LECTURE 17: MIPS (LAB 11, 12)

Computer Systems and Networks

Dr. Pallipuram
(vpallipuramkrishnamani@pacific.edu)

University of the Pacific

You can do almost everything just using these

Arithmetic Instructions

```
add <destination register>, <register 1>, <register 2>
sub <destination register>, <register 1>, <register 2>
mul <destination register>, <register 1>, <register 2>
addi <destination register>, <register 1>, value
```

Branching Instructions

```
beq <register 1>, <register 2>, label
bgt <register 1>, <register 2>, label
blt <register 1>, <register 2>, label
ble <register 1>, <register 2>, label
bge <register 1>, <register 2>, label
```

Memory Instructions

```
la <register>, memory  lw/sw <register>, offset(base)
```

Functions

The Program Counter

Instructions are stored in memory sequentially

Each MIPS32 instruction occupies 4 bytes

How does the processor know from where to fetch the next instruction?

- A special 32-bit register called Program Counter (`PC`) holds the address of the next instruction

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

PC →

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr. 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Reverse engineer: Write a C code for this assembly

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

PC →

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr. 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

PC →

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr. 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr. 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

PC →

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	addi $t0,$zero,0
	8	addi $t1,$zero, 2
	12	bge $t0, $t1, <label to addr. 24>
	16	addi $t0, $t0, 1
PC →	20	j <label to addr. 12>
	24	li $v0, 10
	28	syscall

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	addi $t0,$zero,0
	8	addi $t1,$zero, 2
PC →	12	bge $t0, $t1, <label to addr. 24>
	16	addi $t0, $t0, 1
	20	j <label to addr. 12>
	24	li $v0, 10
	28	syscall

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	`addi $t0,$zero,0`
	8	`addi $t1,$zero, 2`
	12	`bge $t0, $t1, <label to addr. 24>`
PC →	16	`addi $t0, $t0, 1`
	20	`j <label to addr. 12>`
	24	`li $v0, 10`
	28	`syscall`

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	`addi $t0,$zero,0`
	8	`addi $t1,$zero, 2`
	12	`bge $t0, $t1, <label to addr. 24>`
	16	`addi $t0, $t0, 1`
PC →	20	`j <label to addr. 12>`
	24	`li $v0, 10`
	28	`syscall`

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	`addi $t0,$zero,0`
	8	`addi $t1,$zero, 2`
PC →	12	`bge $t0, $t1, <label to addr. 24>`
	16	`addi $t0, $t0, 1`
	20	`j <label to addr. 12>`
	24	`li $v0, 10`
	28	`syscall`

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

	Address	Instruction
	4	addi $t0,$zero,0
	8	addi $t1,$zero, 2
	12	bge $t0, $t1, <label to addr. 24>
	16	addi $t0, $t0, 1
	20	j <label to addr. 12>
PC →	24	li $v0, 10
	28	syscall

Program Execution – Program Counter (PC)

Instructions are stored in memory and each occupy 4 bytes.

Address	Instruction
4	`addi $t0,$zero,0`
8	`addi $t1,$zero, 2`
12	`bge $t0, $t1, <label to addr. 24>`
16	`addi $t0, $t0, 1`
20	`j <label to addr. 12>`
24	`li $v0, 10`
28	`syscall`

PC →

Functions in MIPS

Basic Functions in MIPS

1. Program saves the context (registers) of calling function (caller)

2. Program saves the arguments in registers (`$a0 - $a3`)

3. Program calls the callee via jump-and-link instruction

```
    jal <function label>
```

`jal` saves the address of the next instruction in return address reg., `$ra`

Program Counter (PC) points to the callee's location. Callee saves return values in regs. `$v0-$v1`

4. Callee returns via jump register instruction,

```
        jr <register name> #usually $ra
```

`jr` sets PC to `$ra`. PC continues there onwards

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

PC →

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

`$ra=12`

PC → 8

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

`$ra=12`

PC

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

`$ra=12`

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

PC → 24

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

`$ra=12`

PC →

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

Function Execution

Instructions are stored in memory and each occupy 4 bytes.

`$ra=12`

PC →

Address	Instruction
4	`addi $a0, $zero, 5 #argument 5`
8	`jal <function at 20>`
12	`li $v0, 10`
16	`syscall`
20 function:	`add $v0, $a0, $a0  #return value v0`
24	`jr $ra`
28	

More Jumps

Jump and Link
(side effect: `$ra` stores address of next instruction)

```
jal <destination>
```

Use this to *call* a function!

Jump Register
(destination address is stored in <reg1>

```
jr <reg1>
```

Use this to *return from* a function!

Problem 1: Write Code

```
#include <stdio.h>

int function(int a);

int main()
{
  int x=5;
  int y;

  y = function(x);
  printf(“y:%d”,y);
  return 0;
}

int function(int a)
{
  return 3*a+5;
}
```

Place arguments in `$a0-$a3`

Place return values in `$v0-$v1`

Return address saved automatically in `$ra`

Ignore the stack for this example. (Thus, the function will destroy registers used by calling function)

What if…

Callee needs some of the registers (`$s0 - $s9`) to compute and these were already in use by the caller?

Callee calls another function, overwriting the return address, `$ra`?

Stack to the rescue!

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

$sp →

Address	Value
20	
16	
12	
8	
4	

2 3 4 5 →

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

push 5

2 3 4 →

$sp →

Address	Value
20	5
16	
12	
8	
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

push 4

2 3 →

$sp →

Address	Value
20	5
16	4
12	
8	
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

push 3

2 →

$sp →

Address	Value
20	5
16	4
12	3
8	
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

push 2

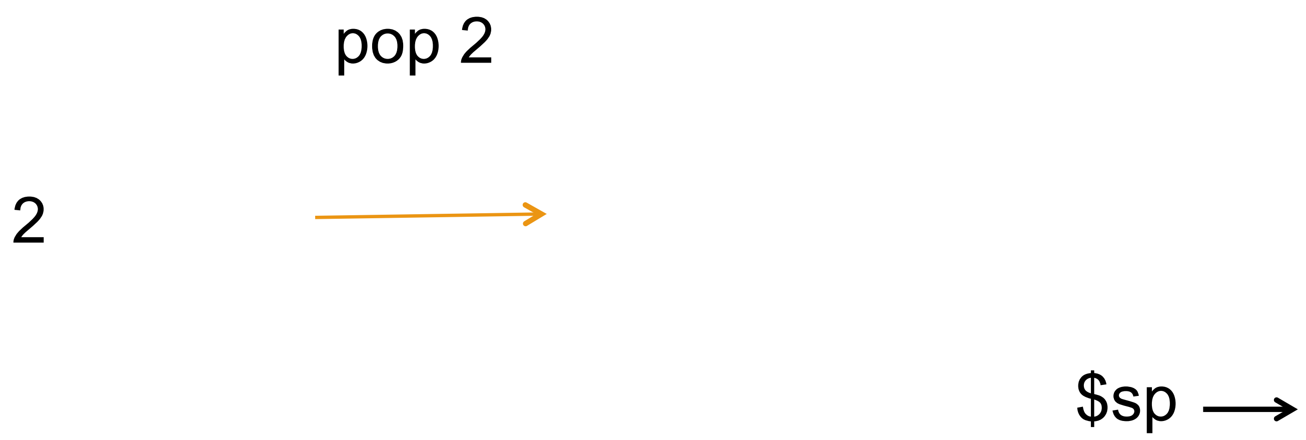
→

$sp →

Address	Value
20	5
16	4
12	3
8	2
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure


pop 2

2

$sp

Address	Value
20	5
16	4
12	3
8	2
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

pop 3

2 3 →

$sp →

Address	Value
20	5
16	4
12	3
8	2
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

pop 4

2 3 4 →

$sp →

Address	Value
20	5
16	4
12	3
8	2
4	

How Stack Operates

Stack is a Last In, First Out (LIFO) data structure

pop 5

2 3 4 5 →

$sp →

Address	Value
20	5
16	4
12	3
8	2
4	

Problem 2: Using $sp, write the set of commands for pushing and popping a register value (say $s0)

What a caller must do with the Stack prior to function call?

Must use the stack if:

it wants to store temporary registers (`$t0-$t9`) or its argument registers (`$a0-$a3`) onto the stack. This is done before calling another function

it wants to pass arguments via stack. For our purposes, `a` registers should suffice

After return, it should pop the stack

What a callee must do with the stack?

1. Push `$s` registers onto the stack, so that it does not overwrite the caller’s data

2. Push `$ra` onto the stack because a callee may call another function, overwriting the return address.

3. Do function stuff

4. Pop `$ra` from the stack

5. Pop `$s` registers from the stack

Caller and Callee MIPS portion

Caller

```
<some code>
<push t and a regs. in use>
<pass args using a regs>
jal callee
<pop t and a regs.>
<some code>
```

Callee

```
<push s regs. used by caller>
<push ra>
<some code>
<pop ra>
<pop s regs. used by caller>
<save return values in v regs>
jr $ra
```

Problem 3: Convert this to MIPS

```
int array[] = {2, 3, 4, 5, 6};
int main() {
int num, position;
scanf(“%d”,&num);
position = search(array, num, 5);
printf(“\n The position is: %d”,position);
}
int search(int *array, int num, int size)
{
        int position =-1;
        for(int i=0;i<size;i++)
                if(array[i]==num)
                {       position=i;
                        break;}
        return position;
}
```

Register map:

```
$s0:num
$s1: position
$a0: array addr.
$a1: num
$a2: size
$v0: return val.
```

Aggressive context saving

As your code gets larger, it may be too difficult to keep track of registers in use

Do not want to remember too much?

- Have the caller save all of the `t` and `a` registers!
- Have the callee save all of the `s` and `r` registers!

Pro: guaranteed to work, **if** implemented correctly

Con: longer program footprint. OK for our programs

Aggressive context saving
Caller and Callee MIPS portion

Caller Portion

```
<some code>

<aggressively push t and a regs>

<pass args in a regs>

jal callee

<aggressively pop a and t regs.>

<some code>
```

Callee Portion

```
Callee:

<aggressively push s regs.>

<push ra>

<callee code>

<pop ra>

<aggressively pop s regs.>

<save return values in v regs>

jr $ra
```

Some tips if you want to perform Aggressive Saving

Create a text file that contains stub for:

- Aggressive pushing and popping of `t,a`-registers. Use it for the caller portion
- Aggressive saving and popping of `s,ra`-registers. Use it for the callee portion
- Copy and paste and have fun!