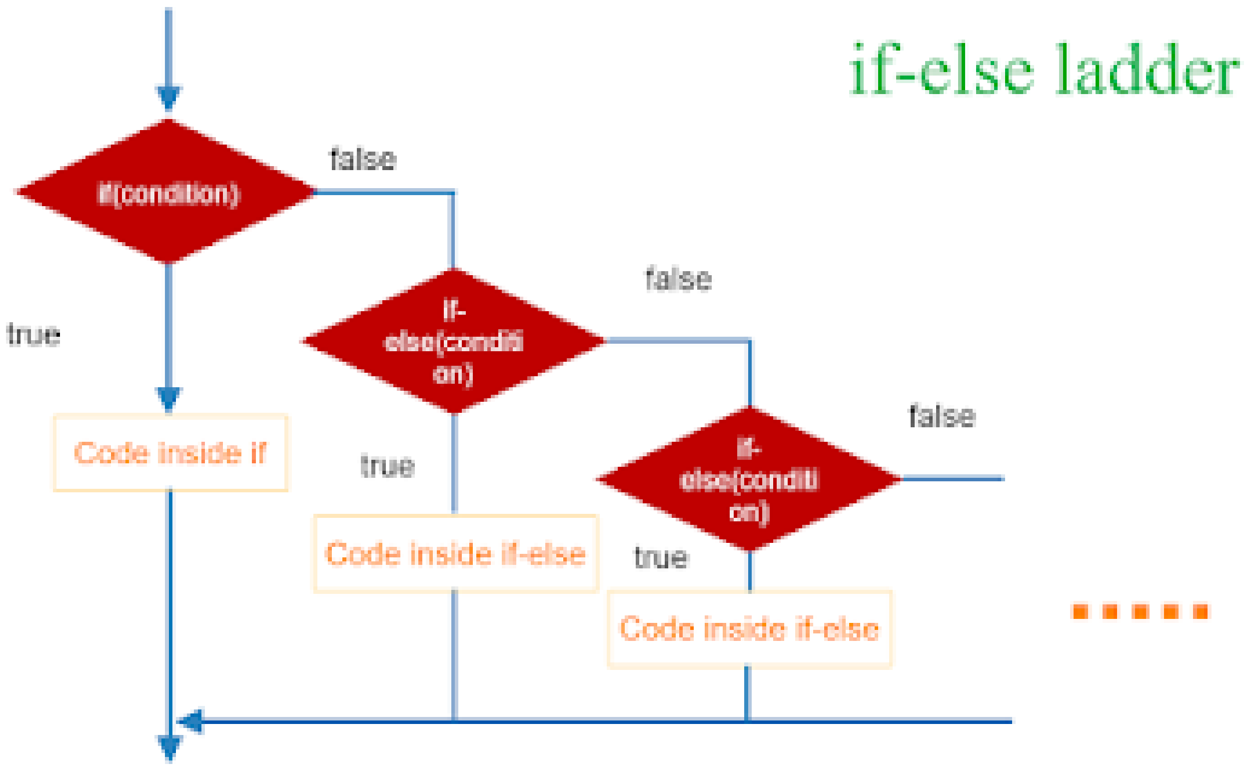
Let's try

MIPS assembly code

```
# $a0 contains an input variable / initialized to an integer n
# $v0 stores the output
        addi  $t0, $zero, 0       # $t0 = $zero + 0
        addi  $t1, $zero, 2       # $t1 = $zero + 2
loop:   slt   $t2, $t1, $a0       # if $t1 < $a0, then $t2 = 1
        beq   $t2, $zero, done    # if $t2 = $zero, then jump to done
        add   $t0, $t0, $t1       # $t0 = $t0 + $t1
        addi  $t1, $t1, 2         # $t1 = $t1 + 2
        j     loop                # jump to loop
done:   add   $v0, $t0, $zero     # $v0 = $t0 + $zero
```

High-Level Code Constructs

- if (else)
- for
- while
- switch



Branching

- execute instructions out of sequence

```
    …
    beq $s0, $s1, target
    addi $s1, $s1, 1
    …
target:
    add $s1, $s1, $s0
```

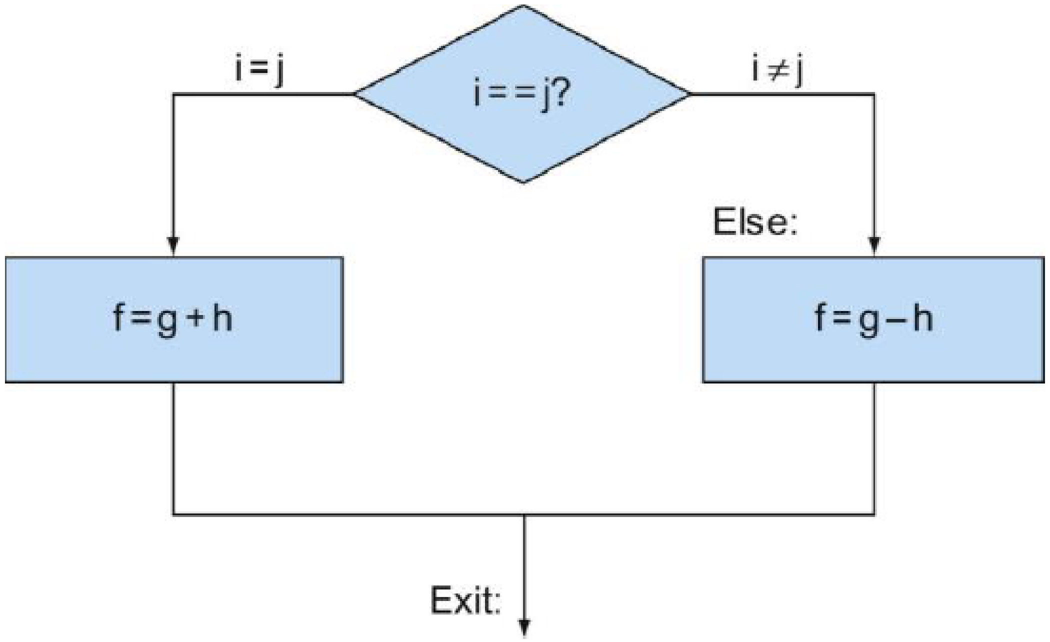
- Conditional branches
  - branch if equal: **beq** (I-type)
  - branch if not equal: **bne** (I-type)
- Unconditional branches
  - jump: **j** (J-type)
  - jump and link: **jal** (J-type)
  - jump register: **jr** (R-type)

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

```
if $s1 &lt; $s2 then
        $t0 = 1
else
        $t0 = 0
```

if (else)

```
if (i== j)
        f = g + h;
else
        f = f –i;
```



MIPS assembly code

```
# $s0 -> f, $s1 -> g, $s2 -> h,
# $s3 -> i, $s4 -> j

        bne $s3, $s4, L1      # i!= j
        add $s0, $s1, $s2
        j done
L1:     sub $s0, $s0, $s3
done:
```

if (else)

```
if ( condition ) {
    statements
}
```

MIPS assembly code

```
        # MIPS code for the condition expression
        # (if condition satisfied set $t0=1)
        beq $t0, $zero, if_end_label
        # MIPS code for the statements
if_end_label:
```

Let's try

```
int y;
if (x == 5) {
        y = 8;
}
else if (x < 7) {
        y = x + x;
}
else {
        y = -1;
}
```

MIPS assembly code

```
# $t0 -> x, $t1 -> y
```

Let's try

```c
int y;
if (x == 5) {
        y = 8;
}
else if (x < 7) {
        y = x + x;
}
else {
        y = -1;
}
```

MIPS assembly code

```
# $t0 -> x, $t1 -> y
        addi $t2, $zero, 5          #$t2=5
        beq $t0, $t2, equal_5
        addi $t2, $zero, 7          #$t2=7
        slt $t3, $t0, $t2
        bne $t3, $zero, less_than_7
        addi $t1, $zero, -1         #y=-1
        j after_branches
equal_5:
        addi $t1, $zero, 8          #y=8
        j after_branches
less_than_7:
        add $t1, $t0, $t0           #y=x+x
after_branches:
```

MIPS assembly code

if (else)

```
if ( condition ) {
        if-statements
} else {
        else-statements
}
```

```
        # MIPS code for the condition expression
        #(if condition satisfied set $t0=1)
        beq $t0, $zero, else_label
        # MIPS code for the if-statements
        j if_end_label
else_label:
        # MIPS code for the else-statements
if_end_label:
```

for

executes before the loop begins

tested at the beginning of each iteration

executes at the end of each iteration

for (initialization; condition; loop operation)

executes each time the condition is met

loop body

```
// add numbers from 0 to 9
int sum= 0;
int i;
for(i = 0; i != 10; i = i+1) {
        sum = sum + i;
}
```

(i== 10) ?

```
# $s0 -> i, $s1 -> sum
     addi $s1, $0, 0
     addi $s0, $0, 0
     addi $t0, $0, 10
for: beq $s0, $t0, done
     add $s1, $s1, $s0
     addi $s0, $s0, 1
     j for
done:
```

MIPS assembly code

for loop

for (init ; condition ; incr) {
statements
}

```
        # MIPS code for the init expression
for_start_label:
        # MIPS code for the condition expression
        #(if condition satisfied set $t0=1)
        beq $t0, $zero, for_end_label
        # MIPS code for the statements
        # MIPS code for the incr expression
        j for_start_label
for_end_label:
```

Let‘s try

```
// add powers of 2
int sum = 0;
int i;
for(i= 1; i< 101; i= i*2) {
        sum = sum + i;
}
```

MIPS assembly code

```
# $s0 -> i, $s1 -> sum
```

Let's try

```c
// add powers of 2
int sum = 0;
int i;
for(i= 1; i< 101; i= i*2) {
        sum = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0
        addi $s0, $0, 1
        addi $t0, $0, 101
loop:   slt $t1, $s0, $t0
        beq $t1, $0, done
        add $s1, $s1, $s0
        sll $s0, $s0, 1        #$s0=$s0*2
        j loop
done:
```

MIPS assembly code

while

while (condition) {
statements
}

```
while_start_label:
        # MIPS code for the condition expression
        #(if condition satisfied set $t0=1)
        beq $t0, $zero, while_end_label
        # MIPS code for the statements
        j while_start_label
while_end_label:
```

Let's try

```
n = 3;
sum = 0;
while (n != 0) {
sum += n;
n--;
}
```

MIPS assembly code

```
# $t0 = n, $t1 = sum
```

Let's try

```
n = 3;
sum = 0;
while (n != 0) {
sum += n;
n--;
}
```

MIPS assembly code

```
# $t0 -> n, $t1 -> sum


        addi $t1,$zero,3
        add $t0,$zero,$zero
loop:
        beq $t0, $zero, loop_exit
        add $t1, $t1, $t0
        addi $t0, $t0, -1
        j loop
loop_exit:
```

MIPS assembly code

do-while

```
do {
statements
} while ( condition );
```

```
do_start_label:
        # MIPS code for the statements
do_cond_label:
        # MIPS code for the condition expression
        #(if condition satisfied set $t0=1)
        beq $t0, $zero, do_end_label
        j do_start_label
do_end_label:
```

switch

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

- code for all different (labels L0-L3)
- … and jump

switch

```
switch ( expr ) {
case const1: statement1
case const2: statement2
...
case constN: statementN
default: default-statement
}
```

MIPS assembly code

```
        # MIPS code for $t0=expr
        beq $t0, const1, switch_label_1
        beq $t0, const2, switch_label_2
        ...
        beq $t0, constN, switch_label_N
        j switch_default
switch_label_1:
        # MIPS code to compute statement1
switch_label_2:
        # MIPS code to compute statement2
...
switch_default:
        # MIPS code to compute def-statement
switch_end_label:
```

Let‘s try

```
switch (i) {
case 0: j = 3; break;
case 1: j = 5; break;
case 2: ;
case 3: j = 11; break;
case 4: j = 13; break;
default: j = 17;
}
```

MIPS assembly code

```
# $s1 -> i,  $s2 -> j
```

Let's try

```
switch (i) {
case 0: j = 3; break;
case 1: j = 5; break;
case 2: ;
case 3: j = 11; break;
case 4: j = 13; break;
default: j = 17;
}
```

MIPS assembly code

```
# $s1 -> i,  $s2 -> j
        add $t0, $zero, $zero     # $t0 = 0, temp. variable
        beq $t0, $s1, case0       # go to case0
        addi $t0, $t0, 1          # $t0 = 1
        beq $t0, $s1, case1       # go to case1
        addi $t0, $t0, 1          # $t0 = 2
        beq $t0, $s1, case2       # go to case2
        addi $t0, $t0, 1          # $t0 = 3
        beq $t0, $s1, case3       # go to case3
        addi $t0, $t0, 1          # $t0 = 4
        beq $t0, $s1, case4       # go to case4
        j default                 # go to default case
case0:
        addi $s2, $zero, 3        # j = 3
        j finish                  # exit switch block
…
```

MIPS assembly code

logical AND

```
if (cond1 && cond2){
statements
}
```

```
        # MIPS code to compute cond1
        # Assume that this leaves the value in $t0
        # If cond1=false $t0=0
        beq $t0, $zero, and_end
        # MIPS code to compute cond2
        # Assume that this leaves the value in $t0
        # If cond2=false $t0=0
        beq $t0, $zero, and_end
        # MIPS code for the statements
and_end:
```

```
lw $s1, 100($s2)
# $s1 = Memory[$s2 + 100]
sw $s1, 100($s2)
# Memory[$s2 + 100] = $s1
```

Accessing Memory

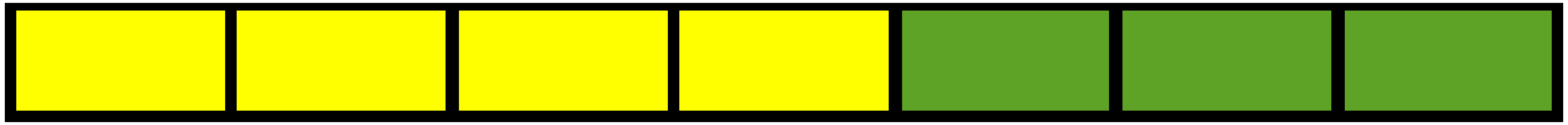
- Some data structures have to be stored in memory
- Two base instructions:
  - load-word (**lw**) from memory to registers
  - store-word (**sw**) from registers to memory
- Also
  - load-half (**lh**), load-byte (**lb**), store-half (**sh**), store-byte (**sb**)

Alignment Requirements

- Word accesses (lw or sw) should be word-aligned → (divisible by 4)
- Half-word accesses → halfword aligned addresses (even addresses)
- No constraints for byte accesses

- Access to word at address 8 is aligned

Address: 8 9 10 11 12 13 14



- Access to word at address 10 is unaligned

Address: 8 9 10 11 12 13 14

Example: Array Access

Array → A sequence of data elements which is contiguous in memory

B is an array of 9 bytes starting at address 8:

Address:	8	9	10	11	12	13	14	15	16
	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]

offset = i * the size of a single element
address [i] = address of first element + offset

```
# Access the i-th element of an array A (element is 32-bit long)
# $t0 = A[0], $t1 = i
     sll $t1,$t1,2    # $t1 = 4*i
     add $t2,$t0,$t1  # add offset to the address of A[0]
                      # now $t2 = address of A[i]
     lw $t3,0($t2)    # $t3 = whatever is in A[i]
```

Arrays

- Base address → address of the first array element, array[0]
- Loop through an array
  - *compute the address &A[i]*
  - *read from that address*

```
// A is a byte array
int sum = 0;
for(int i = 0; i < n; i++)
        sum += A[i];
```

MIPS assembly code

```
#$a0= A[0],$a1=n,$t0 = sum, $t1 = i
        addi $t1, $0, 0          # i =0
        addi $t0, $0, 0          # sum =0
loop:
        beq $a1, $t1, done       # $a1 ==? $t1
        add $t2, $a0, $t1        # t2 = &A[i]
        lb $t2, 0($t2)           # read byte
        add $t0, $t0, $t2        # sum += t2
        addi $t1, $t1, 1         # i++
        j loop
done:
```

Arrays

```
// A is a byte array
int sum = 0;
for(int i = 0; i < n; i ++)
      sum += A[i];
```

```
#$a0= A[0],$a1=n,$t0 = sum
        addi $t0, $0, $0
        add $t1, $a0, $a1        # t1 = &A[n]
loop:
        beq $a0, $t1, done
        lb $t2, 0($a0)           # read byte
        add $t0, $t0, $t2        # sum += t2
        addi $a0, $a0, 1         # advance pointer
        j loop
done:
```

Version 2

Let's try

MIPS assembly code

```
# $a0 = x[0], $a1 = y[0], $s0 = i
```

```
void strcpy (char x[], char y[]) {
 int i;
 i=0;
 while ((x[i]=y[i]) != 0)
        i=i+1;
}
```

Let's try

```c
void strcpy (char x[], char y[])
{
 int i;
 i=0;
 while ((x[i]=y[i]) != 0)
        i=i+1;
}
```

MIPS assembly code

```
# $a0 = x[0], $a1 = y[0], $s0 = i


strcpy:
        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 for x[i]
        sb $t2,0($t3)       # store byte y[i] in x[i]
        addi $s0,$s0,1
        bne $t2,$zero,L1    # if y[i]!=0 go to L1
```

Let's try

```c
//int x[100], y[100], z[100];
void sumarray(int a[], int b[], int c[])
{
        int i;
        for(i= 0; i< 100; i++)
                c[i] = a[i] + b[i];
}
```

MIPS assembly code

```
# $a0=a[0], $a1=b[0], $a2=c[0]
```

Let's try

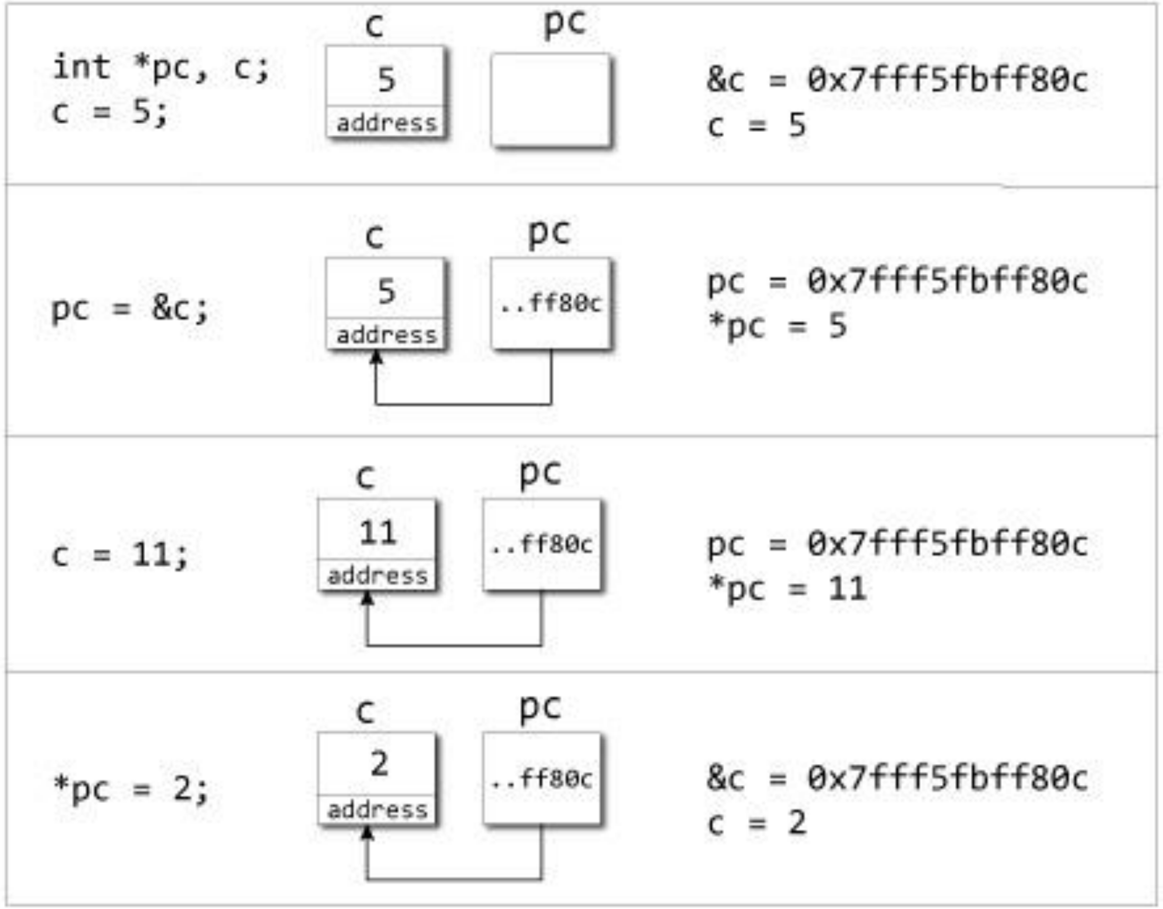
```c
//int x[100], y[100], z[100];
void sumar(int a[], int b[], int c[])
{
        int i;
        for(i= 0; i< 100; i++)
                c[i] = a[i] + b[i];
}
```

MIPS assembly code

```
# $a0 = a[0], $a1= b[0], $a2= c[0]
        addi $t0,$a0,400 # beyond a[]
Loop:   beq $a0,$t0,Exit_loop
        lw $t1, 0($a0)      # $t1=a[i]
        lw $t2, 0($a1)      # $t2=b[i]
        add $t1,$t1,$t2     # $t1=a[i] + b[i]
        sw $t1, 0($a2)      # c[i]=a[i] + b[i]
        addi $a0,$a0,4      # $a0++
        addi $a1,$a1,4      # $a1++
        addi $a2,$a2,4      # $a2++
        j Loop
Exit_loop:
```

Pointers

- an object that stores a memory address
  - a pointer references a location in memory (&)
  - obtaining the value stored at that location is dereferencing the pointer. (*)



Let‘s try

MIPS assembly code

```
# $a0= dst, $a1=value, $a2=num_words
```

```
// writes an integer value num words times
// starting from address dst
void memset(int *dst, int value, int num_words) {
  for (int i = 0; i < num_words; ++i) {
        *dst = value;
        dst++;
  }
}
```

MIPS assembly code

Let's try

```c
void memset(int *dst, int value, int
num_words) {
  for (int i = 0; i < num_words; i++) {
        *dst = value;
        dst++;
  }
}
```

```
# $a0= dst, $a1=value, $a2=num_words
memset:
        add $t1,$zero,$zero
loop:
        slt $t2, $t1, $a2   # If i>=num_words
        beq $t2, $zero, exit # exit loop
        sw $a1, $a0          # *dst = value
        addi $a0, $a0, 4     # address inc by 4
        addi $t1, $t1, 1     # i++
        j loop
exit:
```

Procedures

```
// High level code
void main() {
int y;
y = sum(42, 7);
...
}

int sum(int a, int b) {
 return(a + b);
}
```

Caller

- calling procedure
- passes arguments to callee

  $a0-$a3
- jumps to the callee

  jump and link (jal)

  saves (PC+4) in the return address register ($ra)

Callee

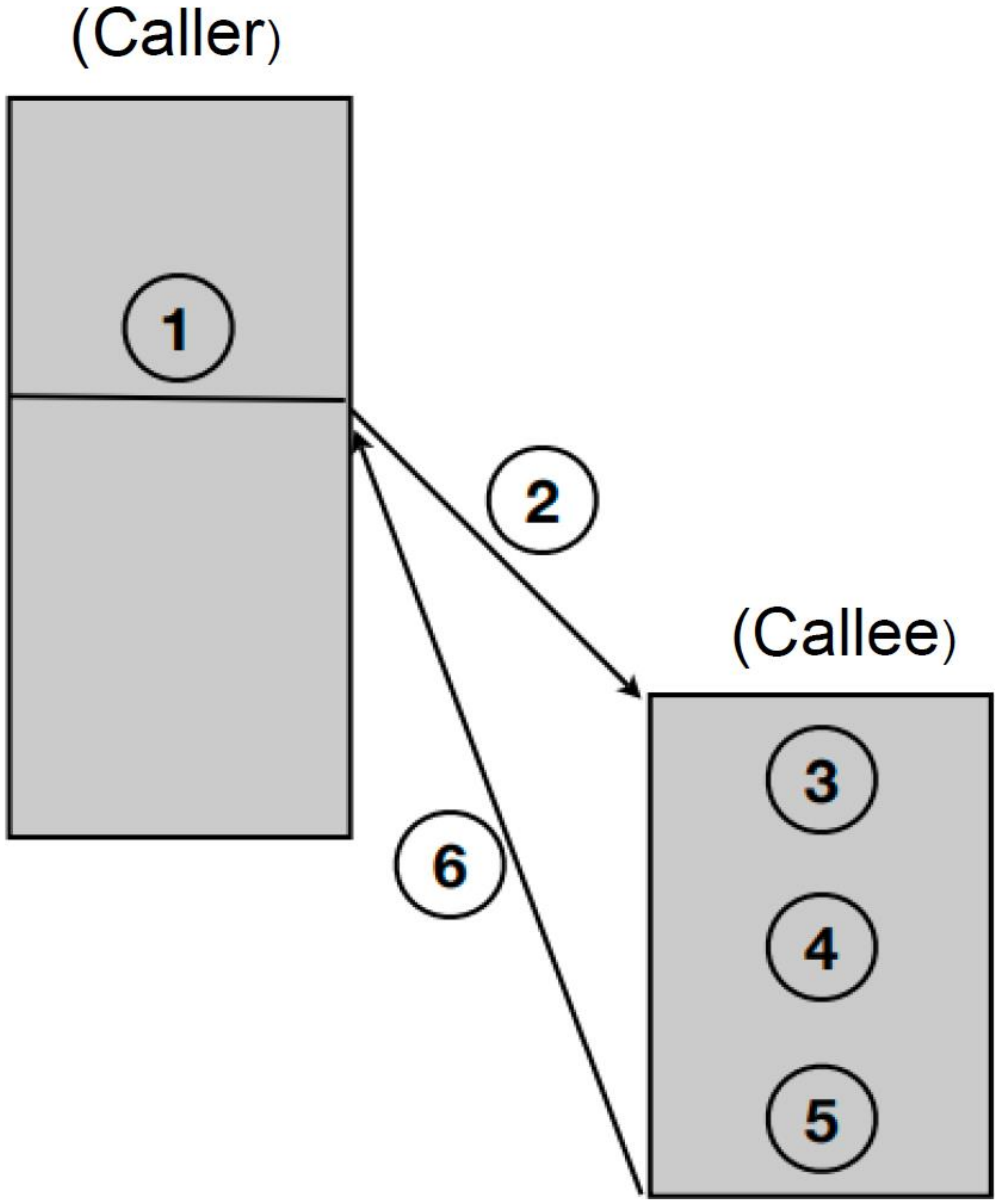
- called procedure
  - performs the procedure
- returns the result to caller $v0 - $v1
- returns to the point of call

  jump register (jr)

- **must not overwrite registers or memory needed by the caller**
- **unintended side effects**

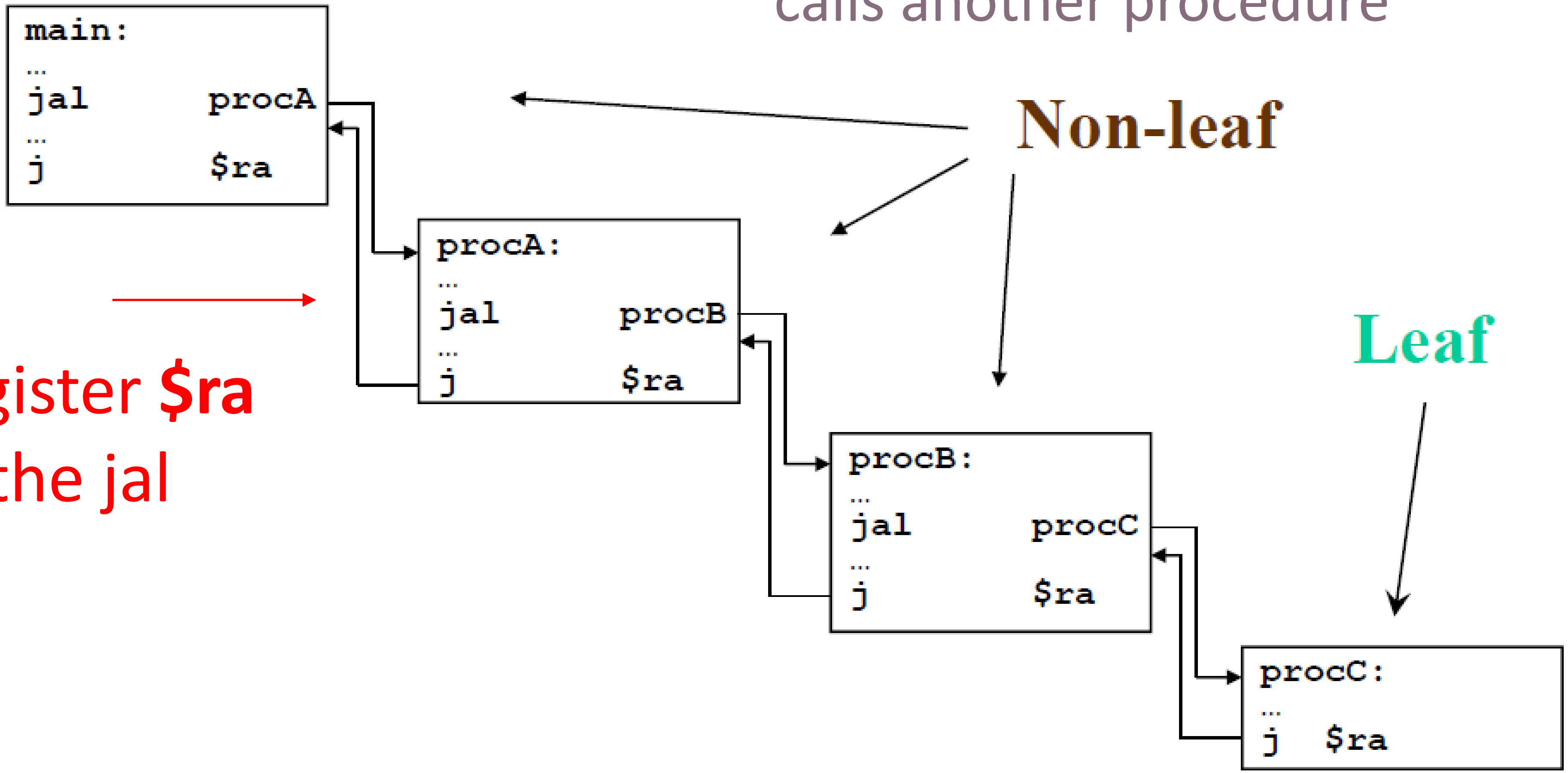
Procedure Steps

1. set up parameters
2. transfer to procedure
3. acquire storage resources
4. do the desired function
5. make result available to caller
   (release storage resources)
6. return to point of call



Non-Leaf - Leaf Procedure

calls another procedure



save at least register **$ra**

overwritten by the jal

Where ?

does not call another procedure

The Stack

- When a program starts executing, a certain *contiguous* section of memory is set aside for the program called the **stack**

stack pointer $sp

The stack grows
from high addresses to low addresses

The *smallest* valid address of a stack

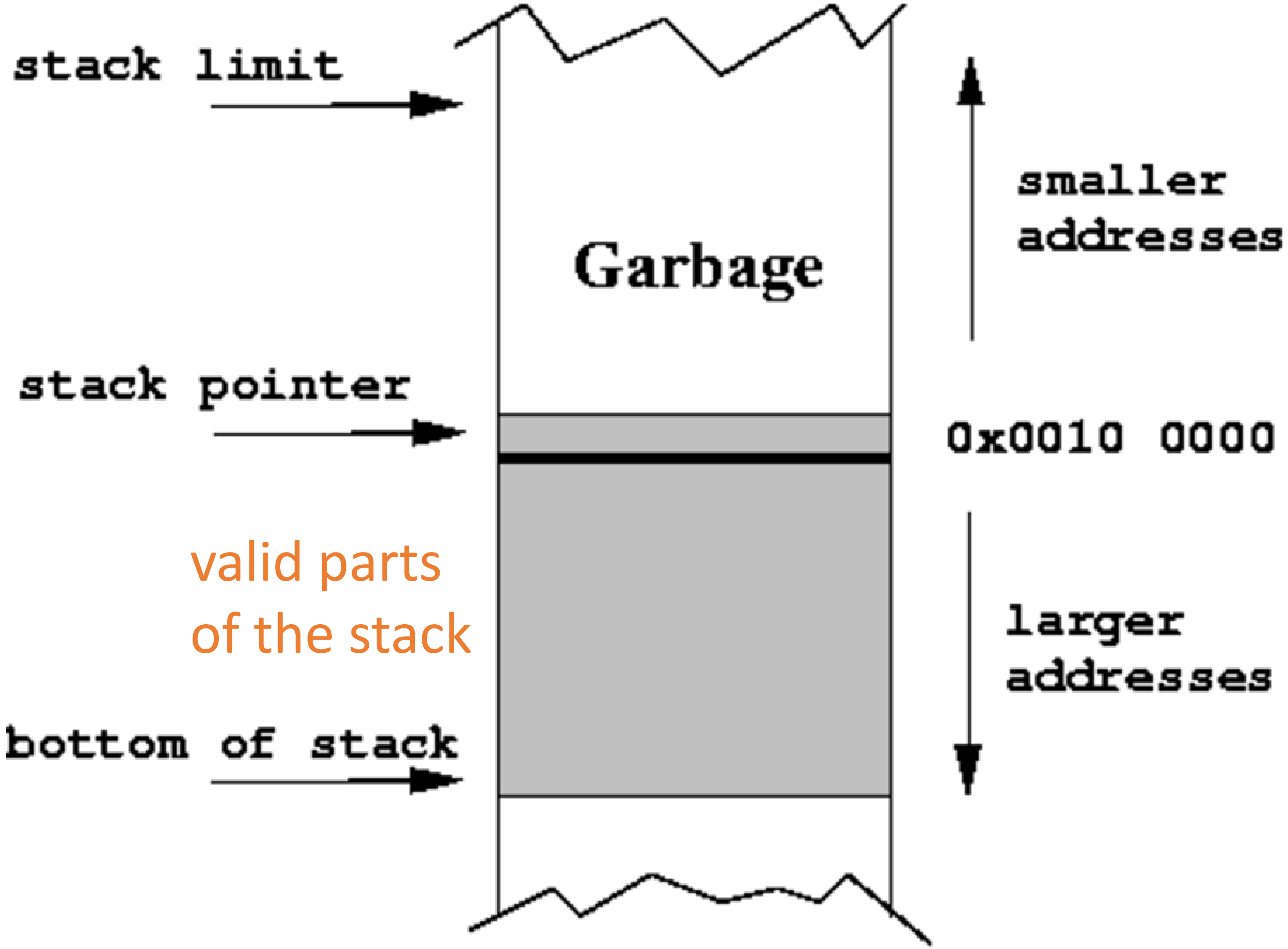
stack overflow error

$sp

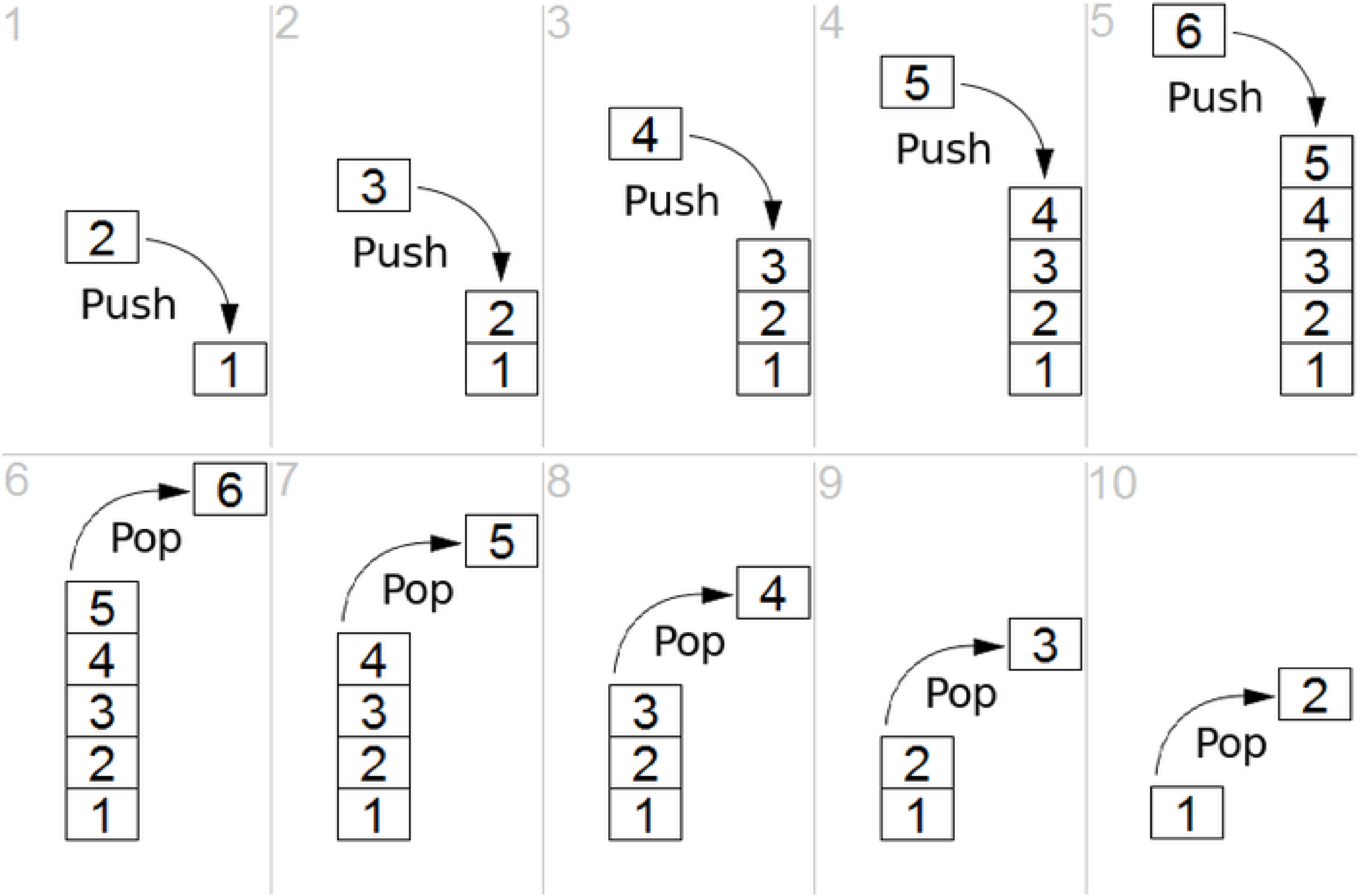
When a stack is initialized,
$sp points to the stack bottom

valid parts
of the stack

The largest valid address of a stack



STACK (LIFO) PUSH & POP



Stack Push & Pop

PUSH

- store registers into the stack

addi $sp, $sp, -8 # 2 registers to save

sw $s0, 4($sp)

sw $s1, 0($sp)

make room for more data

once on entry

POP

- reverse process from **push**

lw $s0, 4($sp)

lw $s1, 0($sp)

addi $sp, $sp, 8 # 2 registers to restore

release space on the stack

once on exit

Saving Registers during Calls

Callee can overwrite

Caller saved

- **$t0 … $t7**
- **$a0 … $a3**
- **$v0, $v1**
- the calling method must save and restore

  (if it depends on them after the call)

Callee saved

- **$s0-$s7**
- **$ra**
- **$sp**
- called method must save & restore
- values ***MUST be the same*** immediately
  - before a function call
  - after the function returns

Caller-saved vs Callee-saved

```
void main() {
int y;
y = sum(42, 7);
...
}
int sum(int a, int b) {
 return(a + b);
}
```

Caller

- Put arguments in $a0-$a3
- Save any registers that are needed ($ra, ? $t0-$t9)
- Call procedure jal callee
- Restore registers
- Look for result in $v0

Callee

- Save registers that might be disturbed ($s0-$s7)
- Perform procedure
- Put result in $v0
- Restore registers
- Return from procedure jr $ra

Let‘s try

```
void main() {
        int y;
        y = sum(42, 7);
}
int sum(int a, int b) {
        return(a + b);
}
```

MIPS assembly code

```
# $s0 = y
```

Let‘s try

```c
void main() {
        int y;
        y = sum(42, 7);
}
int sum(int a, int b) {
        return(a + b);
}
```

MIPS assembly code

```
# $s0 = y
main:
        addi $a0, $0, 42
        addi $a1, $0, 7
        jal sum
        add $s0, $v0, $0 #y = sum(42, 7)
sum:
        add $v0,$a0,$a1 #return value $v0
        jr $ra
```

Let‘s try

```c
int main() {
        int y;
        y = sum(42, 7);
        return y;
}
int sum(int a, int b) {
         return(a + b);
}
```

MIPS assembly code

1 save

2 restore

```
# $s0 = y
main:
        addi $sp, $sp, -4      # make space
        sw $ra, 0($sp)         # save $ra
        addi $a0, $0, 42
        addi $a1, $0, 7
        jal sum
        add $s0, $v0, $0      #y = sum(42, 7)
        add $v0, $s0, $0
        lw $ra, 0($sp)         # restore $s0
        addi $sp, $sp, 4 # deallocate space
        jr $ra
sum:
        add $v0,$a0,$a1 #return value $v0
        jr $ra
```

Let‘s try

```
int addTwo(int a, int b){
        int temp = a + b;
        return temp;
}

int doSomething(int x, int y) {
        int a = addTwo(x, y);
        int b = addTwo(y, x);
        return a + b;
}
```

MIPS assembly code

```
# What should I map a and b to?

# Can I map temp to $t0?

# What should I map x and y to?

# What should I map a + b to?
```

Let‘s try

```
int addTwo(int a, int b){
        int temp = a + b;
        return temp;
}

int doSomething(int x, int y) {
        int a = addTwo(x, y);
        int b = addTwo(y, x);
        return a + b;
}
```

MIPS assembly code

What should I map a and b to?

$a0 , $a1

Can I map temp to $t0?

OK -- *I don't care about **$t****

temp** -> **$v0

What should I map x and y to?

***$s0, $s1** (preserved)*

What should I map a and b to?

*"**a+b**" has to eventually be **$v0**.*

***a** -> preserved, b -> preserved ?*

Let‘s try

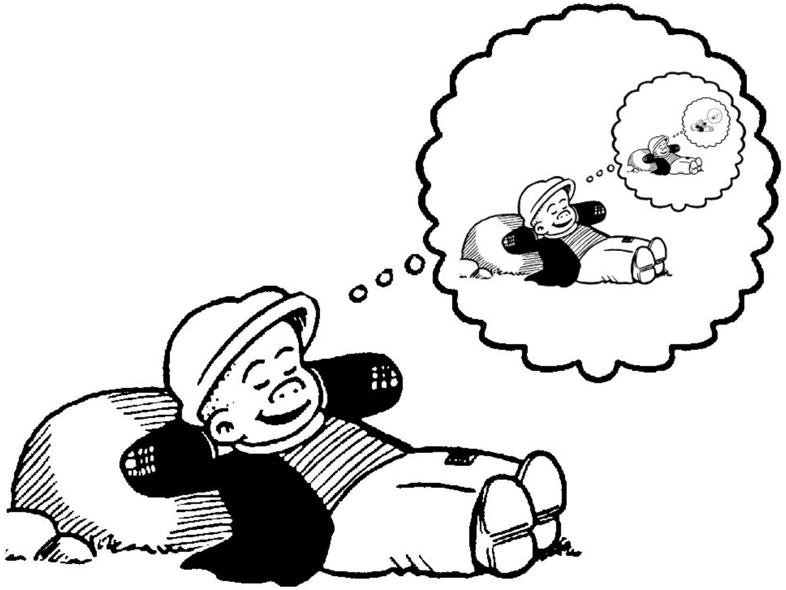
```
int addTwo(int a, int b){
	int temp = a + b;
	return sub;
}

int doSomething(int x, int y) {
	int a = addTwo(x, y);
	int b = addTwo(y, x);
	return a + b;
}
```

MIPS assembly code

```
addTwo:
	add $v0, $a0, $a1
	jr $ra
doSomething:
	# save
	addi $sp, $sp, -16
	sw $s0, 0($sp)
	sw $s1, 4($sp)
	sw $s2, 8($sp)
	sw $ra, 12($sp)
	add $s0, $a0, $0
	add $s1, $a1, $0
	jal addTwo
```

```
	add $s2, $v0, $0
	add $a0, $s1, $0
	add $a1, $s0, $0
	jal addTwo
	add $v0, $v0, $s2
	# restore
	lw $ra, 12($sp)
	lw $s2, 8($sp)
	lw $s1, 4($sp)
	lw $s0, 0($sp)
	addi $sp, $sp, 16
	jr $ra
```

Recursion



MIPS assembly code

```
# What should I map n to?

# ? words in the stack
```

```
//sum of all numbers from 0 to n
int f2(int n) {
	if(n <= 1)
		return 1;
	else
		return(n * f2(n-1));
}
```

MIPS assembly code

Recursion

```
int f2(int n) {
if(n <= 1)
return 1;
else
return(n * f2(n-1));
}
```

```
f2:
        addi  $sp, $sp, -8
        sw $a0, 4($sp)
        sw $ra, 0($sp)
        addi $t0, $0, 2
        slt $t0, $a0, $t0
        beq $t0, $0, else
        addi $v0, $0, 1
        addi $sp, $sp, 8
        jr $ra
```

```
else:
        addi $a0, $a0, -1
        jal f2
        lw $ra, 0($sp)
        lw $a0, 4($sp)
        addi $sp, $sp, 8
        mul $v0, $a0, $v0
        jr $ra
```

Recap

```
// High level code
void main() {
int y;
y = sum(42, 7);
...
}

int sum(int a, int b) {
 return(a + b);
}
```

- The jal instruction is used to jump to the procedure and save the current PC (+4) into the return address register
- Arguments are passed in $a0-$a3; return values in $v0-$v1
- Since the callee may over-write the caller’s registers
  - relevant values may have to be copied into memory
- Each procedure may also require memory space for local variables
- A stack is used to organize the memory needs for each procedure

Recap

```
// High level code
void main() {
int y;
y = sum(42, 7);
...
}

int sum(int a, int b) {
 return(a + b);
}
```

Steps for the code that is calling a function: Caller

- Before Entry:
  - Step 1: Pass the arguments:
    - The first four arguments (arg0-arg3) are passed in registers $a0-$a3
    - Remaining arguments are pushed onto the stack
  - Step 2: Save caller-saved registers
    - Save registers $s0-$s7 if they contain values you need to remember
  - Step 3: Save $ra, $fp if necessary
  - Step 4: Execute a jal instruction
    - Jump and Link will save your return address in the $ra register
    - so that you can return to where you started
- After Return:
  - Step 1: Restore $s0-$s7, $ra, $fp if saved

Recap

```
// High level code
void main() {
int y;
y = sum(42, 7);
...
}

int sum(int a, int b) {
 return(a + b);
}
```

Steps for the code that is calling a function: Callee

- On Entry:
  - Step 1: Save callee-saved registers
    - Save registers $s0-$s7 if they are used in the callee procedure
- Before Exit:
  - Step 1: Save return values in $v0-$v1 registers
  - Step 2: Restore $s0-$s7 registers, if they were saved
  - Step 3: Execute jr $ra

Arrays vs Pointers

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

MIPS assembly code

```
# $a0 = array[], $a1 = size,  $t0 = i
```

Arrays vs Pointers

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

MIPS assembly code

```
# $a0 = array[], $a1 = size,  $t0 = i
clear1:
        move $t0, $zero      # i = 0
loop1:
        sll $t1, $t0, 2      # $t1 = i * 4
        add $t2, $a0, $t1    # $t2 = &array[i]
        sw $zero, 0($t2)     # array[i] = 0
        addi $t0, $t0, 1     # i = i + 1
        slt $t3, $t0, $a1    # $t3 = (i < size)
        bne $t3, $zero, loop1
        # if (i < size) goto loop1
        jr $ra               # return to calling routine
```

Arrays vs Pointers

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

MIPS assembly code

$a0 = *array, $a1 = size, i in $t0 = p (address of array[0])

Arrays vs **Pointers**

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

MIPS assembly code

```
# $a0 = *array, $a1 = size, i in $t0 = p
(address of array[0])
clear2:
        move $t0, $a0 # p = & array[0]
        sll $t1, $a1, 2   # $t1 = size * 4
        add $t2, $a0, $t1 # $t2 = &array[size]
loop2:
        sw $zero, 0($t0) # Memory[p] = 0
        addi $t0, $t0, 4 # p = p + 4
        slt $t3, $t0, $t2 # $t3=(p<&array[size])
        bne $t3, $zero, loop2
        # if (p<&array[size]) goto loop2
        jr $ra  # return to calling routine
```

Arrays vs Pointers

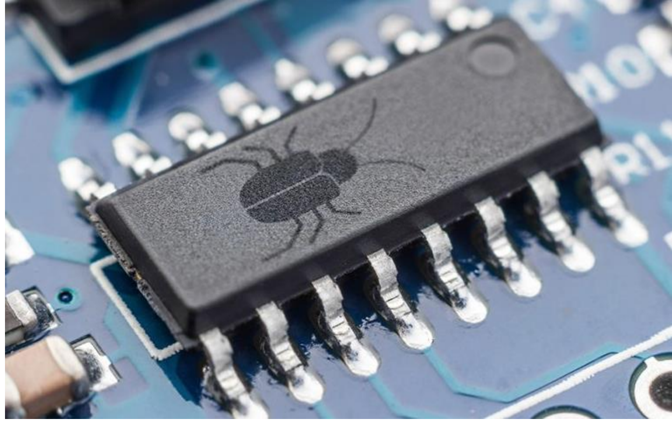
```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

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

```
       move $t0,$zero   # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
       add $t2,$a0,$t1  # $t2 = &array[i]
       sw $zero, 0($t2) # array[i] = 0
       addi $t0,$t0,1   # i = i + 1
       slt $t3,$t0,$a1  # $t3 = (i < size)
       bne $t3,$zero,loop1 # if (…) goto loop1
```

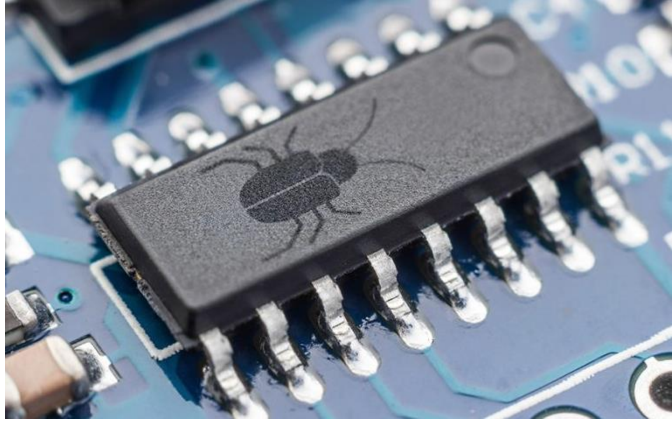
```
       move $t0,$a0     # p = & array[0]
       sll $t1,$a1,2    # $t1 = size * 4
       add $t2,$a0,$t1 # $t2 = &array[size]
loop2: sw $zero,0($t0) # Memory[p] = 0
       addi $t0,$t0,4   # p = p + 4
       slt $t3,$t0,$t2 # $t3 = (p<&array[size])
       bne $t3,$zero,loop2 # if (…) goto loop2
```

Fix the Bugs (?)



```
# copy words from the address in register $a0
# to the address in register $a1
# counting the number of words copied in register $v0
# stop copying when a word equals to 0.
# this terminating word should be copied but not counted

        addi $v0, $zero, 0     # Initialize count
loop:
        lw, $v1, 0($a0)        # Read next word from source
        sw $v1, 0($a1)         # Write to destination
        addi $a0, $a0, 4       # Advance pointer to next source
        addi $a1, $a1, 4       # Advance pointer to next destination
        beq $v1, $zero, loop # Loop if word copied != zero
```

Fix the Bugs (?)



```
# copy words from the address in register $a0
# to the address in register $a1
# counting the number of words copied in register $v0
# stop copying when a word equals to 0.
# this terminating word should be copied but not counted

        addi $v0, $zero, 0        # Initialize count
loop:
        lw, $v1, 0($a0)           # Read next word from source
        sw $v1, 0($a1)            # Write to destination
        addi $a0, $a0, 4          # Advance pointer to next source
        addi $a1, $a1, 4          # Advance pointer to next destination
        beq $v1, $zero, loop      # Loop if word copied != zero
```

new version

```
# copy words from the address in register $a0
# to the address in register $a1
# counting the number of words copied in register $v0
# stop copying when a word equals to 0.
# this terminating word should be copied but not counted

        addi $v0, $zero, -1    # Initialize count
loop:
        lw, $v1, 0($a0)        # Read next word from source
        addi $v0, $v0, 1       # Increment count words copied
        sw $v1, 0($a1)         # Write to destination
        addi $a0, $a0, 4       # Advance pointer to next source
        addi $a1, $a1, 4       # Advance pointer to next destination
        bne $v1, $zero, loop # Loop if word copied != zero
```

Pseudoinstruction

- The assembler will build instruction sequences for you
- “macros” of other actual instructions

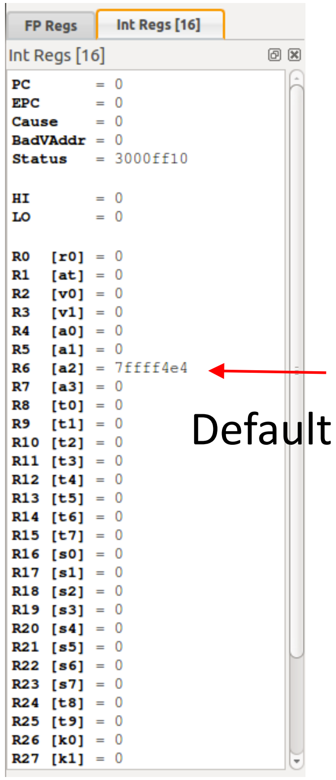
Pseudoinstruction	MIPS Instructions
clear $t0	add $t0, $0, $0
move $s1, $s2	add $s2, $s1, $0
`blt $t0, $t1, fexit`	`slt $at, $t0, $t1` `bne $at, $zero, fexit`
`ble $t0, $t1, fexit`	`slt $at, $t1, $t0` `beq $at, $zero, fexit`
`li $s0, 0x1234AA77`	`lui $s0, 0x1234` `ori $s0, 0xAA77`

SPIM simulator

- **QtSpim** **http://spimsimulator.sourceforge.net/**
- Any text editor to write source code (atom, notepad++,…)
- Tutorial
  - https://ecs-network.serv.pacific.edu/ecpe-170/tutorials/qtspim-tutorial
  - Check “mips-example-programs” section
- MIPS Reference Data Card
  - https://www.comp.nus.edu.sg/~cs2100/lect/MIPS_Reference_Data_page1.pdf

Layout

Register Panel



Default view Hex

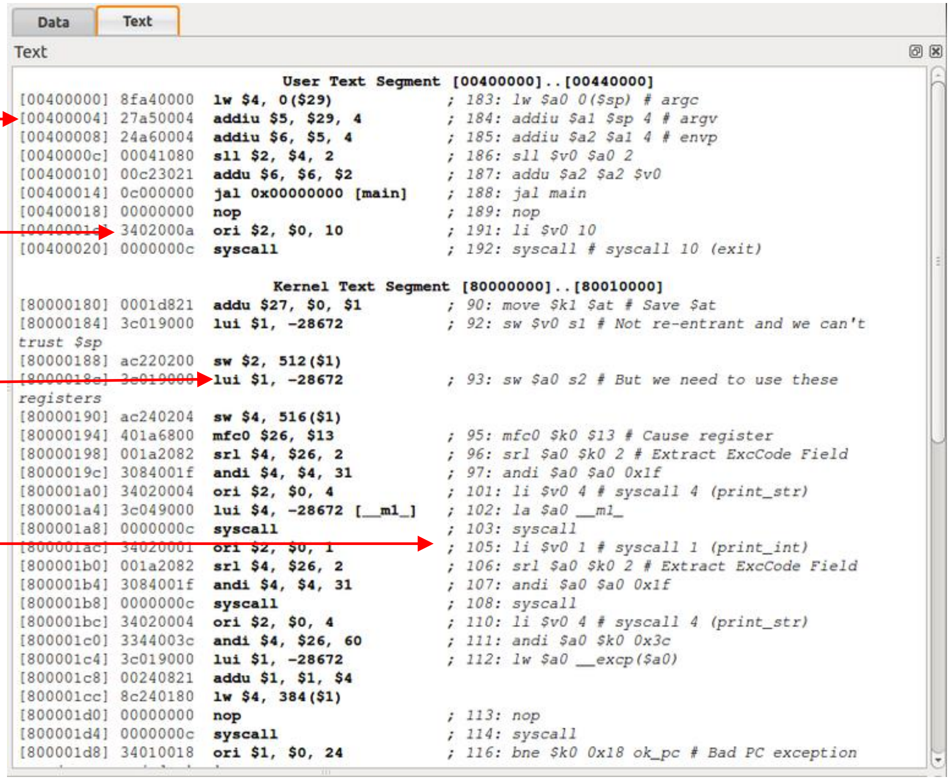
Memory Panel

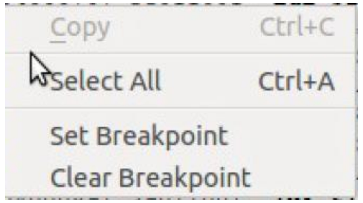
memory address

OpCodes (actual MIPS instructions)

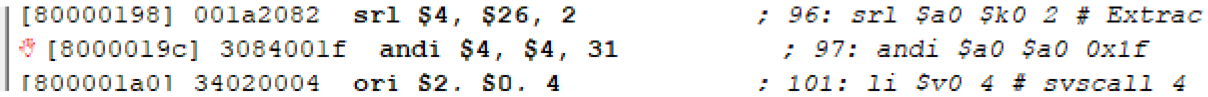
human-readable

what I wrote





Breakpoint – single step



Clear Registers – Reinitialize simulator

Typical Program Layout

#comments

. assembler directives

Define start & end of data declarations

Program Code **.text**

Data Declarations **.data**

```
variableName: .datatype initialValue

val: .word 1234
myval: .byte 5
myarray: .word 13, 34, 16

message: .asciiz "Hello
World\n"
```

```
# A demonstration of some simple MIPS instructions
# used to test QtSPIM

    # Declare main as a global function
    .globl main

    # All program code is placed after the
    # .text assembler directive
    .text

# The label 'main' represents the starting point
main:
    li $t2, 25        # Load immediate value (25)
    lw $t3, value         # Load the word stored in value (see bottom)
    add $t4, $t2, $t3    # Add
    sub $t5, $t2, $t3    # Subtract
    sw $t5, Z         #Store the answer in Z (declared at the bottom)

    # Exit the program by means of a syscall.
    # There are many syscalls - pick the desired one
    # by placing its code in $v0. The code for exit is "10"
    li $v0, 10 # Sets $v0 to "10" to select exit syscall
    syscall # Exit

    # All memory structures are placed after the
    # .data assembler directive
    .data

    # The .word assembler directive reserves space
    # in memory for a single 4-byte word (or multiple 4-byte words)
    # and assigns that memory location an initial value
    # (or a comma separated list of initial values)
value:  .word 12
Z:   .word 0
```

Syscall Interface

- $v0 – what action to take
- $a0 – $a3 parameters of that action

```
# Print string msg1

li        $v0, 4
la        $a0, msg1
syscall

msg1: .asciiz “give num"
```

```
# Get & save input  from user

li        $v0, 5

syscall

move      $t0, $v0
```

```
# Exit

li        $v0, 10

syscall
```

What Did We Learn?

- How to translate common programming constructs
- Stack / Procedures
- SPIM simulator

Thank You

Questions ?