Computer Architecture

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr
nizamettinaydin@gmail.com

http://www.yildiz.edu.tr/~naydin

MIPS Instruction Set-II

Outline

- **MIPS Instruction Set-II**
  - Control Flow Instructions
    - Control Flow
    - Specifying Branch Destinations
    - Compiling If-Else Statements
  - Unconditonal Jump
  - Branch Instruction Design
  - For Loop
  - Procedure Call
    - Procedure Call Instructions
  - MIPS Register Usage Convention
  - Temporary and Saved Registers
  - Stack allocation in MIPS
  - Storage Classes
  - Memory Layout

Control Flow Instructions

- What are control flow statements in a programming language?
  - Loops:
    - Do, For, While
  - If then else
  - Case and Switch Statements
  - Function Calls
  - Goto, Labels (not recommended)
  - Return Statement

Control Flow

- The kinds of control flow statements supported by different languages vary, but can be categorized by their effect:
  - continuation at a different statement
    - unconditional branch or jump,
  - executing a set of statements only if some condition is met
    - choice - i.e., conditional branch,
  - executing a set of statements zero or more times, until some condition is met
    - i.e., loop - the same as conditional branch,
  - executing a set of distant statements, after which the flow of control usually returns
    - subroutines or functions,
  - stopping the program, preventing any further execution
    - unconditional halt.

MIPS Control Flow Instructions

- MIPS conditional branch instructions (I format):

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

- Branch to a labeled instruction if a condition is met
  - Otherwise, continue sequentially
- Example: if (i==j) h = i + j;

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

Specifying Branch Destinations

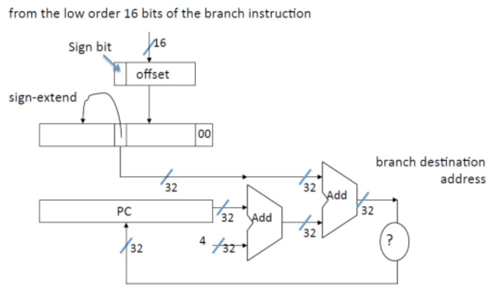
bne $s0, $s1, Lbl

0x05	16	17	16 bit offset

- How is the branch destination address specified?
- 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
    - PC gets updated to (PC + 4 + offset) if the branch is taken
  - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

7

Specifying Branch Destinations



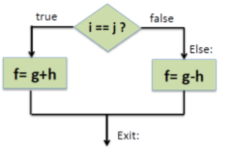
8

Compiling If-Else Statements

- C code:

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

  - f, g, h, i, j are in $s0, $s1, …



- Compiled MIPS code:

```
        bne  $s3, $s4, Else
        add  $s0, $s1, $s2
        j    Exit
Else:   sub  $s0, $s1, $s2
Exit:   …
```

unconditional jump to instruction labeled Exit

Assembler calculates addresses

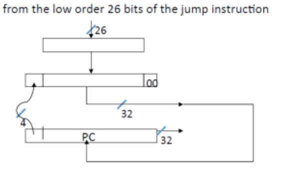
9

Unconditonal Jump (J Format)

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

  j label #go to label

0x02	26-bit Address



10

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:     …
```

11

Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions are penalized!
- beq and bne are the common case
  - Use in combination with beq, bne with slt
- This is a good design compromise

12

Set on Less Than (slt)

- Use in combination with beq, bne with slt

```
slt   $t0,  $s1,  $s2     # if ($s1 < $s2)
bne   $t0,  $zero, L      # branch to L
```

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- slt rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;

13

For Loop

for (j = 0; j < 10; j++)
    a = a + j;

This is not correct since the loop bound is not j!=10 but j< 10

#assume s0 == j; s1 == a; t0 == temp;

Instructions					Comments
	addi	$s0,	$zero,	0	#j = 0 + 0
	addi	$t0,	$zero,	10	#temp = 0 + 10
Loop:	beq	$s0,	$t0,	Exit	#if (j == temp)goto Exit
	add	$s1,	$s1,	$s0	#a = a + j
	addi	$s0,	$s0,	1	#j = j + 1
	j	Loop			#goto Loop
Exit: …					#exit from loop & cont.

14

For Loop

for (j = 0; j < 10; j++)
    a = a + j;

#assume s0 == j; s1 == a; t0 == temp;

Instructions					Comments
	addi	$s0,	$zero,	0	#j = 0 + 0
	addi	$t0,	$zero,	10	#temp = 0 + 10
Loop:	slt	$t0,	$s0,	$t1	#if (j < n)
	beq	$t0,	$zero,	Exit	
	add	$s1,	$s1,	$s0	#a = a + j
	addi	$s0,	$s0,	1	#j = j + 1
	j	Loop			#goto Loop
Exit: …					#exit from loop & cont.

15

Procedure Call

caller

```
char s;
int num = 5;
….
s=myfunction(num);
….
```

callee

```
char myfunction(int number)
 {
     char selection[] =
{'S','M','T','W','T','F','S'};
     return selection[number];
 }
```

- Procedure P calls procedure Q, and Q then executes and returns back to P
  - Passing Control:
    - The PC must be set to the starting address of the code for Q upon entry and then set to the instruction in P following the call to Q upon return
  - Passing Data:
    - P must be able to provide one or more parameters to Q and Q must be able to return a value back to P
  - Allocating and Deallocating memory:
    - Q may need to allocate space for local variables when it begins and then free that storage before it returns.

16

Procedure Call

- The execution of a procedure
  - Place parameters in a place where the procedure can access
  - Transfer control to the procedure
  - Acquire the storage resources needed for the procedure
  - Perform the desired task
  - Place the result value in a place where the calling program can access
  - Return control to the point of origin

17

Procedure Call Instructions

- Procedure call:
  - jump and link

    jal ProcedureLabel
    - Address of following instructon put in $ra
    - Jumps to target address
- Procedure return: jump register

    jr $ra
    - Copies $ra to program counter

18

MIPS Register Usage Convention

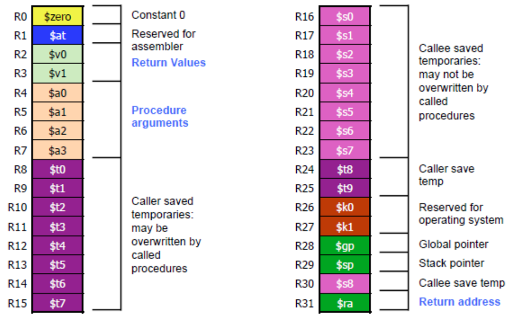
- $a0-$a3 :
  - four argument registers in which to pass parameters
- $v0-$v1:
  - two value registers in which to return values
- $ra:
  - one return address register to return to the point of origin
- At the end of the procedure we jump back to the $ra (an unconditional jump)

  jr $ra #jump register
- The jump-and-link instruction (jal) :
  - jumps to an address and simultaneously saves the address of the following instruction (PC + 4) in register $ra

  jal ProcedureAddress

19

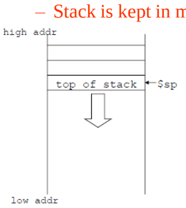
MIPS Register Conventions



20

Spilling Registers

- What if the callee needs more than 4 arguments?
- What happens to the content of the register file?
  - callee uses a sotware stack
    - a last-in-first-out queue
  - Stack is kept in memory



- One of the general registers, $sp ($29), 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

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

21

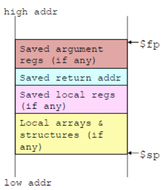
Temporary and Saved Registers

- Temporary registers $t0 through $t9 can also be used as by MIPS convention they are not preserved by the callee across subroutine boundaries
  - i.e., if the caller must first save it if it concerns that it may lose its content
- However, saved registers $s0 through $s7 must be preserved by the callee
  - 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

22

Allocating Space on the Stack

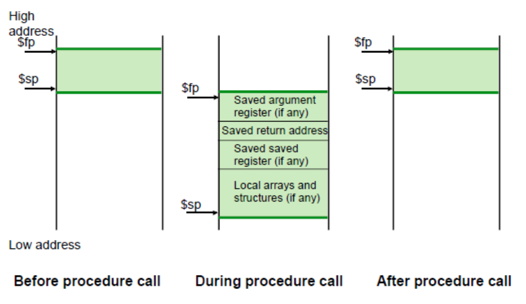
- The segment of the stack containing a procedure's saved registers and local variables in its procedure frame (a.k.a. activation record)



- The frame pointer ($fp) points to the first word of the frame of a procedure
  - providing a stable base register for the procedure
- $fp is initialized using $sp on a call and $sp is restored using $fp on a return

23

Stack allocation in MIPS



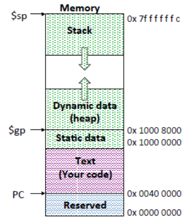
24

Storage Classes

- Storage classes
  - Variables that are local to a procedure and are discarded when the procedure exits
  - Variables that exist across procedures are kept in static memory.
- To simplify access to static data MIPS uses global pointer ($gp)

25

Memory Layout



- Text
  - program code
- Static data
  - global variables
    - – e.g., static variables in C, constant arrays and strings
    - $gp initialized to address allowing ±offsets into this segment
- Stack:
  - automatic storage
- Dynamic data segment (a.k.a. heap)
  - for structures that grow and shrink (e.g., linked lists)
    - Allocate space on the heap with malloc() and free it with free() in C

26

Leaf Procedure Example

- C code:

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

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

27

Leaf Procedure

- MIPS code:

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

leaf_example:

addi	$sp,	$sp,	-4	
sw	$s0,	0($sp)		Save $s0 on stack
add	$t0,	$a0,	$a1	
add	$t1,	$a2,	$a3	Procedure body
sub	$s0,	$t0,	$t1	
add	$v0,	$s0,	$zero	Result on the return value
lw	$s0,	0($sp)		Restore stack
addi	$sp,	$sp,	4	
jr	$ra			Return

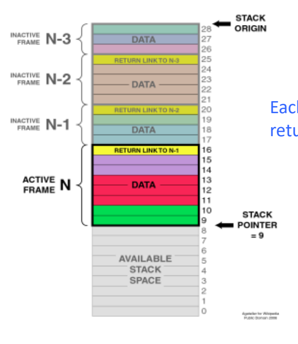
28

Non-Leaf Procedures

- Procedures that call other procedures
- For nested calls, need to save the data on the stack:
  - Return address of the procedure
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call
- Recursive functions are optimized to prevent stack overflow.

29

Non-Leaf Procedures



Each function (callee) has a return link to the caller

30

Non-Leaf Procedure Example - Recursion

- C code:

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

  - Argument n in $a0
  - Result in $v0

31

Non-Leaf Procedure Example - Recursion

- MIPS code:
-

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

```
fact:
    addi    $sp,    $sp,    -8      # adjust stack for 2 items
    sw      $ra,    4($sp)          # save the return address
    sw      $a0,    0($sp)          # save the argument n
    slti    $t0,    $a0,    1       # test for n<1
    beq     $t0,    $zero,  Else    # if n>=1, goto Else
    addi    $v0,    $zero,  1       # return 1
    addi    $sp,    $sp,    8       # pop 2 items off stack
    jr      $ra                     # return to after jal
Else:
    addi    $a0,    $a0,    -1      # n>=1: argument gets (n-1)
    jal     fact                    # call fact with (n-1)
    lw      $a0,    0($sp)          # return from jal: restore argument n
    lw      $ra,    4($sp)          # restore the return address
    addi    $sp,    $sp,    8       # adjust stack pointer to pop 2 items
    mul     $v0,    $a0,    $v0     # return n*fact(n-1)
    jr      $ra
```

32

Procedure Calls in MIPS (Summary)

- The caller passes arguments to the callee by placing the values into the argument registers $a0-$a3.
- The caller calls jal followed by the label of the subroutine.
  - This saves the return address in $ra.
    - The return address is PC + 4, where PC is the address of the jal instruction
- The callee starts by pushing any registers it needs to save on the stack.
- If the callee calls a another subroutine, then it must push $ra on the stack.
  - It may need to push temporary registers as well.
  - Once the subroutine is complete, the return value is place in $v0-$v1.
- The callee then calls jr $ra to return back to the caller.

33